Problem A. A + B

Input file:	aplusb.in
Output file:	aplusb.out
Time limit:	8 seconds
Memory limit:	512 mebibytes

Andrew is studying in the seventh grade. The class has recently learned about periodic decimals, so Andrew is very curious how to work with them.

Recall that a periodic decimal is the way to represent rational numbers by specifying the preperiod and the period of its infinite decimal representation. For example, $1/7 = 0.(142857)$, $1/12 = 0.08(3)$.

There are some numbers that have two representations as periodic decimals, those that are actually finite decimals such as $0.(9) = 1.(0)$. In this problem such numbers must be represented as ending with zeroes, so $0.(9)$ is an incorrect periodic decimal for the purpose of this problem.

Now Andrew wants to add two numbers given as periodic decimals. The sum of two periodic decimals is always again a periodic decimal, but the length of the period can be quite big. Therefore Andrew only asks you to find some particular digits of the sum. You have to output digits at positions $a_1, a_2, \ldots, a_n$.

To simplify the task, Andrew only wants to add numbers of the form $0.(\alpha)$ where α is a period.

Input

The input file contains multiple test cases. The first line of the input file contains t — the number of tests in the input file.

The first two lines of each test case contain periods of the two numbers to add, the periods contain digits from 0 to 9. The length of each period is at most 100 000. The following line contains n — the number of queries, the line with n integers a_i follows ($1 \le n \le 300\,000$, $1 \le a_i \le 10^{18}$).

The total length of periods of all numbers in the input file doesn't exceed 200 000. The sum of n in the input file doesn't exceed 300 000.

Output

For each a_i in the test case print a digits that is at the a_i-th position after the decimal point in the sum of two periodic decimals specified in the input. Positions are numbered from 1.

Do not separate digits by spaces. Print answer for each test case on a separate line.

Example

aplusb.in	aplusb.out
2 142857 3 10 1 2 3 4 5 6 7 8 9 10 4 5 3 1 2 3	4761904761 000

In the first example $0.(142857) = 1/7$, $0.(3) = 1/3$, $1/7 + 1/3 = 0.4761904761904761\ldots$

In the second example $0.(4) = 4/9$, $0.(5) = 5/9$, $4/9 + 5/9 = 1.00000000\ldots$

Problem B. Break Free

Input file:	`breakfree.in`
Output file:	`breakfree.out`
Time limit:	2 seconds
Memory limit:	512 mebibytes

Bob is taking part in TV show. The show takes place at at a large flat arena with n tigers. Arena is a half-plane, let us introduce coordinate system in such way that arena consists of all points with $y \geq 0$. Bob is blindfolded and put initially to a point (x, y).

The goal of Bob is to get out of the arena without being caught by a tiger. The exit from the arena a gate that is a segment connecting points $(0, 0)$ and $(a, 0)$. Bob knows his position and can choose any direction, but he is blindfolded, and he cannot see where the tigers are. Bob can run with the speed not exceeding v.

The tigers can see Bob, the i-th tiger is located at a point (x_i, y_i) and can move with a maximal speed u_i. After the start of the show Bob and the tigers are simultaneously set free.

Bob has decided to use the following strategy to escape the arena. He chooses some straight line segment that connects his initial point with some point of the gate, and runs along it with his maximal speed. Bob calls the point P of the gate *safe* if when running along the segment from his initial point to the point P he cannot be caught by a tiger. Let us denote the set of safe points as S. Now Bob wants to evaluate his chance of escaping, so he wants to find the *measure* $\mu(S)$ of the set of safe points.

In this problem the measure $\mu(S)$ is equal to the infimum of the set A of such a that there is a set of segments with total length a that completely cover S. Infimum of the set B is such b that there is no $t \in B$ which is less than b, but for each $b_0 > b$ there is such t.

Help Bob to find the measure of the set of safe points.

Input

The input file contains multiple test cases. The first line of the input file contains t — the number of tests in the input file ($1 \leq t \leq 100$).

The following lines describe test cases. The first line of each test case contains four integers: a, x, y and v ($1 \leq a \leq 10^4$, $-10^4 \leq x \leq 10^4$, $1 \leq y \leq 10^4$, $1 \leq v \leq 100$).

The second line contains n — the number of tigers ($1 \leq n \leq 100$).

The following n lines describe tigers. The i-th tiger is described by three integers: x_i, y_i, u_i ($-10^4 \leq x_i \leq 10^4$, $0 \leq y_i \leq 10^4$, $1 \leq u_i \leq 100$). No tiger is located at a point (x, y).

