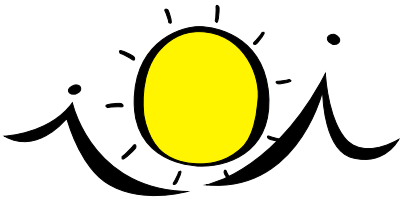


Nowy Sącz, POLAND 2005
The 17-th International
Olympiad in Informatics

IOI'2005

Tasks and Solutions

Nowy Sącz, 2005

Authors:

Szymon Acedański
Piotr Chrząstowski
Mathias Hiron
Łukasz Kowalik
Marcin Kubica
Tomasz Malesiński
Anna Niewiarowska
Krzysztof Onak
Pavel Pankov
Arkadiusz Paterek
Jakub Pawlewicz
Jakub Radoszewski
Piotr Stańczyk
Marcin Stefaniak
Tom Verhoeff
Szymon Wąsik

Cover:

Wojciech Rygielski

Typesetting:

Tomasz Waleń

Proofreaders:

Szymon Acedański
Marcin Kubica
Tomasz Waleń

Volume editor:

Marcin Kubica

ISBN 83–917700–9–5

Contents

Preface

The International Olympiad in Informatics (IOI) is an annual competition for talented students from secondary education all over the world. This booklet contains the main material of the IOI 2005 competition.

The IOI 2005 competition was prepared and executed by the Polish Olympiad in Informatics and the Institute of Informatics of Warsaw University. IOI 2005 took place in Nowy Sącz, Poland, on the campus of the Nowy Sącz School of Business — National Louis University, from 18 through 25 August 2005. There were two competition days preceded with one practice day, with three competition tasks on each day. Over 280 contestants from 72 countries participated.

This booklet describes the preparation of the IOI 2005 competition, the tasks, the theory behind various solution approaches. We hope that this booklet will prove to be useful to both organizers and participants of future IOIs. Additional materials, including test data, are available at: `http://www.ioi2005.pl`.

Tom Verhoeff, ISC Chair
Marcin Kubica, HSC Chair

Nowy Sącz, Poland, 22 August 2005

Tom Verhoeff, Marcin Kubica

Introduction

It may be interesting to know a little bit about the preparation and execution of the IOI 2005 competition.

Task preparations started over a year before the actual competition. The IOI 2005 Host Scientific Committee (HSC) sent out a Call for Tasks in October 2004. All externally submitted tasks have been collected until the end of the year. In response, eight task proposals have been received. Additionally, HSC have prepared another seven tasks.

IOI competition tasks must satisfy, among others, the following requirements:

- they must be of an algorithmic nature,
- solutions must be implementable in Pascal, C, or C++,
- there should exist a variety of possible solutions, differing in difficulty and efficiency.

Collected proposals were further refined by the HSC, and 15 of the resulting tasks were reviewed by the IOI Scientific Committee (ISC) in April 2005. The ISC selected 12 potential tasks for IOI 2005. Three of them were used as practice tasks, six were used in the competition, and three more were available as backup tasks, in case some irreparable problem did come to light after presentation to the General Assembly.

HSC have developed many programmed solutions in all allowed programming languages, extensive test data for all the selected tasks, and appropriate time and memory limitations. This effort is necessary to distinguish various solutions.

The evaluation of the programs submitted by the contestants during the competition is based on a batch of carefully designed test cases, each of which involves running the submitted program one or more times with different input data. These test cases probe both the correctness of the submitted program and its time and memory efficiency. The number of test cases varied between 20 and 24, depending on the task. The maximum score for each task was 100 points. Correct but less efficient programs could obtain partial scores.

The IOI 2005 competition made use of the System for Internet Olympiads (SIO) developed by students of the Institute of Informatics at Warsaw

University. SIO is the fourth generation of grading systems developed by Polish Olympiad in Informatics. They have been successfully used in various contests, also abroad, including: Polish Olympiad in Informatics, Baltic Olympiad in Informatics, Central European Olympiad in Informatics and ACM Central European Programming Contest. The current system was used for the first time at the Baltic Olympiad in Informatics in 2001, which was organized in Poland.

If you have a good idea for a competition task, please consider submitting it to IOI 2006.

See you next year in Mexico.

Tom Verhoeff, ISC Chair
Marcin Kubica, HSC Chair

Nowy Sącz, Poland, 22 August 2005

Tasks and Solutions

Krzysztof Onak
Szymon Acedański
Task idea and formulation

Jakub Pawlewicz
Marcin Stefaniak
Analysis

Available memory: 32 MB, Maximum running time: 1 s.

Divisor game

Alice and Bob invented a two-player game. At first, Alice chooses a positive integer k in the range from 1 to some fixed integer n. Then Bob asks questions of the form 'Is k divisible by m?', where m is a positive integer. Alice answers each such question with 'yes' or 'no'. Bob wants to know what number Alice bears in mind by asking as few questions as possible. Your task is to write a program, which plays the game as Bob.

Let us denote by $d(n)$ the minimal number of questions, which have to be asked to find k, regardless of what k Alice chooses (for given n). Your program's answer for a test case will be considered correct, if k is correctly determined using no more than $d(n)$ questions.

Library

Your program must use a special library to play the game. The library consists of the files: `alice.pas` *(for Pascal),* `alice.h` *and* `alice.c` *(for C/C++). The library provides the following functionality:*

- `function get_n: longint` / `int get_n()` — *Your program should call this function to initialize a game, before it calls any other function/procedure provided by the library. Function* `get_n` *returns n, the upper bound on the number that Alice has in mind. Number n satisfies the limitations $1 \leqslant n \leqslant 1\,000\,000$. Moreover, in 50 % of test cases n satisfies $1 \leqslant n \leqslant 10\,000$.*

- `function is_divisible_by(m: longint): boolean` / `int is_divisible_by(int m)` — *Your program may ask questions by calling this function. Function* `is_divisible_by` *returns* `True`/`1` *if the number k Alice has in mind is divisible by m. Otherwise it returns* `False`/`0`. *The parameter m must satisfy $1 \leqslant m \leqslant n$. Your program should ask as few questions as possible.*

- `procedure guess(k: longint)` / `void guess(int k)` — *To end the game your program should report the number k Alice has in mind, by calling the procedure* `guess(k)`. *The parameter k should satisfy*

$1 \leqslant k \leqslant n$. After calling this procedure your program will be terminated automatically.

If your program makes an illegal call, it will be terminated.

Your program should communicate only by means of the above functions and procedures. Your program must not read or write any files, it must not use standard input/output and it must not try to access any memory outside your program.

Compilation

If your program is written in Pascal, then you must include '`uses alice;`*' statement in your source code. To compile your program, use the following command:*

```
ppc386 -O2 -XS div.pas
```

If your program is written in C or C++, then you must include '`#include "alice.h"`*' statement in your source code. To compile your program, use one of the following commands:*

```
gcc -O2 -static div.c alice.c -lm
g++ -O2 -static div.cpp alice.c -lm
```

Experimentation

To let you experiment with your solution, you are given an example library playing as Alice: its sources are in `alice.pas`*,* `alice.h` *and* `alice.c` *files. When you run your program, it will be playing against this simple library. You can modify this library, but please do not change the interface part of it. Please remember, that during the evaluation your program will be playing against a different opponent.*

When you submit your program using the TEST interface[1]*, it will be compiled with the unmodified example opponent library. The submitted input file will be given to your program's standard input. The input file should consist of two lines, each containing one integer. The first line should contain number* n*, and the second line should contain number* k*.*

You are also provided with two simple programs illustrating usage of the library: `div.c` *and* `div.pas`*. (Please remember, that these programs are not correct solutions.)*

[1] TEST interface will be available during the trial session.

Sample interaction

Below there is a sample interaction between a program and the library.

Your program calls	**What happens**
`get_n()`	*returns* `1 000`
`is_divisible_by(10)`	*returns* `True`/`1`
`is_divisible_by(100)`	*returns* `True`/`1`
`is_divisible_by(1000)`	*returns* `False`/`0`
`is_divisible_by(200)`	*returns* `False`/`0`
`is_divisible_by(500)`	*returns* `False`/`0`
`is_divisible_by(300)`	*returns* `False`/`0`
`is_divisible_by(700)`	*returns* `False`/`0`
`guess(100)`	*Alice's secret number is 100. Your program wins and is terminated automatically.*

Solution

At first, let us consider a simpler version of the divisor game, in which Alice is restricted to choose as k either some prime number or 1. In such a game a good strategy is to check whether k is divisible by successive prime numbers. If we find a prime divisor p of k, then p must equal k. If k is not divisible by any prime $p \leqslant n$, then k must be 1. In the worst case we have to ask as many questions as there are prime numbers less than or equal n. Let us denote this number by $d(n)$. We will show that in the original game, in the worst case, we need at most $d(n)$ questions, too.

Model solution

In model solution, we test consecutive primes (in the ascending order) whether they divide k. If k is divisible by some prime p, we continue checking divisibility by p^2, p^3 and so on, until we get a negative answer. Moreover, we stop checking divisibility for successive powers of p as soon as a positive answer implies $k > n$. For example, for $n = 5$, if 2^2 divides k, then k is neither divisible by 2^3, nor 3, nor 5.

It turns out that using the above strategy, we will never ask more questions than $d(n)$. To prove this fact we will apply Chebyshev's theorem.

