Problem A. Silver Star Stands Alone

Source file name:	Alone.c, Alone.cpp, Alone.java, Alone.py
Input:	`Standard`
Output:	`Standard`

One of standard units of length used in Solar system distance measurements is one astronomical unit, abbreviated as AU. It is also used in the following problem.

The exploration probe is being assembled on Mars and it will be launched soon. The probe destination is an asteroid named Silver Star after its unexpectedly bright ultraviolet albedo. The journey of the probe will be long, and it may visit some other asteroids on the way. The set of candidate asteroids to be visited, including Silver Star, has been carefully chosen so that they are on the path from Mars to Silver Star and the sequence of their distances from Mars expressed in AU and arranged in ascending order, is the sequence of the first few prime numbers: 2, 3, 5, 7, 11, ... On the journey, not necessarily all asteroids between the launch site and Silver Star have to be visited. On the other hand, the distance between two consecutive visited asteroids can be at most 14 AU, otherwise the probe would not collect enough scientific data. There are two rules, neither of which can be broken: First, the closest asteroid, which is in distance 2 AU, will be visited, and second, Silver Star will be visited.

There are many possible trajectories of the probe dictated by the choice of particular asteroids to be visited. Two trajectories are considered different if the sequences of visited asteroids in each trajectory differ by at least one asteroid. Otherwise they are considered equal. Each trajectory contains the launch site asteroid and Silver Star. The probe will be moving steadily towards Silver Star, it will never turn back. The probe navigation team needs to evaluate the most promising trajectories. For that aim, they first need to calculate the number of all mutually different probe trajectories.

Input

The input contains one line with one integer P ($2 \leq P \leq 211$), the distance of the Silver Star from Mars expressed in AU. P is a prime number.

Output

Print the number of different probe trajectories.

Example

Input	Output
7	4
31	416

Problem B. TomTom Cruise

Source file name:	Cruise.c, Cruise.cpp, Cruise.java, Cruise.py
Input:	`Standard`
Output:	`Standard`

You intend to take a trip from your base to Ganymede. You have a spaceship which you will use for the whole trip. It will take some fuel to get there, some fuel to traverse part of the moon, and some fuel to get back to the base.

There are some points on Ganymede you can visit. The map of Ganymede is given as a graph where the weight of each vertex is equal to the amount of fuel it will take to traverse between the corresponding point and your base. The weight of each edge equals to the amount of fuel it will take to traverse between vertices which are connected by the edge, in either direction. In order for your trip to be reasonable, you do not want to visit any edge or any vertex twice. However, you have to traverse at least one edge.

What is the least amount of fuel you will need to accomplish such a trip?

Input

First line of the input contains two integers N and M, the number of vertices and edges, respectively $\left(1 \leq N \leq 10^4,\ 0 \leq M \leq \min\left(\binom{N}{2},\ 2 \cdot 10^4\right)\right)$. Next line contains N integers $v_0,\ v_1,\ \ldots\ ,\ v_{N-1}$ $\left(1 \leq v_i \leq 10^6\right)$, weights of respective vertices 0, 1, ..., $N-1$. Each of the next M lines contains three integers f_i, t_i, and w_i $\left(0 \leq f_i,\ t_i < N,\ 1 \leq w_i \leq 10^6\right)$, the two vertices connected by the edge and the edge weight, respectively.

Output

Output the minimum amount of fuel you have to spend on your trip.

Example

Input	Output
2 1 4 5 0 1 20	29
3 3 10 40 20 0 1 1 0 2 4 2 1 2	33

Problem C. Mad Diamond

Source file name:	Diamond.c, Diamond.cpp, Diamond.java, Diamond.py
Input:	`Standard`
Output:	`Standard`

Mars lacks global magnetic field. Advanced gadgets can detect weak local magnetic fields and serve as compasses for navigation.

A compass of this kind depends of movement of a highly magnetized diamond crystal in an intricate maze inside the gadget. The crystal is often called Mad Diamond because of the apparent complexity of its movement.

A compass maze is built on a system of concentric rings located on a flat disk with a single so-called base point on its perimeter. On each ring, there are located 360 so-called principal points in regular distances from each other. The angle of a principal point is the angle between two specific rays, measured in degrees. The first ray points from the maze center to the base point, the second ray points from the center of the maze to the principal point. The angle of the first principal point on a ring is 0°, the angle of the next principal point in the clockwise direction is 1°, and so on.