Output

For each test case output one floating point number: the measure of the set of safe points. Your output must have absolute or relative error not exceeding 10^{-7}.

Example

breakfree.in	breakfree.out
1 10 5 4 10 1 4 1 3	7.173513477283126

Problem C. Covering Words with Carrying

Input file:	`cover.in`
Output file:	`cover.out`
Time limit:	2 seconds
Memory limit:	512 mebibytes

Consider a word p of length m and a word s of length n, $n \geq m$. The word p is said to *cover* s *with carrying* if there is integer k such that p^k is prefix of ss and length of p^k is greater or equal to the length of s. Here p^k means the concatenation of k copies of p.

You can imagine that the word s is written on a circle and you cover it from the beginning by concatenating copies of p, but if the next copy is exceeding the end of s, it must continue to cover its beginning.

For example, the word "`01001001`" is covered with carrying by the word "`010`", but the word "`1011011`" is not covered with carrying by a word "`101`", neither is the word "`10110101`".

One word can be covered with carrying by several other words. For example, a word "`11111`" is covered with carrying by any word of length from 1 to 5 that consists of '`1`'-s only. Let us denote as $cc(s)$ the number of words that cover s with carrying. For example, $cc($"`11111`"$) = 5$ and $cc($"`01001001`"$) = 2$.

Consider all words of length n that consist of characters '`0`' and '`1`'. Find the sum of values $cc(s)$ for all such words. Print it modulo 998 244 353.

For example, if $n = 3$, $cc($"`000`"$) + cc($"`001`"$) + cc($"`010`"$) + cc($"`011`"$) + cc($"`100`"$) + cc($"`101`"$) + cc($"`110`"$) + + cc($"`111`"$) = 3 + 1 + 1 + 1 + 1 + 1 + 1 + 3 = 12$.

Input

The input file contains several test cases. Each test case consists of an integer n on a line ($1 \leq n \leq 1000$). The last test case is followed by a line that contains zero. It must not be processed.

Output

For each test case output one integer: the sum of $cc(s)$ for all binary words of length n, modulo 998 244 353.

Examples

cover.in	cover.out
3 0	12

Problem D. Decisions

Input file:	`decisions.in`
Output file:	`decisions.out`
Time limit:	5 seconds
Memory limit:	512 mebibytes

Denny is preparing for a class trip. Denny has several items that he is planning to take to the trip. He has put them as a single row and numbered from 1 to n. For each item Denny knows its weight w_i. The knapsack Denny is going to take to the trip has capacity of c, so Denny can take several items with total weight at most c.

Unfortunately Denny is very bad in making decisions. He cannot decide which items would be more useful in a trip. Additionally he doesn't like carrying too much, so probably he is not going to fill his knapsack completely. To help himself choose, Denny has decided to set additional constraints about items to choose.

Now Denny generates various plans of constraints. Each plan is a triple (l_j, r_j, t_j) where $1 \le l_j \le r_j \le n$ and $0 \le t_j \le c$. To satisfy constraints plan Denny must choose several items with numbers between l_j and r_j, inclusive, with the total weight exactly equal to t_j. For each plan Denny immediately wants to know whether it is possible to satisfy it. Help him!

Input

The input file contains several test cases.

Each test case starts with an integer n on a line ($1 \le n \le 2000$). The second line of each test case contains n integers w_i ($1 \le w_i \le 100$). The third line contains the capacity of the knapsack c ($1 \le c \le 1000$). The fourth line contains q — the number of plans that Denny is going to propose ($1 \le q \le 300\,000$). Each of the following q lines describe plans.

The j-th plan is described by three integers: a_j, b_j, c_j ($0 \le a_j < n$, $1 \le b_j \le n$, $0 \le c_j \le c$). Let there be k plans in this test case before the j-th one, that can be satisfied. Use the following equation to get the plan: $l_j = (a_j + k) \bmod (n - b_j + 1) + 1$, $r_j = l_j + b_j - 1$, and $t_j = (c_j + k) \bmod (c + 1)$.

The last test case is followed by a line that contains a single 0, it must not be processed. The sum of n for all test cases in one input file doesn't exceed 2000. The sum of q for all test cases in one input file doesn't exceed 300 000.

Output

For each test case output a line of characters '`Y`' and '`N`'. For each plan print '`Y`' if it can be satisfied, or '`N`' if it cannot.

Examples

decisions.in	decisions.out
5 2 3 4 5 9 10 5 0 3 5 0 2 4 0 3 4 0 4 6 2 3 5 0	YNYYN

In the given example the plans are $(1, 3, 5)$, $(2, 3, 5)$, $(2, 4, 5)$, $(1, 4, 8)$, $(3, 5, 8)$.