Theorem 1 *If m is a positive integer, then there exists at least one such prime p, that*

$$m+1 \leqslant p \leqslant 2 \cdot m$$

Now please note, that anytime we discover that k is divisible by p, we can reduce the problem of finding k not greater than n, to a smaller problem of finding such k' not greater than $\lfloor \frac{n}{p} \rfloor$, that $k = k' \cdot p$. Now observe, that $d(\lfloor \frac{n}{p} \rfloor)$ is strictly smaller than $d(n)$, because the interval $[\lfloor \frac{n}{p} \rfloor + 1, n]$ contains at least one prime. This follows from Chebyshev's theorem for the interval $[\lfloor \frac{n}{2} \rfloor + 1, 2\lfloor \frac{n}{2} \rfloor]$, which is a subinterval of $[\lfloor \frac{n}{p} \rfloor + 1, n]$, since $p \geqslant 2$. Therefore, by induction, we can find k' using not more than $d(\lfloor \frac{n}{p} \rfloor) \leqslant d(n) - 1$ questions. Moreover, all the negative answers obtained so far, are also negative while looking for k', so we do not need to ask them once again. So we need at most $d(n)$ questions to find k, because we have just asked one to obtain p.

To implement the above method, one should generate the list of primes not exceeding n. This can be done using the Eratosthenes sieve.

Alice's evil strategy

The library playing as Alice, used during the evaluation, does not establish the secret number at the beginning of the game. It adapts to Bob's questions and tries to fix k as late as possible. This way, the library is increasing the number of questions Bob has to ask to find k.

For each test case, the library is given an interval $[l, r]$ of possible values of k. For each question the library answers 'no' if only it is possible. It answers 'yes' only when the negative answer contradicts the previous answers or makes k exceed the interval $[l, r]$. Additionally, for each test case (excluding the sample test) the limit on number of questions is set to $d(n)$.

Piotr Chrząstowski,
Anna Niewiarowska
Task idea and formulation

Szymon Acedański,
Jakub Pawlewicz
Analysis

Available memory: 32 MB, Maximum running time: 1 s.

Domino

You are given a chessboard of size $n \times m$. There are also some lines drawn on it. Each line separates two adjacent fields. You are going to put dominoes on the chessboard. Each domino covers two adjacent fields. You can put a domino on two adjacent fields only if they are not separated by a line. Your task is to find such an arrangement of dominoes on the chessboard, that each field is covered by exactly one domino. You can assume that for each input data there is a solution.

Task

This is an output-only task. This means, that you are given input files `dom1.in`, `dom2.in`, *...,* `dom20.in`. *Your task is to produce output files* `dom1.out`, `dom2.out`, *...,* `dom20.out`. *Your submission should comprise a single* `zip` *or* `tar-gzip` *archive containing all the output files without any sub-directories. The format of input files, as well as the format of output files, which you are to produce, is described below.*

Input

The first line of an input file contains two integers, n and m, separated by a single space — n is the number of rows and m is the number of columns of the chessboard, $1 \leqslant n,m \leqslant 100$, $n \cdot m$ is even. The fields of the chessboard are numbered from 1 to $n \cdot m$. The i-th field (from the left) in the j-th row (from the top) has number $(j-1) \cdot m + i$ (for $1 \leqslant i \leqslant m$, $1 \leqslant j \leqslant n$).

The second line contains one positive integer l, $0 \leqslant l \leqslant 5000$. Each of the following l lines contains two integers separated by a single space. The $i+2$-nd line contains integers p_i and q_i (for $i = 1,\ldots,l$), where $1 \leqslant p_i, q_i \leqslant n \cdot m$, p_i and q_i are numbers of two adjacent fields. It represents a line between fields number p_i and q_i.

Output

A single output file should consist of $\frac{n \cdot m}{2}$ lines describing an arrangement of the dominoes, one domino per line. Each of these lines should contain two integers separated by a single space: numbers of two adjacent fields covered by a domino. The dominoes may be described in any order. If there are several solutions, you should find any one of them.

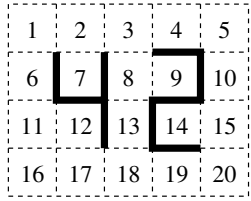
Example

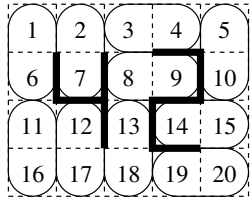
For the input data:

```
4 5
9
8 7
13 14
14 19
6 7
12 7
4 9
12 13
14 9
9 10
```

the correct result is:

```
3 4
1 6
2 7
8 9
5 10
14 15
11 16
12 17
13 18
19 20
```





Solution

Let $G = (V_1, V_2, E)$ be an undirected bipartite graph. A *matching* in this graph is such a set $M \subseteq E$, that no two edges of M have a common end. We say,

that vertex $v \in V$ is *matched* in M if v is an end of some edge in M. If v is not matched, then we say that it is *free*. Moreover, we say, that a matching M is *perfect*, if all vertices of G are matched.

The problem of covering a chessboard with dominoes can be reduced to a problem of finding a perfect matching in a bipartite graph $G = (V_1, V_2, E)$, where vertices represent the chessboard fields — white fields form V_1 and black fields form V_2. Two vertices are connected with an edge if they represent two adjacent fields not separated by a line (that is, both of them can be covered using one domino). Such a graph has $n \cdot m$ vertices and $O(n \cdot m)$ edges.

A matching M in such a graph represents an arrangement of dominoes on the chessboard: i–$j \in M$, if there is a domino on the chessboard covering both fields i and j. As in a correct matching, where each vertex belongs to at most one edge, in a correct covering each chessboard field must be covered by one domino. However, a matching does not necessarily represent the covering of the whole chessboard. In fact, we have to find a perfect matching. This guarantees, that each field is covered.

Backtracking solution

Quite simple, but very time-consuming method for finding a covering of the chessboard (or perfect matching in a graph) is some kind of backtracking. It can be implemented in the following way: we start recursion with the empty chessboard. Then, the recursive procedure performs the following steps:

1. Check whether the board is fully covered. If it is so, then write the solution and exit the program.
2. Pick any two free adjacent fields v and w not separated by a line.
3. Recursively search for a covering with new domino put on the fields v and w.
4. Recursively search for a covering with additional line drawn between fields v and w.

The running time of backtracking algorithms tend to depend strongly on applied optimizations. At first please note, that if there is an empty field on the chessboard, which is already completely surrounded by covered fields or drawn lines, then it is not possible to fully expand such covering. Therefore,

in such a situation we can truncate the backtracking and return from the recursive procedure immediately.

Another thing is to carefully design step 2. We will call a field *w*, adjacent to *v*, *available*, if *w* is not covered by a domino, and if *v* and *w* are not separated by a line. Of course, one can simply iterate over all fields on the chessboard and pick the first one, which is is adjacent to an available field. A better idea is to choose such *v* and *w*, that *v* has a minimal possible number of available adjacent fields. Using this heuristic, coverings which are not fully expandable, are often quickly detected and eliminated.

Model solution

The typical matching algorithm, which finds not only a perfect matching, but more generally — a *maximum* matching, is based on the idea of *augmenting paths*. A matching is called maximum if it contains the maximum possible number of edges. Let us consider a matching M in a bipartite graph $G = (V_1, V_2, E)$. We call a vertex *free* if it is not matched in M. An *augmenting path* in a bipartite graph G is a path with the following properties:

- it contains no loops,
- it starts in a free vertex in V_1, ends in a free vertex in V_2, and
- every even edge in the path belongs to M (it follows that no odd edge belongs to M).

If we remove from M these edges that appear also on the augmenting path, and add those edges, which appear on the augmenting path but are not in M, we get a matching containing one edge more. Now, if the matching M is maximum, it is obvious that there are no augmenting paths. The converse observation is not so trivial, but it is also true: if the matching is not maximum, then there exists an augmenting path. This idea leads to an algorithm for finding a maximum matching: one should keep searching for augmenting paths in the graph as long as there are any. If we use BFS to find augmenting paths, the time complexity of our algorithm will be $O\left((n \cdot m)^2\right)$.

There exists also an improved version of this method, known as the Hopcroft-Karp matching algorithm, which works in $O\left((n \cdot m)^{3/2}\right)$ time complexity.

Pavel S. Pankov
Szymon Acedański
Task idea and formulation

Szymon Acedański
Jakub Pawlewicz
Analysis

Available memory: 64 MB, Maximum running time: 2 s.

Polish Flag

Three children are building Polish flag from square blocks. The flag will be a rectangle, $3n$ blocks wide and $2n$ blocks high, where n is a positive integer. It will consist of $3n^2$ white blocks and $3n^2$ red blocks. The children are going to lay blocks on a rectangle table. There are $6n^2$ slots on the table. The white blocks should occupy the top n rows, and the red blocks should occupy the bottom n rows. Rows are numbered from 1 to $2n$ from top to bottom. Columns are numbered from 1 to $3n$ from left to right.

The children are laying blocks in turns. In the first turn Lucy puts her block on the left edge at position $(1,l)$, Bob puts his block on the bottom edge at position $(b,2n)$, Roy puts his block on the right edge at position $(3n,r)$, where $1 \leqslant l,r < 2n, 1 < b < 3n$.

Every next turn they lay blocks as follows. The child can put a block in a given slot only if the slot is empty and the block to be put would be adjacent to one of the blocks put in the preceding turn. (Two blocks are adjacent if they have a common side.) In a given turn the child puts as many blocks as possible. Only one block can be put into a single slot. If two or more children want to put a block into the same slot in the same turn then the highest priority has Lucy, then Bob and the lowest priority has Roy.