The maze itself consists of circular arc segments and straight radial segments. A circular arc segment is always completely embedded in one of the rings. It begins in a principal point and it ends also in a principal point. Each radial segment connects two neighbour rings and it is perpendicular to the tangents of the rings at the respective points of connection. Often, at least one endpoint of a radial segment belongs to a circular arc segment too, but that is not a strict necessity. Any two segments can intersect only at a principal point. No two segments overlap, not even partially.

Before calibration of a compass is started, two points in the maze are given, a principal start point and a principal end point. The diamond is located in the principal start point. The diamond has to travel from the principal start point, move along the segments in the maze, and finally stop at the principal end point.

During calibration, the compass is hanged vertically on a wall, with the base point initially at the top of the compass. Calibration proceeds in phases. In a phase, the compass remains fixed, and the diamond falls through the maze, from its previous position to the lowest currently reachable point in the maze. The phase is terminated when the diamond stops. The diamond may also remain still during a phase, if its position in the maze prevents it from movement. Between each two consecutive phases, the compass is rotated clockwise or counterclockwise by 1°.

Most of the time, due to the maze shape, the diamond cannot move exactly vertically. At any point of time, the vector of diamond movement can deviate from vertical vector by at most 89°. Note that at the end of each phase the diamond is in some principal point.

When the diamond arrives at a principal point from which two segments continue currently downwards, it always checks the radial segment. If the current vector of diamond movement through the radial segment would deviate from the vertical by at most 45°, the diamond falls into the segment and continues its movement down through it. Otherwise, the diamond continues its present movement through the circular segment.

Input

The first line contains a single number N ($1 \leq N \leq 20$), the number of rings in the maze. Then, N pairs of lines follow. Each pair describes one ring, in the order from the center of the maze (the rings are numbered 0, 1, ..., $N-1$). The first line in a pair starts with number K, the number of maze circular arc segments on the current ring. It is followed by K pairs of numbers (X and Y) representing an arc starting in a principal point whose angle is X and ending in principal point whose angle is Y. The second line in the pair contains integer L, the number of radial segments leading from the current circle out to

the next ring. This number is followed by L values, the angles of principal points on which the radial segments are located.

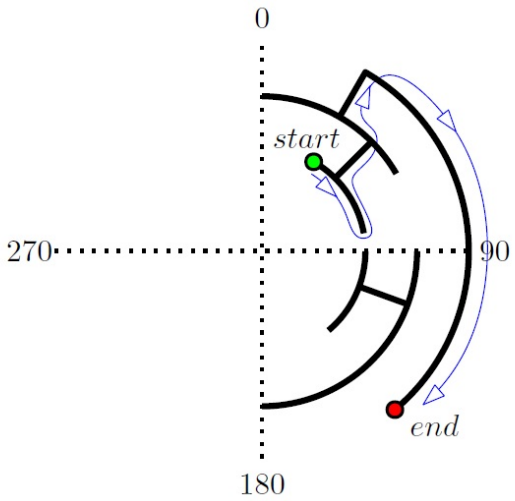
Last two lines of input represent the principal start point and the principal end point, respectively. Each point is described by the number of ring it lies on and the angle of the principal point it coincides with.

Output

Print the minimum number of compass rotations between the phases which bring the diamond from the principal start point to the principal end point. The diamond must be standing still in the principal end point at the end of the last phase. Print the string "`Impossible`" if the diamond cannot be brought to the principal end point at the end of any phase.

Example

Input	Output
3 2 30 80 90 140 2 45 110 2 0 60 90 180 1 30 1 30 140 0 0 30 2 140	260



Example input

Problem D. Burizon Fort

Source file name:	Fort.c, Fort.cpp, Fort.java, Fort.py
Input:	`Standard`
Output:	`Standard`

On Mars there is a huge complex of forts. Martians living in these forts are fond of eating Burizons and get very grumpy if they run out of them. Therefore it's quite common task to transport Burizons among these forts when the supplies are running low somewhere.

The transportation of Burizons is not so simple because of all the space debris falling down on the planet. If some space debris hits the transportation vehicle, not only the vehicle gets destroyed, but even all the supplies transported. That's why Martians can transport their Burizon supplies only at very precisely timed moments.

Luckily, Martian scientists developed a way to predict practical times when no space debris will be endangering the transport vehicles. For this matter Martians developed a Martian UNIX timestamp, which is represented as a positive integer. Practical time for transportation is a Martian UNIX timestamp m, such that each positive integer smaller than m can be represented as a sum of distinct divisors of m.