Problem E. Enigmatic Matrix

Input file:	`enigmatic.in`
Output file:	`enigmatic.out`
Time limit:	2 seconds
Memory limit:	512 mebibytes

Eleanor likes brackets. She likes correct bracket sequences most of all. A word w composed of '`(`' and '`)`' characters is called a correct bracket sequence if the following two conditions are satisfied:

- w has the same number of opening and closing brackets;
- the number of opening brackets in any prefix of w is greater or equal to the number of closing brackets in that prefix.

For example, "`(())()`" or "`()()()`" are correct brackets sequences, but "`((((`" or "`())(`" are not.

Eleanor wants to generalize the notion of the correct bracket sequence to matrices, but her first attempt that any row and any column must be a correct bracket sequence failed: there are clearly no such matrices.

Her second attempt is the following definition of an *enigmatic* bracket matrix.

The matrix composed of '`(`' and '`)`' characters is called enigmatic, if each of its rows is a cyclic shift of some correct bracket sequence, and each of its columns is a cyclic shift of some correct bracket sequence.

A word x is a cyclic shift of a word y if $x = uv$ and $y = vu$ for some (possibly empty) words u and v.

For example, the picture below shows an enigmatic 4×4 matrix.

```
( ( ) )
( ) ( )
) ( ) (
) ) ( (
```

Now Eleanor wants to count the number of different $h \times w$ enigmatic matrices. Help her! The answer must be printed modulo 998 244 353.

Input

The input file contains several test cases. Each test case consists of two integers h and w on a line ($2 \le h, w \le 16$, h and w are even). The last test case is followed by a line that contains two zeroes. It must not be processed.

Output

For each test case output one integer: the number of enigmatic $h \times w$ matrices modulo 998 244 353.

Examples

enigmatic.in	enigmatic.out
4 4 0 0	90

Problem F. Four Russians on a Tree

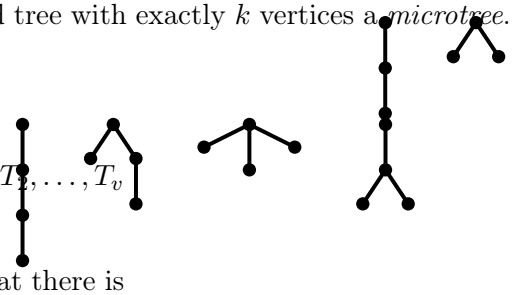
Input file:	four.in
Output file:	four.out
Time limit:	2 seconds
Memory limit:	512 mebibytes

Four Russians method is a general approach to reducing complexity of various combinatorial problems which proceeds as follows: separate the problem instance to small blocks, precompute results for all possible small blocks, and then solve the original problem by processing each small block in one step. The Four Russians are actually the Russian scientists Arlazarov, Diniz, Kronrod and Faradjev.

Fred is solving some problem on the tree and he has decided to use Four Russians method. The Four Russians method on a tree involves splitting the tree to small trees of fixed size. This splitting is usually easy, but unfortunately Fred didn't completely understand the method, so his preparation step is now to difficult for him. He asks you to help.

Consider a rooted tree T with n vertices. All trees in this problem are unlabelled and have unordered children. Fred has chosen an integer k, and calls a rooted tree with exactly k vertices a *microtree*. He now wants split T to microtrees in the following way.

- Remove some leafs of T to get the tree T'.
- Remove some edges of T' to get microtrees $T_1, T_2, \ldots, T_v$ (each microtree T_i must have exactly k vertices).



He wants to choose such way to split T to microtrees that there is minimal number of *microtree types*. Two microtrees have the same type if they are isomorphic as rooted trees. For example, there are 2 types of microtrees with 3 vertices, and 4 types of microtrees with 4 vertices, they are shown at the picture on the right.

Help Fred. For the given tree T and integer k find the minimal number s such that T can be split in the way described above to microtrees of size k that belong to one of s types.

Input

The input file contains several test cases. The first line of each test case contains n and k ($2 \le n \le 100$, $2 \le k \le 5$). The second line contains $n - 1$ integers and describe the tree T. The root of T is vertex number 1. For each vertex i from 2 to n its parent p_i is specified, $p_i < i$.

The last test case is followed by a line that contains two zeroes, it must not be processed. The sum of n for all tests cases is one file doesn't exceed 300.

Output

For each test case output s — the minimal number such that T can be split to microtrees of s types. If it is impossible to split the tree in such way, output -1.