Before the children start building the flag they have to distribute blocks. Here is the problem. They don't know how many blocks of each color they need. Help the children and compute for each child the number of blocks of each color he/she will use while building the flag.

Task

Write a program that:

- *reads from standard input the number n and positions of the first blocks l, b, r,*
- *computes for each child the number of blocks of each color he/she should get to build the flag,*
- *writes the result to the standard output.*

Input

The first and only line contains four integers n, l, b, r, separated by single spaces, $1 \leqslant n \leqslant 1\,000\,000\,000, 1 \leqslant l, r < 2n, 1 < b < 3n$. Additionally, in 50 % of test cases n will not exceed 100.

Output

Output should consist of a single line containing six integers separated by single spaces. The first and the second integer should be the number of white and red blocks respectively, which Lucy needs; the third and fourth number should be the number of white and red blocks respectively, which Bob needs; the fifth and sixth number should be the number of white and red blocks respectively, which Roy needs.

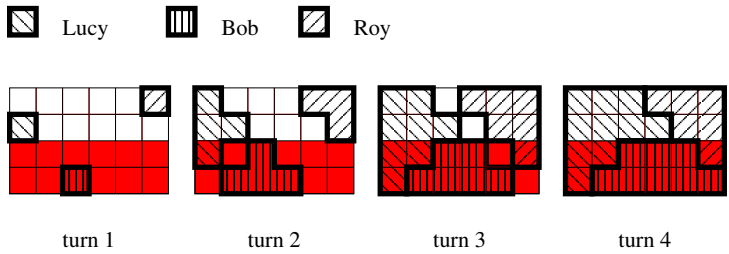
Example

For the input data:

```
2 2 3 1
```

the correct result is:

```
7 3 0 8 5 1
```



Solution

Simple solutions

The simplest idea, how to solve this task, can be to implement an algorithm simulating the process of building the flag, following the description given in the task. Such a solution works in $O(n^3)$ time complexity and $O(n^2)$ memory complexity. Clearly, this is not sufficient to receive the full score, but only about 50%. There are also other faster, although not optimal, solutions.

One can employ Breadth First Search (BFS) to reduce the time complexity of the simulation to $O(n^2)$. Initially, we can put into a queue the starting slots (in order of the children's priorities), and then run BFS to simulate the process of building the flag.

Another idea, working in $O(n^2)$ time and $O(1)$ memory, is to just iterate over all the slots, and to determine, whose block will be in each of them. For a given slot, this can be calculated by comparing Manhattan distances[1] to the children's starting points.

There is also an $O(n)$ solution. At first, please observe, that in each row, its leftmost part (possibly empty) is covered by Lucy's blocks, then there may be some Bob's blocks, and finally on the right there may be some Roy's blocks. We can iterate over the rows, and for each of them we can calculate the positions of the boundaries between the mentioned parts. For the first (the bottommost) row, we can do it in $O(n)$ time. Now, having these boundaries for the first row, we can easily compute them for the next row too, because the new left boundary can either be in the same place as the old one or moved to the right. Similarly, the right boundary can move only to the left. This procedure may be continued until Bob's part disappears. The time complexity of this phase is $O(n)$.

When Bob's part disappears, there is only one boundary and we can calculate its position in $O(n)$ time. For all the remaining rows the boundary can either be in the same place as the boundary in the previous row, or can move one place to the right or to the left (depending on the relative vertical positions of Lucy's starting slot, Roy's starting slot and the current row). This phase also takes $O(n)$ time, so the total time complexity of the above algorithm is $O(n)$.

There is another way to obtain an $O(n)$ solution. Given a row, it is possible to calculate the boundaries in $O(1)$ time for this row, independently of the other rows. This can be done by determining in each row the rightmost slot that is not further from Lucy's starting slot than from Bob's starting slot (and similarly such a slot for Bob and Roy, and for Lucy and Roy) — these computations require only simple arithmetic and case decomposition. Using these positions, it is easy to calculate how many slots belong to Lucy, Bob and Roy in each row.

[1]The *Manhattan distance* between two points is a sum of the absolute differences between their respective coordinates: $d((x_1,y_1),(x_2,y_2)) = |x_2-x_1|+|y_2-y_1|$.

Model solution

The model solution works in $O(1)$ time and memory complexity. It explicitly calculates shapes of the regions covered by Lucy's, Bob's and Roy's blocks. We will show how to determine Bob's region (the other two are calculated analogically). At first, observe, that this region is a figure bounded by the X axis and some discrete function, let us call it *flag function*. Moreover, the domain of this function can be divided into intervals such that, in each interval, the function is either constant or is an arithmetic progression with common difference 1 or -1. An example is shown in Figure 1.

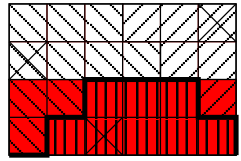


Fig. 1: An example flag function determining the boundary of Bob's region

Now let us imagine, that Roy does not exist. We have only two children, as depicted in Figure 2a. With two children, the flag function always consists of up to three intervals. We can determine them and corresponding function shapes in $O(1)$ time by considering a number of simple cases.

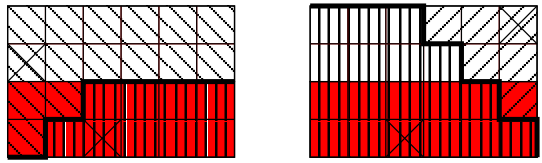


Fig. 2: An example flag function determining the boundary between regions filled with (a) Lucy's and Bob's blocks and (b) Bob's and Roy's bocks

Please note, that this flag function would consist of up to three intervals too, if we had left Bob and Roy only (see Figure 2b).

Having flag functions for Bob and Lucy as well as for Bob and Roy, we can compute the final flag function, which bounds Bob's blocks, as a minimum of these two flag functions. Finally, having this, calculating the appropriate numbers of needed blocks in $O(1)$ time and memory is not very difficult. A similar method can be used to count Lucy's and Roy's blocks.

Mathias Hiron
Szymon Acedański
Task idea and formulation

Szymon Acedański, Łukasz Kowalik
Marcin Stefaniak
Analysis

Available memory: 32 MB, Maximum running time: 0.5 s.

Garden

Byteman owns the most beautiful garden in Bytetown. He planted n roses in his garden. Summer has come and the flowers have grown big and beautiful. Byteman has realized that he is not able to take care of all the roses on his own. He has decided to employ two gardeners to help him. He wants to select two rectangular areas, so that each of the gardeners will take care of the roses inside one area. The areas should be disjoint and each should contain exactly k roses.

Byteman wants to make a fence surrounding the rectangular areas, but he is short of money, so he wants to use as little fence as possible. Your task is to help Byteman select the two rectangular areas.

The garden forms a rectangle l meters long and w meters wide. It is divided into $l \cdot w$ squares of size 1 meter $\times$ 1 meter each. We fix a coordinate system with axes parallel to the sides of the garden. All squares have integer coordinates (x,y) satisfying $1 \leqslant x \leqslant l$, $1 \leqslant y \leqslant w$. Each square may contain any number of roses.

The rectangular areas, which must be selected, should have their sides parallel to the sides of the garden and the squares in their corners should have integer coordinates. For $1 \leqslant l_1 \leqslant l_2 \leqslant l$ and $1 \leqslant w_1 \leqslant w_2 \leqslant w$, a rectangular area with corners (l_1,w_1), (l_1,w_2), (l_2,w_1) and (l_2,w_2):

- *contains all the squares with coordinates (x,y) satisfying $l_1 \leqslant x \leqslant l_2$ and $w_1 \leqslant y \leqslant w_2$, and*
- *has perimeter $2 \cdot (l_2 - l_1 + 1) + 2 \cdot (w_2 - w_1 + 1)$.*

The two rectangular areas must be disjoint, that is they cannot contain a common square. Even if they have a common side, or part of it, they must be surrounded by separate fences.

Task

Write a program, that:

- *reads from the standard input the dimensions of the garden, the number of roses in the garden, the number of roses that should be in each of the rectangular areas, and the positions of the roses,*

- *finds the corners of two such rectangular areas with minimum sum of perimeters that satisfy the given conditions,*
- *writes to the standard output the minimum sum of perimeters of two non-overlapping rectangular areas, each containing exactly the given number of roses (or a single word* `NO`*, if no such pair of areas exists).*

Input

The first line of standard input contains two integers: l *and* w *(*$1 \leqslant l, w \leqslant 250$*) separated by a single space — the length and the width of the garden. The second line contains two integers:* n *and* k *(*$2 \leqslant n \leqslant 5000$*,* $1 \leqslant k \leqslant n/2$*) separated by a single space — the number of roses in the garden and the number of roses that should be in each of the rectangular areas. The following* n *lines contain the coordinates of the roses, one rose per line. The* $(i+2)$*-nd line contains two integers* l_i*,* w_i *(*$1 \leqslant l_i \leqslant l$*,* $1 \leqslant w_i \leqslant w$*) separated by a single space — the coordinates of the square containing the* i*-th rose. Two or more roses can occur in the same square.*

In 50% of test cases, the dimensions of the garden will satisfy $l, w \leqslant 40$*.*

Output

The standard output should contain only one line with exactly one integer — the minimum sum of perimeters of two non-overlapping rectangular areas, each containing exactly k *roses, or a single word* `NO`*, if no such pair of areas exists.*

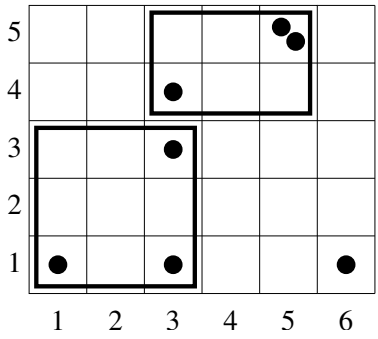
Example

For the input data:

```
6 5
7 3
3 4
3 3
6 1
1 1
5 5
5 5
3 1
```

the correct result is:

```
22
```



Solution

Let us call a rectangular region containing exactly k roses a *k-rectangle*. Let us also denote by $r_{x,y}$ the number of roses in the square (x,y). The problem is to find two disjoint k-rectangles with minimal sum of perimeters.

