1 Maximum Repeating Substring

You must find a substring of an input string, such that the substring consecutively repeats the most number of times.

A string s is said to be a *substring* of string t if the characters of s appear consecutively in order within t. For example, "BAT" is a substring of "ACROBATS" since these characters occur in the same order as follows "ACROBATS".

A substring repeats consecutively if it appears multiple times in a string, with no gaps between its occurrences. For example, the string "BLAAABADBADX" has two repeating substrings: 'A', which repeats three times, and 'BAD', which repeats twice. The answer reported is '3', since the length of the repeating substring is not considered. Only consecutive repetitions of a substring are counted, so the last two occurrences of 'A' don't count toward the total number of repetitions.

Input format

The input will be a single line of text containing the input string. Assume the longest string will be 50 characters.

Output format

Output "Maximum repeat: " and the maximum number of times a substring is consecutively repeated.

Examples

Input:	**Output:**
`blaaabadbadx`	`Maximum repeat: 3`

Input:	**Output:**
`ababcdcdcdcdef`	`Maximum repeat: 4`

2 Line Interval Density

You and your friends Mary and Pete (two social studies students) are walking around Disneyworld (you've suckered your parents into taking you all there during the spring break) when all of a sudden you chance upon this great game booth. Intending to impress your companions and win over a new love interest, you decide to win a stuffed animal in the game.

Since you have your notebook computer with you (don't leave home without it!) a brilliant idea flashes through your mind. In order to ensure your success and to show off your new data structures and algorithms skills to these two liberal-arts people (who, by the way know nothing about either algorithms or data structures), you decide to come up with a program that will ASSURE you of winning.

The game gives you a set of intervals on a line, and in order to win, you must correctly and quickly determine the *density* of the intervals (i.e., what is the maximum number of intervals that overlap at any given point on the line).

The vendor in charge of the game agrees to allow you to use your computer to help you win the game, but agrees only on the conditions that for each set of intervals, all you can tell the computer is how many intervals will be in this set, and what the intervals are. The vendor even gives you a few examples of problems he's used in the past (he is getting pretty intrigued with this computer idea himself!).

The vendor gives you one last warning before you begin your programming — the intervals he uses are half-open. This means that the interval [2,4) includes 2 but not 4. Therefore, if you had two intervals [2,4) and [4,7), there would be a density of 1 — no overlap.

Good luck winning the game and your new love!

Input Format

The input will consist of the number of intervals, followed by a list of half-open intervals [l,r) giving the ranges of each interval. All intervals are in integers. Assume the maximum number of intervals is 100, and the range of each interval is between 1 and 20.

Output Format

Output "Maximum density: " and the maximum line interval density found.

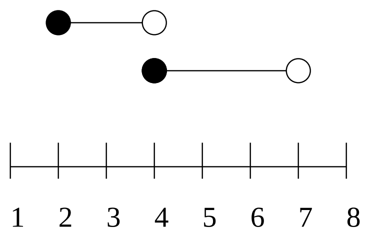
Examples

Input:

```
2
2 4
4 7
```

Output:

```
Maximum density: 1
```

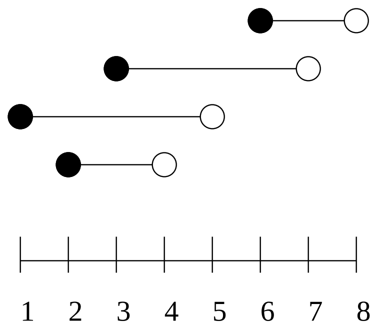


Input:

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

Output:

```
Maximum density: 3
```



3 Garbage Collection

A common problem faced by programmers is the need to reuse dynamically allocated memory when it is no longer needed. One approach is to automatically detect such pieces of unusable memory (*garbage*) by tracking program pointers. This technique, known as *garbage collection*, is used in languages such as Lisp and Java.

As a systems programmer, it is your job to track statements executed by a program and determine what memory may be reused. All memory is allocated as *cells* which contain two pointers each (1&2). Cell pointers initially point nowhere, but may be pointed at specific cells through assignment statements.

A cell is considered to be *garbage* when it can no longer be accessed by a pointer from cell 1 (the main program's cell), or through a pointer from some cell accessible from cell 1. Cell 1 is never considered to be garbage. Your job is to track a number of assignment statements to cells, then decide which cells are garbage and may be reused.

Input Format

The first line of input will be n, the number of cells. The cells are numbered from 1 to n. Remaining lines will be a list of ordered triples (x, y, z) specifying an assignment of cell x's pointer y to cell z. The list is terminated by a triple where $x = 0$. You may assume there are a maximum of 50 cells.

Output Format

The output consists of one line. Output "Garbage cells: " and the list of garbage cells found, in order from lowest to highest.