Examples

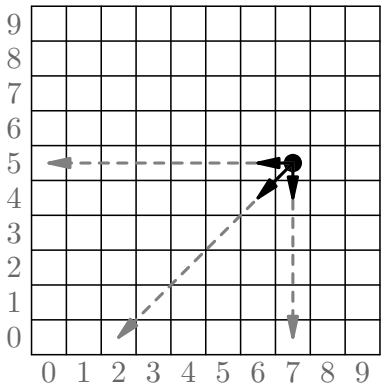
four.in	four.out
7 3 1 1 2 2 4 4 8 3 1 1 2 2 3 4 5 11 3 1 1 2 2 3 3 4 5 6 7 0 0	1 2 -1

Problem G. Grundy

Input file:	grundy.in
Output file:	grundy.out
Time limit:	7 seconds
Memory limit:	512 mebibytes

Grundy or Sprague-Grundy values are a strong tool in combinatorial games analysis. In this problem you will have to find Sprague-Grundy values for Wythoff's Nim.

Consider an infinite quarter-plane chessboard with cells indexed by pairs of non-negative integers. A chess queen is standing at the point (x, y) and can move to any cell other with coordinates (x', y') if $x = x'$ or $y = y'$ or $x - y = x' - y'$ and both $x' \le x$ and $y' \le y$.



9	9	10	11	12	8	7	13	14	15	16
8	8	6	7	10	1	2	5	3	4	15
7	7	8	6	9	0	1	4	5	3	14
6	6	7	8	1	9	10	3	4	5	13
5	5	3	4	0	6	8	10	1	2	7
4	4	5	3	2	7	6	9	0	1	8
3	3	4	5	6	2	0	1	9	10	12
2	2	0	1	5	3	4	8	6	7	11
1	1	2	0	4	5	3	7	8	6	10
0	0	1	2	3	4	5	6	7	8	9
	0	1	2	3	4	5	6	7	8	9

Grundy value of a cell (x, y) is defined recursively and is equal to the smallest number g that does not appear among Grundy values of cells that can be reached from (x, y) in one move. The table above shows Grundy values for cells with coordinates up to 9.

You are given g and k. Consider all cells with Grundy values g. Let us order them all lexicographically: first by x-coordinate, then by y-coordinate. You have to find the k-th among them.

Input

The input file contains several test cases. Each test case consists of one line that contains two integers g and k ($0 \le g \le 10$, $1 \le k \le 500\,000$). There are at most 1000 tests. The last test case is followed by a line that contains two zeroes, it must not be processed.

Output

For each test case print two integers: x and y — the coordinates of the k-th lexicographically cell with Grundy value equal to g.

Examples

grundy.in	grundy.out
0 1 0 2 0 3 0 4 1 1 1 4 2 1 2 4 0 0	0 0 1 2 2 1 3 5 0 1 3 6 0 2 3 4

Problem H. Hardware Hashing

Input file:	`hardware.in`
Output file:	`hardware.out`
Time limit:	2.5 seconds
Memory limit:	512 mebibytes

Harry is working in Hlink company that is developing routing hardware for Hoogle. Recently he was asked to write a hashing software for routing chip HH-932.

Let us describe the architecture of HH-932. The chip has 4 megabytes $= 4 \times 2^{20}$ of read only flash memory that can contain the program that the chip will execute and the data. The processor of the chip has 256 registers named `r0` through `r255`, each register contains a 32-bit integer. Before the execution of the program each register except `r0` contains 0, and `r0` contains the input to the program.

The chip has one more special register called the *instruction pointer*. The program is stored in the flash memory and initially the instruction pointer is equal to 0. The execution of the program proceeds as follows. The instruction at the address equal to the instruction pointer is parsed and executed. If the instruction doesn't result in a jump, the instruction pointer is increased by the size of the parsed instruction and thus points to the byte in memory following the recently executed instruction.

Harry has a task to create a very fast hashing function for the given set of n integers $A = \{a_1, a_2, \ldots, a_n\}$ to the set of integers from 0 to $2n - 1$. The evaluation of the function must terminate after at most 30 instructions. Hashing function must not have collisions. Formally, it is required to write a program, that will satisfy the following conditions:

- If the program is executed with one of the integers from the set A, it must terminate after executing at most 30 instructions and return the value from 0 to $2n - 1$.
- For any two different values a_i and a_j from A the returned values must be different.

Harry is not strong in low level programming, so he asks you to help. He has sent you a table with possible instructions of HH-932 chip and their hexadecimal opcodes. He didn't give comments, so you have searched through internet for documentation, didn't find it, but found the following additional comments.