The simplest solution is to consider all possible rectangular regions of the garden, and for each of them to count the number of roses inside. This way we can enumerate all k-rectangles in $O(w^3 \cdot l^3)$ time. There may be up to $O(w^2 \cdot l^2)$ k-rectangles in total. The second step is to consider all the pairs of k-rectangles and choose the one consisting of disjoint regions with minimum sum of perimeters. Such a solution should receive about 50 % points, despite its terrible time complexity of $O(w^4 \cdot l^4)$.

Model solution

Now we will present a number of gradual improvements, which will finally yield us the model solution. Please note, that we can optimize checking if a given rectangular region is a k-rectangle. Let us denote by $R_{x,y}$ the number of roses in a region, with one corner at $(1,1)$ and the opposite one at (x,y). We can precompute all the values of $R_{x,y}$ iteratively in $O(w \cdot l)$ time using the following formula:

$$R_{x,y} = \begin{cases} 0 & \text{if } x = 0 \text{ or } y = 0 \\ R_{x-1,y} + R_{x,y-1} - R_{x-1,y-1} + r_{x,y} & \text{otherwise} \end{cases}$$

Having this, we can express $\mathcal{R}(x_1,y_1,x_2,y_2)$ — the number of roses in a rectangular region with corners (x_1,y_1) and (x_2,y_2) as:

$$\mathcal{R}(x_1,y_1,x_2,y_2) = R_{x_2,y_2} - R_{x_2,y_1-1} - R_{x_1-1,y_2} + R_{x_1-1,y_1-1}$$

This way, $\mathcal{R}(x_1,y_1,x_2,y_2)$ can be evaluated in $O(1)$ time. Using the presented method, we can enumerate all k-rectangles in $O(w^2 \cdot l^2)$ time. Unfortunately, this does not solve the problem of considering all pairs of k-rectangles.

But fortunately, there is another method, which copes with this problem. Please observe, that if we have two disjoint rectangular regions, then there must exist either a horizontal or a vertical line such that one rectangle is above it (or respectively to the left) and the other one is below it (or respectively to the right). Hence, for each horizontal line we can find two k-rectangles with the smallest perimeters, laying on the opposite sides of the line. Similar values are to be found for all vertical lines. When we have done this, we can easily compute the final result in $O(w+l)$ by considering all the possible dividing lines and choosing the result which gives us the optimal sum of perimeters.

Now we will show how to find optimal perimeters for the first case (rectangular regions above the given horizontal line). The three other cases can be solved analogously. Let us denote by A_y the minimal perimeter of k-rectangle laying above the horizontal line with the given y-coordinate, whose bottommost coordinate is greater than or equal y. Let us also denote by a_y the minimal perimeter of the k-rectangle with bottommost coordinate equal y. Please note, that:

$$A_y = \min(a_y, \ldots, a_w)$$

A simple way to calculate a_i's is to initially set them to infinity, and then update them while iterating through all k-rectangles. With this improvement our algorithm works in $O(w^2 \cdot l^2)$ time.

This is not all. Please note, that we do not need to consider *all* k-rectangles. We can limit our considerations to those k-rectangles, which do not contain any other k-rectangles in their interiors. To enumerate all interesting k-rectangles (and maybe some not interesting too), we consider all pairs (y_1,y_2), $1 \leqslant y_1 \leqslant y_2 \leqslant w$. For each such pair, we use a *sliding window*, which is a rectangle having corners at (x_1,y_1) and (x_2,y_2). At the beginning, $x_1 = x_2 = 1$. Then we repeatedly move the sliding window according to the following rules, until $x_2 > l$:

- if there are exactly k roses in the sliding window (i.e. $\mathcal{R}(x_1,y_1,x_2,y_2) = k$), then we have found a new

k-rectangle; after updating the four constructed sequences (a_i and the three other analogous sequences), x_1 is incremented by one,

- if $\mathcal{R}(x_1, y_1, x_2, y_2) < k$ then x_2 is incremented by one, to expand the sliding window,
- if $\mathcal{R}(x_1, y_1, x_2, y_2) > k$ then x_1 is incremented by one, to shrink the sliding window,

The above algorithm works in $O(w^2 \cdot l)$ time, and enumerates (among others) all interesting k-rectangles. Of course, we can reduce its running time to $O(w \cdot l \cdot \min(w, l))$ by adapting the direction, in which this method works.

The presented solution, with all the above optimizations, constitutes the model solution.

Tom Verhoeff
Szymon Acedański
Task idea and formulation

Szymon Acedański
Jakub Radoszewski
Analysis

Available memory: 16 MB, Maximum running time: 5 s.

Mean Sequence

Consider a nondecreasing sequence of integers $s_1, \ldots, s_{n+1}$ ($s_i \leqslant s_{i+1}$ for $1 \leqslant i \leqslant n$). The sequence $m_1, \ldots, m_n$ defined by $m_i = \frac{1}{2}(s_i + s_{i+1})$, for $1 \leqslant i \leqslant n$, is called the **mean sequence** *of sequence $s_1, \ldots, s_{n+1}$. For example, the mean sequence of sequence 1, 2, 2, 4 is the sequence 1.5, 2, 3. Note that elements of the mean sequence can be fractions. However, this task deals with mean sequences whose elements are integers only.*

Given a nondecreasing sequence of n integers $m_1, \ldots, m_n$, compute the number of nondecreasing sequences of $n + 1$ integers $s_1, \ldots, s_{n+1}$ that have the given sequence $m_1, \ldots, m_n$ as mean sequence.

Task

Write a program that:

- *reads from the standard input a nondecreasing sequence of integers,*
- *calculates the number of nondecreasing sequences, for which the given sequence is mean sequence,*
- *writes the answer to the standard output.*

Input

The first line of the standard input contains one integer n ($2 \leqslant n \leqslant 5\,000\,000$). The remaining n lines contain the sequence $m_1, \ldots, m_n$. Line $i + 1$ contains a single integer m_i ($0 \leqslant m_i \leqslant 1\,000\,000\,000$). You can assume that in 50% of the test cases $n \leqslant 1\,000$ and $0 \leqslant m_i \leqslant 20\,000$.

Output

Your program should write to the standard output exactly one integer — the number of nondecreasing integer sequences, that have the input sequence as the mean sequence.

Example

For the input data:	*the correct result is:*
3 2 5 9	4

Indeed, there are four nondecreasing integer sequences for which $2,5,9$ is the mean sequence. These sequences are:

- $2,2,8,10$,
- $1,3,7,11$,
- $0,4,6,12$,
- $-1,5,5,13$.

Solution

At first, observe that the definition of mean sequence can be applied to any sequence, not only *nondecreasing* sequences. If we drop the condition that sequence $s_1,\ldots,s_{n+1}$ is nondecreasing, then fixing a single s_i fixes the entire sequence, given its mean sequence $m_1,\ldots,m_n$. Let us define the *reflection* operation r on integer a with respect to center c as follows: $r(a,c)=b$ if and only if $\frac{1}{2}(a+b)=c$; that is, $r(a,c)=2c-a$. If s_i is fixed, then $s_{i+1}=r(s_i,m_i)$ and $s_{i-1}=r(s_i,m_{i-1})$, etc. Hence, there exist an infinite number of sequences $s_1,\ldots,s_{n+1}$ that have the given sequence $m_1,\ldots,m_n$ as mean sequence — one for each choice of s_1.

There is a finite number of nondecreasing sequences though. A simple upper bound on the number of possible nondecreasing sequences may be m_2-m_1+1, which is the number of integers between m_1 and m_2 (inclusive). This is because s_2 must satisfy $m_1 \leqslant s_2 \leqslant m_2$. Indeed, if $s_2 < m_1$ then $s_1 > m_2$ and therefore $s_2 < s_1$ so the sequence is not nondecreasing. Similarly, if $s_2 > m_2$ then $s_3 < m_2$ giving a sequence which is not nondecreasing either. This way we can construct a solution, which tests all the possible values of s_2 lying between m_1 and m_2, and for each such s_2 it computes the rest of the sequence and then checks if it is nondecreasing. Such a solution works in $O(n(m_2-m_1+1))$ time and can be implemented with $O(n)$ or better $O(m_2-m_1+1)$ memory complexity.

Optimal solution

Before we present the model solution, a following definition will be useful. Let us call a value v *feasible* for s_i in the given sequence, when there exists an nondecreasing sequence $s_1, \ldots, s_{n+1}$ with $s_i = v$ and mean sequence $m_1, \ldots, m_n$. Now we need an important observation.