As recognizing practical time for transport is not that easy for Martians, they will need you to help them to figure out which of the given timestamps are practical ones.

Input

The first line contains one integer T $(1 \leq T \leq 100)$, the number of timestamps to be checked. Each of the next T lines consists of one integer m $\left(1 \leq m \leq 10^{12}\right)$, the Martian UNIX timestamp.

Output

Output T lines, each with either "`Yes`" or "`No`", answering the question whether the i-th timestamp is practical.

Example

Input	Output
5 4 6 10 12 15	Yes Yes No Yes No

Problem E. Terrace Hill

Source file name:	Hill.c, Hill.cpp, Hill.java, Hill.py
Input:	`Standard`
Output:	`Standard`

All explored mountain terraces in Girotti hills in Charitum Montes on the southern Martian hemisphere have a peculiar feature – their sizes are approximately equal, and they all lie on a hypothetical straight line.

The flat surface of the terraces is ideal for future housing development. Unusual configuration of the terraces allows for a daring engineering project which will connect some terraces by bridges. Due to relative geological instability in the surrounding region, the surface of any two terraces connected by a bridge has to be in the same height. Obviously, a bridge between two terraces can be built only when the height of all terraces between the given two is less than the height of the two terraces to be connected.

The engineers of the project want to know the maximum total length of all bridges that can be built. To simplify the introductory calculations, the following assumptions are made. The distance between two neighbour terraces is negligibly small, it is considered to be zero in all cases. The width of a terrace is considered to be one length unit.

Input

The first line contains an integer N $\left(1 \leq N \leq 3 \cdot 10^5\right)$ the number of the terraces. The second line contains N integers a_1, a_2, …, a_N $\left(1 \leq a_i \leq 10^6\right)$ where a_i is the height of i-th terrace. The heights are given in the order of terraces on the (hypothetical) line.

Output

Print one integer – the maximum possible total length of all bridges.

Example

Input	Output
5 1 2 3 3 1	0
6 5 5 5 3 2 3	1
6 2 3 2 1 2 3	4

Problem F. Eidam-Sand Lair

Source file name:	Lair.c, Lair.cpp, Lair.java, Lair.py
Input:	`Standard`
Output:	`Standard`

In your base you have dug through sand and rock to construct a deep pit with stairs and a lift. The pit goes very deep into the ground where on various floors you leave equipment and on the deepest levels you leave cheeses to age, particularly Eidam. You often take a stroll downstairs, but wonder whether it is faster to go up to the surface by foot or take a lift for a part or all of the trip.

You know the position and the speed of yourself and the lift. The surface floor is numbered 0, positive numbers denote underground floors by their distance to the surface. You are alone on the Mars so only you will use the lift. Also, it takes (you and lift) almost no time to start and stop moving so you will neglect these. When the lift is called to multiple floors, it moves to them in the same order in which it was called. The lift is not very special – once you enter it, you may order it to go to an arbitrary floor. In such case, any previous calls must be processed first.

Input

The first input line contains a single integer T $\left(1 \leq T \leq 10^4\right)$, the number of test cases. Next T lines contain one test case each. Each test case is composed of four integers: Y_p, L_p, Y_s, L_s where $0 \leq Y_p,\ L_p \leq 10^9$ are the initial floors of you and the lift, and $0 < Y_s,\ L_s \leq 10^6$ are the times it takes you and of the lift to move by one floor, respectively.

Output

For each test case, print a single integer – the minimum time it takes you to move from your initial floor to the 0-th floor.

Example

Input	Output
2 2 20 10 2 10 20 10 2	20 40

Explanation

Example Input contains two test cases. In the first one, you will rather walk to 0-th floor than to wait for the lift which is 18 floors away. In the second test case, an example of the fastest way is when you call the lift and walk 1 floor upstairs in the meantime, then you call the lift to 9-th underground floor and wait for the lift which you then take to the 0-th floor.

Problem G. Organ-free Man

Source file name:	Organ.c, Organ.cpp, Organ.java, Organ.py
Input:	`Standard`
Output:	`Standard`

Every so often, a shipment of universal robots comes from Earth to Mars in order to help you with routine colonization tasks. The robots are called Organ-free Men (precisely OFMv5001.41.912) and each one of them is identified by a unique serial number, which is a positive integer.