- All registers contain 32-bit integers.
- Addition, subtraction, and bitwise operations do not care whether integers are signed or unsigned.
- Conditional jumps interpret registers as signed integers.
- Result of the multiplication or division is a signed 64-bit integer that is saved into two registers. Register that gets the lower part of the result stores it as unsigned integer.
- Division and remainder use 64-bit integers as a dividend, the lower part is interpreted as unsigned, the upper part as signed. Divisor is signed.
- Division and remainder of signed integers is performed in the same way as in x86 architecture.
- All integers in the chip memory are saved lower byte first (little endian).

You have to create the contents of the flash memory for the chip, so that after it is executed as a program, the required hash function is evaluated and returned.

For any input from the set A your program must not address memory outside of 4M range and must not divide by zero. Instruction pointer must not get out of 4M range either.

Instructions table

Instr.	Size	Opcode	What instruction does
nop	1 byte	`00`	Do nothing
add	4 bytes	`01 r1 r2 r3`	`r3 := r1 + r2`
sub	4 bytes	`02 r1 r2 r3`	`r3 := r1 - r2`
mul	5 bytes	`03 r1 r2 r3 r4`	`(r3, r4) := r1 * r2` r3 gets the lower bits, r4 gets the higher bits of the product
div	6 bytes	`04 r1 r2 r3 r4 r5`	`(r3, r4) := (r1, r5) div r2` r1 contains the lower bits, r5 contains the higher bits of the dividend, r3 gets the lower bits, r4 gets the higher bits of the quotient
mod	5 bytes	`05 r1 r2 r3 r4`	`r3 := (r1, r4) mod r2`, r1 contains the lower bits, r4 contains the higher bits of the dividend
and	4 bytes	`10 r1 r2 r3`	`r3 := r1 and r2`
or	4 bytes	`11 r1 r2 r3`	`r3 := r1 or r2`
xor	4 bytes	`12 r1 r2 r3`	`r3 := r1 xor r2`
neg	2 bytes	`20 r1`	`r1 := -r1`
not	2 bytes	`21 r1`	`r1 := ~r1`
load	3 bytes	`30 r1 r2`	`r1 := memory[r2]` 4 bytes starting from address r2 are copied to register r1, lowest byte first
put	6 bytes	`31 r1 b0 b1 b2 b3`	`r1 := (b0, b1, b2, b3)` here b0 is the lower byte of the number put to the register, b3 — the higher byte
jmp	5 bytes	`40 b0 b1 b2 b3`	Jump to instruction at byte (b0, b1, b2, b3)
jz	6 bytes	`41 r1 b0 b1 b2 b3`	If `r1 == 0` jump to instruction at byte (b0, b1, b2, b3)
jnz	6 bytes	`42 r1 b0 b1 b2 b3`	If `r1 != 0` jump to instruction at byte (b0, b1, b2, b3)
jg	6 bytes	`43 r1 b0 b1 b2 b3`	If `r1 > 0` jump to instruction at byte (b0, b1, b2, b3)
jge	6 bytes	`44 r1 b0 b1 b2 b3`	If `r1 >= 0` jump to instruction at byte (b0, b1, b2, b3)
jl	6 bytes	`45 r1 b0 b1 b2 b3`	If `r1 < 0` jump to instruction at byte (b0, b1, b2, b3)
jle	6 bytes	`46 r1 b0 b1 b2 b3`	If `r1 <= 0` jump to instruction at byte (b0, b1, b2, b3)
ret	1 byte	`ff`	Return from program, the returned value is at `r0`

Input

The first line of the input file contains n ($1 \le n \le 100\,000$). The second line contains distinct n integers $a_1, a_2, \ldots, a_n$ ($1 \le a_i \le 10^9$).

Output

Output the contents of flash memory of the chip. You may output any number of bytes from 1 to 4 194 304. The rest of the memory is filled with zeroes. Each byte must be printed as a hexadecimal two-digit number. Do not print spaces.

Please output only the required prefix of the memory for your program, do not output trailing zeroes, to speed up testing.

Examples

hardware.in	hardware.out
3 2 3 9	3101040000000500010003ff
2 6 10	300000ff00000000000001000000

Explanations

In the first example the program is parsed and executed as follows:

Opcode	Instruction	What happens	What's going on
`31 01 04 00 00 00`	`put r1 4`	`r1 := 4`	`r0` = a_i, `r1 = 4`
`05 00 01 00 03`	`mod (r0, r3) r1 r0`	`r0 := (r0, r3) mod r1`	`r0` = a_i `mod 4`, `r1 = 4`
`ff`	`ret`	`return r0`	$a_i \bmod 4$ is returned

In the second example the program is parsed and executed as follows:

Opcode	Instruction	What happens	What's going on
`30 00 00`	`load r0 r0`	`r0 := memory[r0]`	`r0 = 0` if $a_i = 6$, `r0 = 1` if $a_i = 10$
`ff`	`ret`	`return r0`	0 or 1 is returned

Note that in addition to the program code, the memory in the second example contains some additional data which is used as return values.