Observation 1 *If, for some i, values a and b with $a \leqslant b$ are feasible for s_i, then every c in the interval $[a,b]$ is feasible for s_i.*

The actual number of nondecreasing sequences $s_1, \ldots, s_{n+1}$ can be obtained by generalizing the problem as follows. Given a nondecreasing sequence $m_1, \ldots, m_n$, determine the *interval* of feasible values for s_{n+1}. The answer is then the size of that interval.

In the model solution the interval of feasible values of s_{n+1} is computed inductively. We start with the case $n = 0$. In this case, the interval of feasible values for s_1 consists of all integers: $(-\infty, +\infty)$. Now let $[a,b]$ be the interval for nondecreasing sequences $s_1, \ldots, s_{n+1}$ having mean sequence $m_1, \ldots, m_n$. Let us consider the mean sequence $m_1, \ldots, m_n$ extended by a new element $m_{n+1} \geqslant m_n$. This reduces the possible values for s_{n+1} to the interval $[a, \min(b, m_{n+1})]$, and hence the interval for s_{n+2} is the reflection of this interval, that is, $[r(\min(b, m_{n+1}), m_{n+1}), r(a, m_{n+1})]$. We consider interval $[a,b]$ as empty if $a > b$, and otherwise it contains $b - a + 1$ elements. This way we obtain $O(n)$ time complexity and $O(1)$ memory complexity solution, because there is no need to store the entire sequence in memory. Intervals of feasible values can be computed while reading the input data.

There are also some suboptimal solutions, which use different methods of determining the interval of feasible values for s_{n+1}. One idea would be to use binary search. Even though this can result in an algorithm with $O(n \log n)$ time complexity, it will need $O(n)$ memory, what is too much to pass the largest tests.

Jakub Pawlewicz
Anna Niewiarowska
Task idea and formulation

Szymon Acedański, Arkadiusz Paterek
Jakub Pawlewicz
Analysis

Available memory: 256 MB, Maximum running time: 3 s.

Mountains

The Mountain Amusement Park has opened a brand-new simulated roller coaster. The simulated track consists of n rails attached end-to-end with the beginning of the first rail fixed at elevation 0. Byteman, the operator, can reconfigure the track at will by adjusting the elevation change over a number of consecutive rails. The elevation change over other rails is not affected. Each time rails are adjusted, the following track is raised or lowered as necessary to connect the track while maintaining the start at elevation 0. The figure on the next page illustrates two example track reconfigurations.

Each ride is initiated by launching the car with sufficient energy to reach height h. That is, the car will continue to travel as long as the elevation of the track does not exceed h, and as long as the end of the track is not reached.

Given the record for all the day's rides and track configuration changes, compute for each ride the number of rails traversed by the car before it stops.

Internally, the simulator represents the track as a sequence of n elevation changes, one for each rail. The i-th number d_i represents the elevation change (in centimetres) over the i-th rail. Suppose that after traversing $i-1$ rails the car has reached an elevation of h centimetres. After traversing i rails the car will have reached an elevation of $h + d_i$ centimetres.

Initially the rails are horizontal; that is, $d_i = 0$ for all i. Rides and reconfigurations are interleaved throughout the day. Each reconfiguration is specified by three numbers: a, b and D. The segment to be adjusted consists of rails a through b (inclusive). The elevation change over each rail in the segment is set to D. That is, $d_i = D$ for all $a \leqslant i \leqslant b$.

Each ride is specified by one number h — the maximum height that the car can reach.

Task

Write a program that:

- *reads from the standard input a sequence of interleaved reconfigurations and rides,*
- *for each ride computes the number of rails traversed by the car,*
- *writes the results to the standard output.*

Input

The first line of input contains one positive integer n — the number of rails, $1 \leqslant n \leqslant 1\,000\,000\,000$. The following lines contain reconfigurations interleaved with rides, followed by an end marker. Each line contains one of:

- *Reconfiguration*
  — a single letter '`I`*', and integers a,b and D, all separated by single spaces ($1 \leqslant a \leqslant b \leqslant n$, $-1\,000\,000\,000 \leqslant D \leqslant 1\,000\,000\,000$).*
- *Ride — a single letter '*`Q`*', and an integer h ($0 \leqslant h \leqslant 1\,000\,000\,000$) separated by a single space;*
- *A single letter '*`E`*' — the end marker, indicating the end of the input data.*

You may assume that at any moment the elevation of any point in the track is in the interval $[0, 1\,000\,000\,000]$ centimetres. The input contains no more than 100 000 lines.

In 50 % of test cases n satisfies $1 \leqslant n \leqslant 20\,000$ and there are no more than 1 000 lines of input.

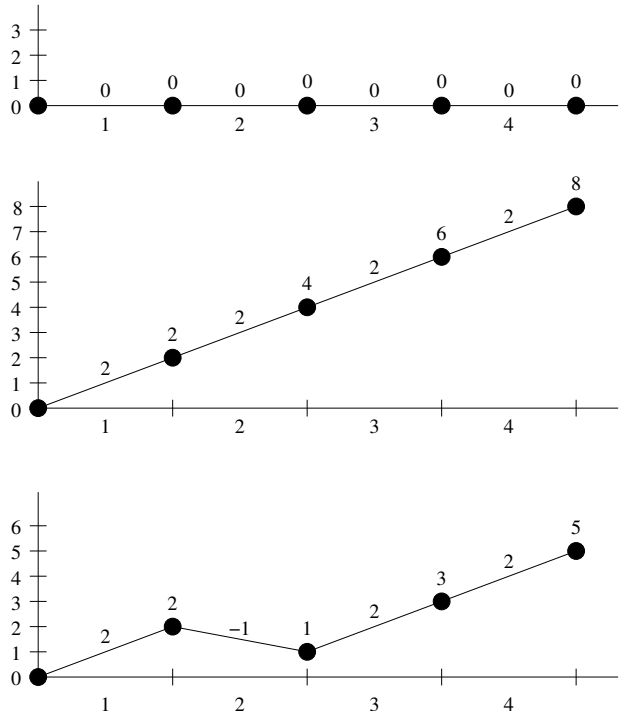
Output

The i-th line of output should consist of one integer — the number of rails traversed by the car during the i-th ride.

Example

For the input data:	*the correct result is:*

`4`
`Q 1`
`I 1 4 2`
`Q 3`
`Q 1`
`I 2 2 -1`
`Q 3`
`E`
	`4`
	`1`
	`0`
	`3`



Views of the track before and after each reconfiguration. The x axis denotes the rail number. The y axis and the numbers over points denote elevation. The numbers over segments denote elevation changes.

Solution

At the beginning, let us denote by I the number of input lines starting with letter '`I`' (i.e. reconfigurations). Similarly, let Q be the number of rides.

Simple solutions

One possible simple, although not optimal, solution is to represent the trail as a vector A, where $A[i]$ denotes the elevation after the i-th rail. The complexity of processing a reconfiguration (we call this operation insertion) is $O(n)$. Single query about a ride also takes $O(n)$, so the total time complexity is

$O((I+Q)\cdot n)$. Moreover, memory complexity is $O(n)$ which is far too high to not exceed the memory limit in large test cases.

Another simple approach is to represent the track as a sorted list of disjoint intervals such that throughout each interval, the difference of elevation over every rail is the same. This way we have insertion and query complexity of $O(I)$, so the entire solution is $O((I+Q)\cdot I)$. Memory complexity is $O(I)$.

Model solution

A data structure used in the model solution is a binary tree. Each of its nodes describes an interval (i.e. a number of consecutive rails) $J = \left[2^k t, 2^k(t+1)\right)$, for some integers k and t. The information contained in the node is

- $S_J = \sum_{i\in J} d_i$
- $H_J = \max\left(\{0\} \cup \left\{\sum_{2^k t \leqslant i \leqslant j} d_i \colon\ j \in J\right\}\right)$

The node is a leaf if all the values d_i are equal. In this case computing S_J and H_J is trivial. Otherwise, the node has two sons, with assigned intervals $J_1 = \left[2^{k-1}(2t), 2^{k-1}(2t+1)\right)$ and $J_2 = \left[2^{k-1}(2t+1), 2^{k-1}(2t+2)\right)$. In this case S_J and H_J are computed as $S_J = S_{J_1} + S_{J_2}$ and $H_J = \max\{H_{J_1}, S_{J_1} + H_{J_2}\}$. The root of the tree represents the interval $\left[0, 2^{\lceil \log_2 n \rceil}\right)$.

The key operation is reconfiguration of the track. It requires inserting or modifying at most $2\lceil \log_2 n \rceil$ nodes into the tree. Similarly, processing a query about a ride requires traversing at most $\lceil \log_2 n \rceil$ nodes. Therefore, such a solution has time complexity of $O((I+Q)\cdot \log n)$ and memory complexity $O(I \cdot \log n)$.

Even better solution

We are not required to process the input line by line. Instead, we can read in the entire input and know in advance, which parts of the track will be reconfigured at all. Let M be a sorted vector of the ends of all the intervals contained in all the `I`-records of the input. We can also assume, that the length of M is some power of 2 (if not, we extend M by adding some large values).

Now we use a tree very similar to the one described in the previous section. The difference is, that each node describes an interval $I = \left[M[2^k t], M[2^k(t+1)]\right)$. We can store such a tree in a single vector, just like in the standard implementation of a heap. Since the size of M is $O(I)$,

so the total time complexity is $O((I+Q)\cdot \log I)$ and memory complexity is reduced to $O(I+Q)$.