To prevent space theft, OFMs are transported from Earth to Mars in a locked state and have to be first unlocked by a special password. The password to unlock an OFM is based on its serial number and a function $f(x)$. The function $f(x)$ is defined as follows:

If $0 \leq x \leq 9$, then $f(x) = x!$, and if $x > 9$, then $f(x) = (x \bmod 10)! + f(\lfloor x/10 \rfloor)$. The brackets $\lfloor\ \rfloor$ denote the floor value of a number (e.g. $\lfloor 2.43 \rfloor = 2$). Exclamation mark denotes the factorial, i.e., $x! = 1 \cdot 2 \cdot \ldots \cdot x$ for $x > 0$ and $0! = 1$.

To unlock an OFM with a serial number y, you need to input smallest such non-negative integer x, so that $f(x) = y$ holds.

Will you manage to unlock all robots that were shipped to you?

Input

The input consists of one integer y $\left(1 \leq y \leq 10^9\right)$, the serial number of a particular OFM.

Output

Output a single non-negative integer x, the password to unlock the particular OFM.

Example

Input	Output
3	12
20	2333

Problem H. Bread Pit

Source file name:	Pit.c, Pit.cpp, Pit.java, Pit.py
Input:	`Standard`
Output:	`Standard`

New breeds of underground bacteria help in terraforming Mars. The bacteria feed mainly on regular bread loafs which are baked in huge quantities on the Mars surface and subsequently dropped into a specially constructed pit. To distribute the bread evenly in the subsurface, the pit consists of nearly vertical tunnels arranged in a tree-like structure. Each tunnel ends either in a cave filled with bacterial colonies or in a gate which connects into one or more other downward tunnels. The gates are mechanized and each of them keeps open only one downward tunnel at a time. When a loaf falls through a gate into a downward tunnel, the gate closes the tunnel and opens another one, to which it is connected, for the next arriving loaf. Opening the downward tunnels works in a cyclic fashion: When a gate closes the last downward tunnel it again opens the one which was open first. The order of open downward tunnels in each gate is fixed. At most one of the gates is on the surface. All loafs falling through at least one tunnel also fall through this topmost gate. The exception to the described scheme is the situation when the topmost gate is completely closed for maintenace, all tunnels become inaccessible and the loafs remain at the surface. In this scenario, for formal reasons, the surface is considered to be a cave and simutaneously the only node in the whole pit.

When the system started operating, before any bread loaf was deposited in it, the first downward tunnel in each gate was open.

Both caves and gates are commonly denoted as nodes, each node is labeled by a unique integer.

Determine which cave did each bread loaf fall into, as they were subsequently dropped, one after another, into the pit.

Input

The first input line contains two integers N, Q $\left(1 \leq N,\ Q \leq 3 \cdot 10^5\right)$, the number of all nodes (gates and caves) in the pit and the number of bread loafs dropped into the pit. The nodes are labeled by integers 0, 1, ..., $N-1$, the surface gate node is labeled by 0. The second line contains $N-1$ integers. The value of i-th integer on the line is the label of the predecessor of node i. The predecessor of a node is the closest gate from which a falling loaf arrives to node i. The second input line also encodes the order of open downward tunnels in each gate. When value X appears on j-th and k-th positions and $j < k$, the tunnel connecting X to j opens before the tunnel connecting X to k opens.

Output

Print Q lines. The i-th line should contain the label of the cave that received the i-th loaf dropped into the pit.

Example

Input	Output
5 5 0 0 1 1	3 2 4 2 3
7 10 0 0 0 2 2 2	1 4 3 1 5 3 1 6 3 1

Problem I. Cyanide Rivers

Source file name:	Rivers.c, Rivers.cpp, Rivers.java, Rivers.py
Input:	`Standard`
Output:	`Standard`

Cyanide rivers flowing out from Martian south polar ice cap are quite dangerous due to their toxic contents and any activity in their close proximity is often extremely time consuming.

A row of communication towers has been built in the area, before the rivers even appeared after Martian global warming events.

Some of the towers are now standing directly in a river, some remain standing outside, on the river shores or on islands in the rivers. The first and the last tower in the row stand on river shores.

All towers are to be officially certified for the next operation period in the present difficult conditions.

The towers which are standing on the shore or on an island can be certified immediately. The access to the towers in rivers is hazardous and requires a lot of caution. The certification process of a tower standing in a river takes one whole day. Moreover, a tower standing in a river can be certified only if at least one of its immediate neighbour towers has been certified at least one day earlier. Fortunately, the certification process can be performed independently on each tower, thus it is possible to certify more than one tower in a day.

The certification process has to be completed as soon as possible.