Problem I. Interactive Proofs

Input file:	*standard input*
Output file:	*standard output*
Time limit:	2 seconds
Memory limit:	512 mebibytes

This is interactive problem.

A concept of interactive proof in complexity theory was introduced by Goldwasser, Micali and Rackoff in 1985. There are two parties: *Prover* and *Verifier* that have access ot a common word x and Verifier wants to determine whether x belongs to some set L, and Prover wants to convince Verifier that this is the case. Verifier is a randomized program that must work in polynomial time, but Prover's time and memory are not limited. Prover and Verifier interact by sending each other messages, and then Verifier either accepts, or rejects.

Verifier's goal is to accept only if x indeed belongs to L. Prover's goal is to force Verifier to accept. The proof system must generally satisfy the following conditions: if $x \in L$ Prover can make Verifier accept with high probability, if $x \notin L$ Verifier accepts with negligible probability no matter how hard the Prover tries.

In this problem we investigate the properties of one important interactive proof protocol for the $\#SAT$ problem.

Let φ be a boolean formula with n boolean variables denoted as $x_1, \ldots, x_n$. The allowed operations are *logical or* denoted as |, *logical and* denoted as &, and *logical not* denoted as !. Operations are listed from lowest precedence to highest, parenthesis can be used to change operations precedence as usually. Let us consider an example $\varphi_\oplus(x_1, x_2) = x_1\&x_2|!x_1\&!x_2$.

We will use 0 to denote *false* and 1 to denote *true* further on. The boolean vector $(a_1, \ldots, a_n)$ is said to satisfy φ if after setting $x_i = a_i$ for all i the value of φ is 1. For example, vectors $(0, 0)$ and $(1, 1)$ satisfy the formula $\varphi_\oplus$ above, and vectors $(1, 0)$ and $(0, 1)$ do not. The number of satisfying vectors is called the *weight* of φ and is denoted as $w(\varphi)$, so $w(\varphi_\oplus) = 2$.

We will now consider correct interaction protocol to prove that $w(\varphi) = k$. From the formal side we define the language $\#SAT = \{\langle\varphi, k\rangle | w(\varphi) = k\}$ and prove that $\langle\varphi, k\rangle \in \#SAT$.

First both Prover and Verifier pick a big engough prime p. All further calculations are performed modulo p. For the boolean formula φ denote its *arithmetization* $A(\varphi)$ recursively in the following way.

- If y is a variable, $A(y) = y$.
- If ψ is a formula and $\varphi = !\psi$, $A(\varphi) = 1 - A(\psi)$.
- If ξ and ψ are formulas and $\varphi = \xi\&\psi$, $A(\varphi) = A(\xi) \cdot A(\psi)$.
- If ξ and ψ are formulas and $\varphi = \xi|\psi$, $A(\varphi) = 1 - (1 - A(\xi)) \cdot (1 - A(\psi))$.

So, for example, $A(\varphi_\oplus) = 1 - (1 - x_1x_2)(1 - (1 - x_1)(1 - x_2))$.

It is easy to see, that if $(a_1, \ldots, a_n)$ is a satisfying assignment for φ then $A(\varphi)(a_1, \ldots, a_n) = 1$, and if it is not then $A(\varphi)(a_1, \ldots, a_n) = 0$. Therefore

$$w(\varphi) = \sum_{x_1=0}^{1} \sum_{x_2=0}^{1} \ldots \sum_{x_n=0}^{1} A(\varphi)(x_1, \ldots, x_n).$$

The proof that $w(\varphi) = k$ proceeds in n steps.

At the first step consider the function

$$P_1(x_1) = \sum_{x_2=0}^{1} \ldots \sum_{x_n=0}^{1} A(\varphi)(x_1, \ldots, x_n).$$

This function is a polynomial in x_1: $P_1(x_1) = c_{1,d_1}x_1^{d_1} + c_{1,d_1-1}x_1^{d_1-1} + \ldots + c_{1,1}x_1 + c_{1,0}$. Clearly, $P_1(0) + P_1(1) = k$. Prover sends Verifier P_1 by transmitting d_1 followed by $c_{1,d_1}, \ldots, c_{1,0}$. Verifier checks that $P_1(0) + P_1(1) = k$ and randomly chooses r_1 — integer from 0 to $p-1$. Verifier sends r_1 to Prover.