Jakub Pawlewicz
Task idea and formulation

Szymon Acedański, Piotr Stańczyk
Analysis

Available memory: 32 MB, Maximum running time: 2 s.

Birthday

It is Byteman's birthday today. There are n children at his birthday party (including Byteman). The children are numbered from 1 to n. Byteman's parents have prepared a big round table and they have placed n chairs around the table. When the children arrive, they take seats. The child number 1 takes one of the seats. Then the child number 2 takes the seat on the left. Then the child number 3 takes the next seat on the left, and so on. Finally the child number n takes the last free seat, between the children number 1 and $n-1$.

Byteman's parents know the children very well and they know that some of the children will be noisy, if they sit too close to each other. Therefore the parents are going to reseat the children in a specific order. Such an order can be described by a permutation $p_1, p_2, \ldots, p_n$ ($p_1, p_2, \ldots, p_n$ are distinct integers from 1 to n) — child p_1 should sit between p_n and p_2, child p_i (for $i = 2, 3, \ldots, n-1$) should sit between p_{i-1} and p_{i+1}, and child p_n should sit between p_{n-1} and p_1. Please note, that child p_1 can sit on the left or on the right from child p_n.

To seat all the children in the given order, the parents must move each child around the table to the left or to the right some number of seats. For each child, they must decide how the child will move — that is, they must choose a direction of movement (left or right) and distance (number of seats). On the given signal, all the children stand up at once, move to the proper places and sit down.

The reseating procedure throws the birthday party into a mess. The mess is equal to the largest distance any child moves. The children can be reseated in many ways. The parents choose one with minimum mess. Help them to find such a way to reseat the children.

Task

Your task is to write a program that:

- *reads from the standard input the number of the children and the permutation describing the desired order of the children,*
- *determines the minimum possible mess,*
- *writes the result to the standard output.*

40 *Birthday*

Input

The first line of standard input contains one integer n ($1 \leqslant n \leqslant 1\,000\,000$). The second line contains n integers $p_1, p_2, \ldots, p_n$, separated by single spaces. Numbers $p_1, p_2, \ldots, p_n$ form a permutation of the set $\{1, 2, \ldots, n\}$ describing the desired order of the children. Additionally, in 50% of the test cases, n will not exceed 1 000.

Output

The first and the only line of standard output should contain one integer: the minimum possible mess.

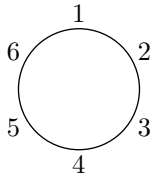
Example

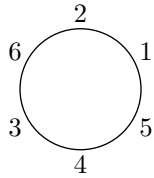
For the input data:

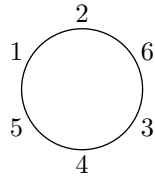
```
6
3 4 5 1 2 6
```

the correct result is:

```
2
```







The left figure shows the initial arrangement of the children. The middle figure shows the result of the following reseating: children number 1 and 2 move one place, children number 3 and 5 move two places, and children number 4 and 6 do not change places. The conditions of arrangement are fulfilled, since 3 sits between 6 and 4, 4 sits between 3 and 5, 5 sits between 4 and 1, 1 sits between 5 and 2, 2 sits between 1 and 6, and 6 sits between 2 and 3. There exists another possible final arrangement of children, depicted in the right figure. In both cases no child moves more than two seats.

Solution

The task is to find such an arrangement of the children, that the maximum number of seats any child has to be moved is minimized. At first we should

note, that there are two classes of possible final arrangements. For example, if we have a permutation $(1,2,3)$, then child 1 can be either a left-hand-side or a right-hand-side neighbour of child 2. The first case will be called *counterclockwise* arrangement, the second will be called *clockwise*. In both cases calculations are similar, therefore we will only consider clockwise arrangements. Contestants have to consider both cases and choose the smaller result.

Simple solution

The first idea may be to perform simulations of all possible rearrangements. Let us fix the position of the first child. Now, using the given permutation, we can calculate the final positions (and also the distances of movements) of all the children in $O(n)$ time complexity. Since we have to perform this step for all the possible positions of the first child, the total time complexity is $O(n^2)$.

Optimal solution

There exists a better solution. We denote by (p_i) the given permutation of the children. Let us consider a final arrangement of the children, where the final position of child p_1 is f. To achieve this arrangement, some (maybe all) of the children have to move. We can describe the movement of child i by a number d_i^f, where $|d_i^f|$ is the distance of movement, it is positive if the child moves clockwise, and negative if the child moves counterclockwise. Moreover, we assume, that the children always choose such a direction of movement, that the distance is smaller than in other direction (or choose clockwise if both distances are equal), that is $1-\lceil \frac{n}{2} \rceil \leqslant d_i^f \leqslant \lfloor \frac{n}{2} \rfloor$.

Let $S_f = \{d_i^f : i = 1,2,\ldots,n\}$. We can treat S_f as an alternative representation of the considered rearrangement. Having this representation, we can easily calculate the maximum movement distance using formula:

$$R_f = \max(-\min(S_f), \max(S_f))$$

Values of d_i^f depend on the given permutation (p_i) and f. They can be characterized by the following formula:

$$d_{p_i}^f = \min(a, n-a) \quad \text{where} \quad a = (f+i-1-p_i) \bmod n$$

Moreover, given some representation S_f, we can easily compute S_{f+1} by "shifting" all elements of S_f (i.e. we replace x by $x+1$ if $x < \lfloor \frac{n}{2} \rfloor$ and we replace $\lfloor \frac{n}{2} \rfloor$ by $1-\lceil \frac{n}{2} \rceil$).

Now we are interested in calculating the smallest possible result for all f's. Please note, that all the representations S_f are shifts of one base representation, say S_0. Let us denote by C the maximum number of consecutive (modulo n) elements from $\left\{1 - \left\lceil \frac{n}{2} \right\rceil, \ldots, \left\lfloor \frac{n}{2} \right\rfloor\right\}$ not appearing in S_0. It can be calculated in a linear time.

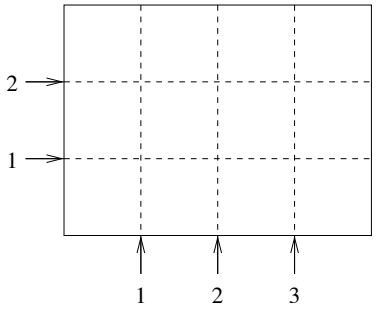
The result is equal $\left\lfloor \frac{n-C}{2} \right\rfloor$. This gives us an algorithm with $O(n)$ time complexity.

Jakub Radoszewski, Marcin Kubica — Task idea and formulation
Szymon Wąsik — Analysis

Available memory: 32 MB, Maximum running time: 14 s.

Rectangle Game

We consider a two-player game. The players are given an $x \times y$ rectangle (where x and y are positive integers). The players take turns moving. A move consists of dividing a rectangle into two rectangles with a single vertical or horizontal cut. The resulting rectangles must have positive integer dimensions.



Possible cuts of a 4×3 rectangle.

After each cut, the smaller rectangle (that is the one with smaller area) is discarded and the other one is passed to the other player. If the rectangle is cut into two equal halves, then one half is discarded. The player who receives a 1×1 rectangle, and therefore is not able to make a move, loses the game.

Your task is to write a program to play and win the rectangle game. The program must use a special library to play the game. The library provides you with functions `dimension_x()` *and* `dimension_y()` *that return the dimensions of the rectangle. Initial dimensions of the rectangle are integers from 1 to 100 000 000. At least one dimension is greater than 1. Moreover, in 50% of test cases the dimensions do not exceed 25.*

There is also a procedure `cut(dir, position)`*, that is called by your program to make moves. Parameters* `dir` *and* `position` *describe the direction and the position of a cut respectively. The parameter* `dir` *must be one of the two values:* `vertical` *or* `horizontal`*. If* `dir` = `vertical` *then the cut is vertical, and the parameter* `position` *specifies the x-coordinate of the cut (see the figure above) and you must ensure that $1 \leqslant$* `position` $\leqslant$ `dimension_x()` -1*. If* `dir` = `horizontal`*, then the cut is horizontal and the parameter* `position` *specifies the y-coordinate of the cut and you must ensure that $1 \leqslant$* `position` $\leqslant$ `dimension_y()` -1*.*

When your program is started, it will act as one player for one game. Your program plays first — it must cut the initial rectangle. When your program calls the `cut` *procedure, your move is recorded and control is passed to your program's opponent. After the opponent moves, control returns to your program. Values returned by* `dimension_x()` *and* `dimension_y()` *will reflect the result of your move and your opponent's move. As soon as your program wins, loses or makes an illegal move (i.e. calls the* `cut` *procedure with invalid parameters) it will be terminated. Termination of your program is an automatic process, so your program should keep making moves as long as possible. You can assume that for the test data, there always exists a winning strategy for your program.*

Your program must not read or write any files, it must not use standard input/output, and it must not try to modify any memory outside your program. Violating any of these rules may result in disqualification.