Input

The input consists of one line containing an odd binary number with up to 300000 digits and with no leading zeros. Each digit represents one tower. Towers standing in a river are represented by 0's, remaining towers are represented by 1's. The order of the digits is the same as the order of towers in the row.

Output

Print minimum number of whole days in which all towers can be certified.

Example

Input	Output
10100110101	1
10000010010001	3

Problem J. Screamers in the Storm

Source file name:	Screamers.c, Screamers.cpp, Screamers.java, Screamers.py
Input:	`Standard`
Output:	`Standard`

On Mars, though its atmosphere is very thin, winds can blow with unexpected force. The effect can grow even stronger in narrow and deep canyons close to the equator. The floor of one of these canyons is extremely flat and it is used for transport because its steep walls provide partial shielding from cosmic radiation. Over the time, sand has accumulated in the canyon. It is blown by the wind into sizeable sand dunes which obstruct the transport. The dunes form one long string of dunes stretched along the canyon floor.

The arrangement of the dunes is not stable. Raging sand storms often remodel the dunes into a different string. Violent movement of sand masses in the storm produce an eerie sound for which the dunes are known as "Screamers in the storm".

Detailed measurements of the dunes revealed that the height of adjacent dunes, expressed in Martian meters, is always a small positive integer. Moreover, probably due to wind interference effects in the canyon, the heights of two adjacent dunes are different, unless they are both 1. In fact, the heights of two adjacent dunes are always relatively prime, that is, they share no common factor bigger than 1.

To model the dunes behavior, the number of all possible configurations of the dunes in the string has to be established.

Input

The input contains one line with two integers K, N $\left(1 \leq K \leq 66,\ 1 \leq N \leq 10^{18}\right)$, the maximum possible height of a dune and the number of dunes in the string of dunes.

Output

Print the number of different strings of dunes. Output the result modulo $10^9 + 7$.

Example

Input	Output
3 4	41
2 4	8
42 75097099101114	673977658

Problem K. Win Diesel

Source file name:	Win.c, Win.cpp, Win.java, Win.py
Input:	`Standard`
Output:	`Standard`

Exploring cave systems on Mars is an easy procedure which was pre-planned on Earth. Each cave system contains several cave rooms which are pairwise disconnected. As the cave rooms are not accessible from the surface, you were given a digging rig so that you are able to dig channels between some of them.

The soil is not uniform and running the digging rig is extremely diesel intensive. To save diesel you need to excavate the minimum number of channels so that every cave is accessible from the surface through the channels. At the same time, the resulting system should be connected in a way so that the number of channels you need to pass to get from the surface to any particular cave is minimized.

You ran a scan over the whole cave system and now you know the danger level of each cave room. The scan also shows which pairs of cave rooms, or which cave room and surface, are separated by soil that can be dug through to create a channel. Let the distance of a cave room be the minimum possible number of channels that need to be traversed to reach the room from the surface, if all the possible channels were dug.

Due to safety reasons, the procedure prescribes that in the process of digging, the closest (least distant) cave rooms must be made accessible before any more distant cave rooms are made accessible. If this is not uniquely determining the next room to be made accessible, then the safest (least dangerous) room should be chosen from the candidates. If this does not uniquely determine the channel that needs to be dug, then out of already accessible starting locations the safest one has to be chosen.

Discovering the rooms in the given order forces you to traverse through the dug channels multiple times. The machine you use to move the digging rig also drinks diesel, although much less than the digging rig. You wonder, what is the amount of diesel that you will need to do this job – this is proportional to the number of channels that will need to be traversed, starting from 0 and counting until all the channels are dug.

Input

The first line contains two integers N and M $\left(1 \leq N \leq 2 \cdot 10^5,\ 0 \leq M \leq 2 \cdot 10^5\right)$ the number of cave rooms (including surface) and the number of possible cave channels, respectively. Surface is denoted by 0 and the cave rooms are denoted by integers from 1 to $N-1$ and are ordered by increasing danger level. Each of the next M lines contains two integers, representing two cave rooms or a cave room and surface, which are separated by soil soft enough so that it can be dug through to create a connecting channel.

Output

Print a single integer – the total number of channel traversals you need to perform to make the whole cave system accessible according to the prescribed procedure.

Example

Input	Output
5 5 0 1 1 2 2 3 3 4 4 0	10
5 4 0 1 1 2 2 3 3 4	4
5 6 0 1 0 2 0 3 0 4 1 4 2 3	7