Now the second step starts. Consider the function

$$P_2(x_2) = \sum_{x_3=0}^{1} \ldots \sum_{x_n=0}^{1} A(\varphi)(r_1, x_2, \ldots, x_n).$$

Note that x_1 is substituted by r_1 as an argument for $A(\varphi)$. This function is again a polynomial in x_2, so Prover sends this polynomial to Verifier. Verifier checks that $P_2(0) + P_2(1) = P_1(r_1)$, and randomly chooses r_2 — integer form 0 to $p-1$. Verifier sends r_2 to Prover.

In general the i-th step for i from 2 to $n-1$ proceeds as follows. The function

$$P_i(x_i) = \sum_{x_{i+1}=0}^{1} \ldots \sum_{x_n=0}^{1} A(\varphi)(r_1, \ldots, r_{i-1}, x_i, x_{i+1}, \ldots, x_n)$$

is considered. This function is a polynomial in x_i, Prover sends this polynomial to Verifier. Verifier checks that $P_i(0) + P_i(1) = P_{i-1}(r_{i-1})$, randomly chooses $r_i \in \{0, \ldots, p-1\}$ and sends r_i to Prover.

The last step differs just a little. The function

$$P_n(x_n) = A(\varphi)(r_1, \ldots, r_{n-1}, x_n)$$

is sent to Verifier. Verifier checks that $P_n(0) + P_n(1) = P_{n-1}(r_{n-1})$, chooses random $r_n \in \{0, \ldots, p-1\}$ and checks whether $A(\varphi)(r_1, \ldots, r_n) = P_n(r_n)$.

Please note again, that all calculations are performed modulo p.

If all tests preformed by Verifier were successful, the proof is accepted. If at least one test failed, the proof is rejected. It can be shown that if the degree of $A(\varphi)$ is d, the probability that the proof is accepted but $w(\varphi) \neq k$ is at most $1 - (1 - d/p)^n$, so choosing p big enough this value can be made negligible.

The core idea of this proof protocol that allows to decrease the probability of Prover forcing Verifier to accept the wrong statement is substituting random integers modulo p instead of just 0-s and 1-s in arithmetization of φ. If Verifier only chooses 0 or 1 as r_i Prover can easily cheat Verifier and force it to accept with high probability even if $w(\varphi) \neq k$. This is exactly what you have to do in this problem.

Your program will act as Prover and try to force jury's Verifier to accept the incorrect statement $w(\varphi) = k$. Verifier will follow the protocol described above with one exception: it will choose $r_i \in \{0, 1\}$. We will use $p = 239$.

Interaction Protocol

Your program will have to run several proofs in each run. Your program will be accepted if it successfully forces Verifier to accept in at least 1/2 cases.

Each proof starts with your program reading n — the number of variables in φ, from standard input ($2 \le n \le 7$). If $n = 0$ that means that test run is over and your program must terminate. If $n > 0$ the proof must start.

The following line of standard input contains formula for φ. Formula uses the following characters: '`&`', '`|`', '`!`', '`(`', '`)`', and '`x`' followed by its 1-based index from 1 to n. Spaces can separate formula parts freely, but '`x`' is never separated from its index. Length of formula is at most 100 characters.

The following line contains k, your program must try to prove that $w(\varphi) = k$ even if that is not the case, following the above protocol ($0 \le k \le 2^n$).

After reading n, φ and k your program must start protocol by sending P_1 to Verifier.

It must print its degree d_1 followed by $c_{1,d_1}, \ldots, c_{1,0}$ to standard output. Separate numbers by spaces or new line characters, print new line character after the last number and flush standard output.

Degrees of all polynomials must not exceed 20. All coefficients must be between 0 and 238. Constant zero polynomial has degree 0 for the purpose of this problem.

Verifier checks the condition $P_1(0) + P_1(1) = k$ and prints r_1 that you must read from standard input, r_1 will be 0 or 1.

After that you must send P_2 in the same format, and read r_2, and so on.

After sending P_n you must not wait for any more input for this test case and proceed to next test case.

Verifier doesn't terminate the protocol even if one of the tests it performed failed. Therefore your program must also run all n rounds of the protocol even if it found out that it cannot cheat Verifier. However it is not allowed to print incorrect polynomials (with incorrect degree or incorrect coefficients).

For each test your program will be asked to make at least 64 and at most 128 proofs and will be accepted if it cheats Verifier in at least 1/2 of all cases for this test case. The only exception is sample test, in which there is only 1 run and the program will be accepted any way if it follows the interaction protocol.