Experimentation

To let you experiment with the library, you are given example opponent libraries: their sources are in `preclib.pas`, `creclib.c` *and* `creclib.h` *files. The library can be downloaded from* `http://contest/`. *They implement a very simple strategy. When you run your program, it will be playing against these simple players. Feel free to modify them, and test your program against a better opponent. However, during the evaluation, your program will be playing against a different opponent.*

When you submit your program using the TEST interface it will be compiled with the unmodified example opponent library. The submitted input file will be given to your program standard input. The input file should consist of two lines, each containing one integer. The first line should contain the initial width, and the second line should contain the initial height of the rectangle. These dimensions are read by the example opponent library.

If you modify the implementation part of the `preclib.pas` *library, please recompile it using the following command:* `ppc386 -O2 preclib.pas`. *This command produces files* `preclib.o` *and* `preclib.ppu`. *These files are needed to compile your program, and should be placed in the directory, where your program is located. Please do not modify the interface part of the* `preclib.pas` *library.*

If you modify the `creclib.c` *library, please remember to place it (together with* `creclib.h`*) in the directory, where your program is located — they are needed to compile it. Please do not modify the* `creclib.h` *file.*

You are also provided with two simple programs illustrating usage of the above libraries: `crec.c` *and* `prec.pas`*. (Please remember, that these programs are not correct solutions.) You can compile them using the following commands:*

```
gcc -O2 -static crec.c creclib.c -lm
g++ -O2 -static crec.c creclib.c -lm
ppc386 -O2 -XS prec.pas
```

Library

You are given a library providing the following functionality:

- **FreePascal Library** *(*`preclib.ppu`*,* `preclib.o`*)*

  ```
  type direction = (vertical, horizontal);
  function dimension_x(): longint;
  function dimension_y(): longint;
  procedure cut(dir: direction; position: longint);
  ```

  Include the following statement in your source file `rec.pas`*:*
  `uses preclib;`
  To compile your program, copy the files `preclib.o` *and* `preclib.ppu` *to the directory, where your source file is placed and run the following command:*
  `ppc386 -O2 -XS rec.pas`

  File `prec.pas` *gives an example of how to use the* `preclib` *library.*

- **GNU C/C++ Library** *(*`creclib.h`*,* `creclib.c`*)*

```
typedef enum __direction {vertical, horizontal} direction;
int dimension_x();
int dimension_y();
void cut(direction dir, int position);
```

Include the following statement in your source file (`rec.c` *or* `rec.cpp`*):*
`#include ''creclib.h''`
To compile your program, copy the files `creclib.c` *and* `creclib.h` *to the directory, where your source file is placed and run the following command:*
`gcc -O2 -static rec.c creclib.c -lm`
or:
`g++ -O2 -static rec.cpp creclib.c -lm`

The file `crec.c` *gives an example of how to use the library in C.*

Sample interaction

Below there is a sample interaction between your program and the judging library. It shows how a sample game can proceed. The game starts with a 4×3 board. There exists a winning strategy for this position.

Your program calls	**What happens**
`dimension_x()`	*returns 4*
`dimension_y()`	*returns 3*
`cut(vertical, 1)`	*your cut is recorded and a 3×3 board is passed to your opponent, who cuts it to a 3×2 board; after this, control is passed back to your program*
`dimension_x()`	*returns 3*
`dimension_y()`	*returns 2*
`cut(horizontal, 1)`	*your cut is recorded and a 3×1 board is passed to your opponent, who cuts it to a 2×1 board; after this, control is passed back to your program*
`dimension_x()`	*returns 2*
`dimension_y()`	*returns 1*
`cut(vertical, 1)`	*your cut gives a 1×1 board, so you win; your program is terminated automatically*

Solution

The key to the solution is to find the characterization of winning and losing positions. The first step to do it can be preparation of a chart depicting the distribution of winning and losing positions. Such a chart is shown in Figure 1.

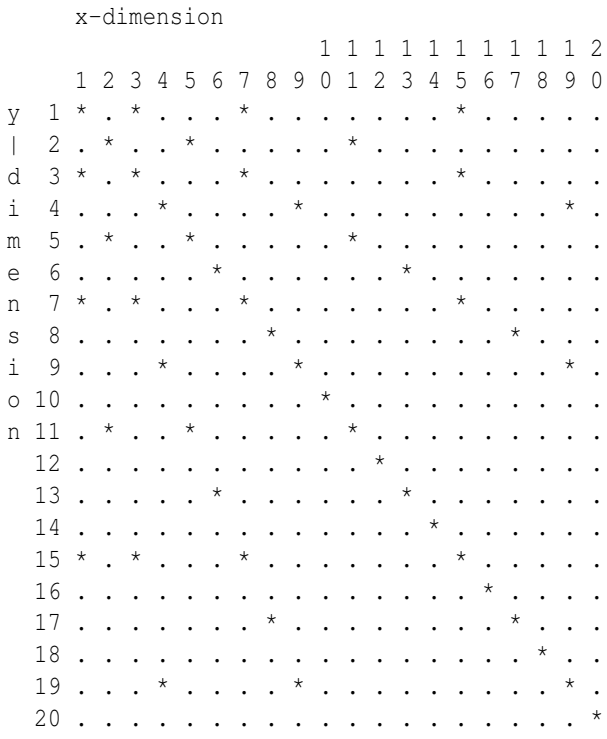


Fig. 1: A chart depicting winning and losing situations. The latter ones are marked with stars.

After a short analysis of the chart, we can observe the following fact:

Lemma 1 *A $n \times m$ rectangle is a losing position if and only if there exists such integer k, that:*

$$m+1 = 2^k \cdot (n+1) \tag{1}$$

We will prove lemma 1 by induction on the rectangle area.

1. Let us assume that n and m fulfill the condition (1) for $k = 0$, that is $n = m$. We can prove that such a position is losing, using simple induction on n:

   (a) If $n = 1$ then the position is losing by the definition of the game.

   (b) Let us assume that $n > 1$ and that the first player makes a move producing a $n \times l$ rectangle, where $\frac{n}{2} \leqslant l < n$. Then his opponent can reduce the rectangle to $l \times l$, which is a losing position.

2. Let us assume that n and m fulfill the condition (1) for $k \neq 0$. Without the loss of generality, we can assume that $k > 0$, because $m+1 = 2^k \cdot (n+1)$ is equivalent to $n+1 = 2^{-k} \cdot (m+1)$. The first player can cut the rectangle in two possible directions:

   (a) Let us assume, that after the first cut we have a $l \times m$ rectangle, where $\frac{n}{2} \leqslant l < n$. Since $n = 2^k(m+1) - 1$, we have $2^{k-1}(m+1) - 1 < l < 2^k(m+1) - 1$. Hence, the second player can reduce the rectangle to $2^{k-1}(m+1) - 1 \times m$, which is a losing position for the first player.

   (b) Let us assume, that after the first cut we have a $n \times l$ rectangle, where $\frac{m}{2} \leqslant l < m$. We have to show that $n \neq 2^i(l+1) - 1$ for all integer values of i.

   We will prove it by contradiction — let us assume, that $n = 2^i(l+1) - 1$ for some integer i. Since $l < m$, we have $n = 2^i(l+1) - 1 < 2^i(m+1) - 1$, and hence $i > k$.

   On the other hand, since $\frac{m}{2} \leqslant l$, we have:

   $$n = 2^i(l+1) - 1 \geqslant 2^i(\frac{m}{2} + 1) - 1 = 2^{i-1}(m+2) - 1 > 2^{i-1}(m+1) - 1$$

   So, $i - 1 < k$, and hence $i < k + 1$.

   We have obtained a contradiction. No integer i can satisfy $k < i < k+1$. Therefore, $n \times l$ is a winning position.

   From the above we obtain that $n \times m$ is a losing position.

3. Let us assume that n and m do not fulfill the above condition. Then, $\log_2(\frac{n+1}{m+1})$ is not an integer. Without the loss of generality, we can assume that $n \geqslant m$. Let us denote by l the following value:

   $$l = 2^{\lfloor \log_2(\frac{n+1}{m+1}) \rfloor}(m+1) - 1$$

We have $\frac{n+1}{2} < l+1 < n+1$, and from this we obtain $\frac{n}{2} \leqslant l < n$. So, the first player can cut a rectangle reducing it to $l \times m$, which is a losing position for the second player.

The model solution follows the proof of the lemma and for every winning position it finds the appropriate move in logarithmic time. However the number of moves can be linear. For example, for a rectangle $2i \times 2(i+1)$ the only winning move is to reduce it to $2i \times 2i$. So, the worst-case time complexity of the solution is $O(n \log n)$.

Alternative solutions

The backtracking (mini-max) solution checking recursively every move is the expected simplest solution which should score half of the points.

Other solutions are based on dynamic programming. For each position (n,m) we can compute whether it is losing or winning one. Simple dynamic programming solution can check all possible moves for a given position (n,m). This leads to $O(n^3)$ time complexity.

There is also a faster dynamic programming solution, which stores the greatest $n' < n$ and $m' < m$ for which (n',m) and (n,m') are losing positions. That way we may find the optimal move for a given position in $O(1)$ time, which results in $O(n^2)$ time complexity.

Łukasz Kowalik
Task idea and formulation

Łukasz Kowalik, Tomasz Malesiński
Analysis

Available memory: 32 MB, Maximum running time: 1 s.

Rivers

Nearly all of the Kingdom of Byteland is covered by forests and rivers. Small rivers meet to form bigger rivers, which also meet and, in the end, all the rivers flow together into one big river. The big river meets the sea near Bytetown.

There are n lumberjacks' villages in Byteland, each placed near a river. Currently, there is a big sawmill in Bytetown that processes all trees cut in the Kingdom. The trees float from the villages down the rivers to the sawmill in Bytetown. The king of Byteland decided to build k additional sawmills in villages to reduce the cost of transporting the trees downriver. After building the sawmills, the trees need not float to Bytetown, but can be processed in the first sawmill they encounter downriver. Obviously, the trees cut near a village with a sawmill need not be transported by river. It should be noted that the rivers in Byteland do not fork. Therefore, for each village, there is a unique way downriver from the village to Bytetown.

The king's accountants calculated how many trees are cut by each village per year. You must decide where to build the sawmills to minimize the total cost of transporting the trees per year. River transportation costs one cent per kilometre, per tree.

Task

Write a program that:

- *reads from the standard input the number of villages, the number of additional sawmills to be built, the number of trees cut near each village, and descriptions of the rivers,*
- *calculates the minimal cost of river transportation after building additional sawmills,*
- *writes the result to the standard output.*

Input

The first line of input contains two integers: n — the number of villages other than Bytetown ($2 \leqslant n \leqslant 100$), and k — the number of additional sawmills to

be built ($1 \leqslant k \leqslant 50$ and $k \leqslant n$). The villages are numbered $1, 2, \ldots, n$, while Bytetown has number 0.

Each of the following n lines contains three integers, separated by single spaces. Line $i + 1$ contains:

- *w_i — the number of trees cut near village i per year ($0 \leqslant w_i \leqslant 10\,000$),*
- *v_i — the first village (or Bytetown) downriver from village i ($0 \leqslant v_i \leqslant n$),*
- *d_i — the distance (in kilometres) by river from village i to v_i ($1 \leqslant d_i \leqslant 10\,000$).*

It is guaranteed that the total cost of floating all the trees to the sawmill in Bytetown in one year does not exceed 2 000 000 000 cents.

In 50 % of test cases n will not exceed 20.

Output

The first and only line of the output should contain one integer: the minimal cost of river transportation (in cents).

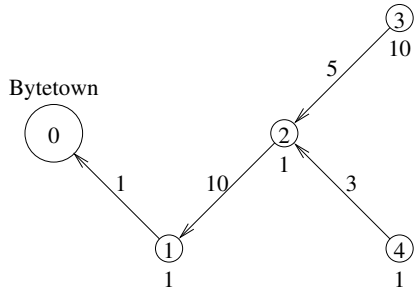
Example

For the input data:

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

the correct result is:

```
4
```



The above picture illustrates the example input data. Village numbers are given inside circles. Numbers below the circles represents the number of trees cut near villages. Numbers above the arrows represent rivers' lengths.

The sawmills should be built in villages 2 and 3.

Solution

Notation

It is not hard to observe that since for each village there is a unique way down the river from the village to Bytetown, we can treat the rivers and villages as a tree with the root in Bytetown. Nodes of the tree correspond to the villages (for convenience we will refer to Bytetown as a village too), and node v is the parent of node u when v is the first village downriver from u.

Let r denote the root of the tree, i.e. r corresponds to Bytetown. By $\text{depth}(u)$ we will denote the number of edges on a unique path from u to r. Clearly, the values of $\text{depth}(u)$ can be computed for all villages u in linear time. The number of children of node u will be denoted by $\deg(u)$, and the number of trees cut near village u will be denoted by $\text{trees}(u)$.

Dynamic Programming

We can apply dynamic programming to solve the task. Let $A_{v,t,l}$ denote the minimal cost of transportation of the trees cut in the subtree rooted in v, assuming that t additional sawmills can be built in the subtree, and the trees not processed in v can be processed in the village of depth l (on the way from v to Bytetown). We compute values of $A_{v,t,l}$ for each village v, and such numbers t, l, that $0 \leqslant t \leqslant k$ and $0 \leqslant l < \text{depth}(v)$. Clearly, when the tree rooted in v has at most t nodes, then $A_{v,t,l} = 0$, as we can simply place a sawmill in every village. We can use the following formula:

$$A_{v,t,l} = \begin{cases} 0 & \text{when the tree rooted in } v \text{ has at most } t \text{ nodes,} \\ \min(A'_{v,t,l}, A''_{v,t}) & \text{otherwise,} \end{cases} \tag{1}$$

where $A'_{v,t,l}$ is the cost of transportation when there is no sawmill in v, and $A''_{v,t}$ is the cost of transportation when there is one. These costs depend on the distribution of sawmills between subtrees rooted in children of v. Let $d = \deg(v)$ and let $v_1, v_2, \ldots, v_d$ be the children of v. Then:

$$A'_{v,t,l} = \text{trees}(v) \cdot (\text{depth}(v) - l) + \min_{t_1+\ldots+t_d=t} \sum_{i=1}^{d} A_{v_i,t_i,l}, \tag{2}$$

$$A''_{v,t} = \min_{t_1+\ldots+t_d=t-1} \sum_{i=1}^{d} A_{v_i,t_i,\text{depth}(v)}. \tag{3}$$

Dynamic Programming One More Time

Let us have a closer look at the recurrences (2) and (3). We do not need to consider every partition of t into a sum of $\deg(v)$ terms, to compute the values of A' and A''. It would be time-consuming in case of trees containing vertices with many children. Once again, we can use dynamic programming. Let $B^{i,s}_{v,l}$ denote the cost of transporting the trees cut in subtrees rooted in $v_1, v_2, \ldots, v_i$ provided that s additional sawmills can be built in these subtrees and the trees not processed in these subtrees are processed in a village of depth l. We can make use of the following recurrence:

$$\begin{aligned} B^{0,s}_{v,l} &= 0, \\ B^{i,s}_{v,l} &= \min_{0 \leqslant j \leqslant s} (B^{i-1,s-j}_{v,l} + A_{v_i,j,l}) \quad \text{for each } s = 1, \ldots, k. \end{aligned} \tag{4}$$

We define $C^{i,s}_v$ analogously, but this time we assume that the trees not processed in the subtrees are processed in v. Then

$$\begin{aligned} C^{0,s}_{v} &= 0, \\ C^{i,s}_{v} &= \min_{0 \leqslant j \leqslant s} (C^{i-1,s-j}_{v,l} + A_{v_i,j,\mathrm{depth}(v)}) \quad \text{for each } s = 1, \ldots, k. \end{aligned} \tag{5}$$

Obviously, $A'_{v,t,l} = B^{\deg(v),t}_{v,l}$ and $A''_{v,t} = C^{\deg(v),t-1}_{v}$. In order to compute all values of $A'_{v,t,l}$ (respectively $A''_{v,t}$), for some $l \in \{0, \ldots, \mathrm{depth}(v)\}$, we compute $B^{i,s}_{v,l}$ (respectively $C^{i,s}_v$) for each $i = 0, \ldots, \deg(v)$ and $s = 0, \ldots, k$. It follows that for each pair v, l it takes $O(k^2(\deg(v)+1))$ time to compute values $B^{i,s}_{v,l}$ and $C^{i,s}_v$, giving total time:

$$O(n \sum_v k^2(\deg(v)+1)) = O(k^2 n^2),$$

since $\sum_v \deg(v)$ is equal to the number of edges in the tree, i.e. $n-1$. After computing all values of $A_{v,t,l}$ the program returns $A'(r,k,0)$ as the final answer.

Too Much Dynamic Programming

Using dynamic programming twice would not be required if all the vertices in the tree had small number of children. Fortunately, we can easily construct a binary tree of villages that has the same minimal cost of transportation as the original one. In order to do this we connect the first child of each vertex

to the parent as its left child, create additional village and connect it as a right child to the parent. Then we connect the other children of the parent vertex in the original tree one by one, each time connecting a child as a left child of the additional village created during connecting the previous child, creating a new additional village and connecting it as a right child of the previous one. Additional villages produce no wood and rivers connecting them to parents are of length 0, so they do not affect the total cost of transportation. Building sawmills in additional villages does not allow to lower the minimal cost of transportation, because every sawmill built in an additional village can be moved to the next non-additional village on the way to Bytetown without raising (and even possibly lowering) the total cost.

There are $t+1$ partitions of t into a sum of 2 non-negative terms. For each pair v, l we can therefore compute $A_{v,t,l}$ for all t in $O(k^2)$ time. There are twice as many vertices in the binary tree as in the original one. The total computation time is:

$$O(n\sum_{v} k^2) = O(k^2 n^2).$$

One River

It is worth noting, that in the special case, where instead of a tree of rivers there is just one river with multiple villages, the problem is significantly simplified and can be solved more efficiently. In such a case we can use simple dynamic programming. Let us number the villages from 1 to n upstream. For each village v, and for each number of sawmills q ($0 \leqslant q \leqslant k$) let us denote by $T[v,q]$ the optimal cost of locating q sawmills in the part of the river downstream from v. The value of $T[v,q]$ can be computed in $O(v)$ time: we check all v possibilities of placing the upstream-most sawmill using stored values of $T[1,q-1],\ldots,T[v-1,q-1]$. Such an algorithm works in $O(k \cdot n^2)$ time.

A special case of such a problem — for just two sawmills — appeared at CEOI 2004, but the upper bound on n was much bigger: $n \leqslant 20\,000$. This required at least an $O(n \cdot \log n)$ algorithm, which did not use dynamic programming, but was based on divide and conquer technique instead. There exists also a solution running in linear time, iterating over all possible locations of the uppermost sawmill and finding the optimal location of the other sawmill in amortized constant time.