It is guaranteed that Verifier's answers do not depend on the polynomials sent to it by Prover.

Examples

standard input	standard output
2 x1 & x2 | !x1 & !x2 3 0 0	 1 1 1 1 238 1

In the example above the protocol proceeds as follows. The correct $P_1(x_1)$ is $P_1(x_1) = 1$. However, Prover cannot send it to verifier because $P_1(0)+P_1(1) = 2$, but it needs to prove that $w(\varphi) = 3$ (which is incorrect, of course). So Prover sends incorrect polynomial $P_1(x_1) = x_1+1$ which satisfies $P_1(0)+P_1(1) = 3$. Verifier checks this and chooses $r_1 = 0$.

Now the Prover sends $P_2(x_2) = -x_2 + 1$, or, since all calculations are performed modulo 239, $P_2(x_2) = 238x_2 + 1$. Verifier checks that $P_2(0) + P_2(1) = P_1(0) = 1$. Now it chooses r_2 either 0 or 1 and checks whether $A_\varphi(0, r_2) = P_2(r_2)$. Fortunately for the Prover this is the case for both $r_2 = 0$ and $r_2 = 1$, so independently of Verifier's last choice it accepts. The goal is reached, the Verifier is cheated.

The last 0 at standard input indicates the end of the test run, so Prover terminates.

Problem J. Jackpot

Input file:	`jackpot.in`
Output file:	`jackpot.out`
Time limit:	3.5 seconds
Memory limit:	512 mebibytes

Johnny is on vacation in Las Figas. He has decided to visit the new casino *Lohal* which has opened there recently. Lohal offers the new game to its visitors, called *Black or White*. It is a single player game that proceeds as follows.

There are m tokens arranged in a line. The player is offered tokens one after another. Each token is either white, black, or diamond. To get a token the player needs to buy a card. There are several types of cards available.

The card of the i-th type has two numbers written on it: w_i and b_i, there are c_i such cards. To buy the card of the i-th type the player must pay $w_i + b_i$ dollars, but he can get a discount if he already owns some tokens.

That is, if the player has a white tokens, he can decrease the number w_i on the card by at most a (but he cannot make it negative). Similarly, if the player has b black tokens, he can decrease the the number b_i by at most b (but still cannot make it negative). Finally, for each diamond token the player can decrease by 1 any of w_i or b_i of his choice.

For example, if the player has 2 white, 1 black and 1 diamond token, he can buy the card that has $w_i = 4$ and $b_i = 0$ for 1 dollar: 2 white tokens decrease w_i to 2, using diamond token allows to decrease w_i to 1. Black token is useless for this card, so its cost now is $1 + 0 = 1$. The same set of tokens allows to get the card that has $w_i = 2$ and $b_i = 2$ for free (now we use diamond token to decrease b_i).

Note that buying a card doesn't require the player to discard the tokens, they stay at him and can later be used again to buy other cards.

The player can choose which tokens to get, but to win a jackpot he needs to get all tokens. Johnny wants to win a jackpot so he decided to get all tokens. Help him to choose which cards to buy to get tokens, so that he wins jackpot spending minimal possible sum.

Input

The input file contains several test cases.

Each test case starts with an integer m — number of tokens ($1 \le m \le 300$). The following line contains m characters, each of them is either '`W`' for white token, '`B`' for black token, or '`*`' for diamond token. The line lists tokens in order they are offered to the player.

The following line contains n — the number of card types ($1 \le n \le 300$). The following n lines contain three integers each: for each card type w_i, b_i and c_i are specified ($0 \le w_i, b_i \le 1000$, $1 \le c_i \le 300$). The sum of all c_i is at least m.

The last test case is followed by a line that contains 0. It must not be processed.

The sum of m for all test cases in one input file doesn't exceed 300. The sum of n for all test cases in one input file doesn't exceed 300.

Output

For each test case output one integer: the minimal sum Johnny must spend to win a jackpot.

Examples

jackpot.in	jackpot.out
5 WWB*W 3 0 1 1 1 0 1 1 2 3 0	4

The optimal strategy in the example is, for example, the following.

Token	**Card type**	**Player has**	**Cost**
W	1		$0 + 1 = 1$
W	2	W	$(1 - 1) + 0 = 0$
B	3	WW	$(1 - 1) + 2 = 2$
*	3	WWB	$(1 - 1) + (2 - 1) = 1$
W	3	WWB*	$(1 - 1) + (2 - 2) = 0$ (use diamond to decrease b_3)