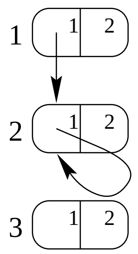
Examples

Input:

```
3
1 1 2
2 1 1
2 1 2
0 0 0
```

Output:

```
Garbage cells:  3
```

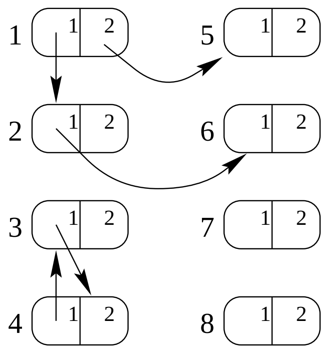


Input:

```
8
1 1 2
1 2 7
2 1 6
1 2 5
3 1 4
4 1 3
0 0 0
```

Output:

```
Garbage cells:   3  4  7  8
```



4 ASCII Art

ASCII art refers to a picture that is made up from only letters, numbers, punctuation marks and symbols. ASCII (pronounced "as-kee") stands for the America Standard Code for Information Interchange, is the character code used on most modern computers. Here is an example of using ASCII art to write a word "*TEST*". Notice how using different letters (for example, "`.`" versus "`$`") affects shading.

```
..............lllllllllll.llllllllll.llllllll.lllllllllll...............
.................lLl.....lLl.......lLl....lLl....lLl....................
................LlL.....LlL.......LlL...........LlL.....................
...............L$L.....L$LLlLL$..L$LLlLL$LL....L$L......................
..............L$L.....L$L..............L$L....L$L.......................
.............$L$.....$L$.......$L$....$L$....$L$........................
............$$$.....$$$$$$$$$$.$$$$$$$$.....$$$.........................
```

For this problem, your program will be given a black-and-white image, represented as a 2-dimensional array of real numbers. Each number is called a *pixel*. The pixel value 0 means black, 1 means white, and values in between 0 and 1 are shades of gray. Your program should convert this image into a collection of ASCII characters that approximate the gray values. The mapping from gray-scale values to characters is given below. (This assumes that you are displaying white characters on a black background.) Let x denote the gray-scale value.

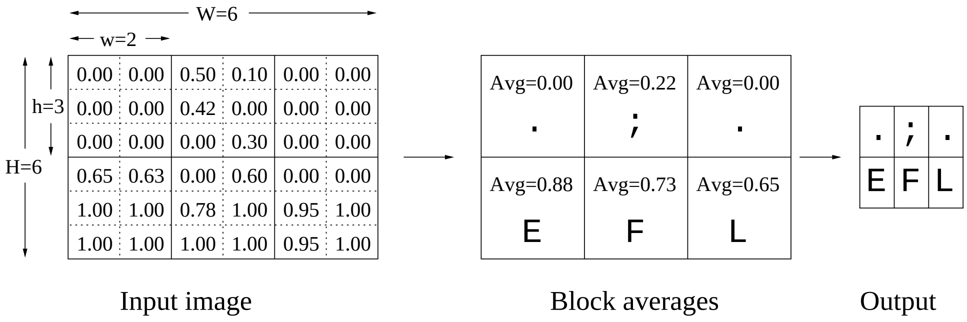
$0.0 \le x < 0.1$	write '`.`'	$0.5 \le x < 0.6$	write '`l`'
$0.1 \le x < 0.2$	write '`,`'	$0.6 \le x < 0.7$	write '`L`'
$0.2 \le x < 0.3$	write '`;`'	$0.7 \le x < 0.8$	write '`F`'
$0.3 \le x < 0.4$	write '`!`'	$0.8 \le x < 0.9$	write '`E`'
$0.4 \le x < 0.5$	write '`v`'	$0.9 \le x \le 1.0$	write '`$`'

Because most images use pixels that are square, but a character displayed on a typical screen is not square (it is taller than wide), your program will not necessarily convert pixels to characters on a 1-to-1 basis. Instead, you will be given a *block size*, specified by two integers h and w. For each block of pixels of size $h \times w$, you will compute the *average* gray-scale value of the pixels in this block (by summing and dividing by the number of pixels in the block) and then use this average value to determine which character to print. You may assume that the image height and width are even multiples of the block height and width, respectively.

The example below illustrates this. The image size is 6×6, and the block size is 3×2. The image is broken into blocks of size 3×2. The gray values within each block are averaged, and then are converted to the appropriate character. The final output is shown on the right.

Input format

The first line of the input contains two positive integers, the height H and width W of the image. You may assume that the height and width are each at most 100. The next line contains two positive integers, the block height h and block width w. The rest of the file consists of H lines of W reals. These are the gray values of the image. They are given row by row, from top to bottom, and within each row horizontally from left to right.



Output format

The output consists of exactly H *div* h lines, each containing exactly W *div* w characters (followed immediately by the end of line). There should be no initial or trailing blanks on each output line.

Examples

Input:

```
6 6
3 2
0.00 0.00 0.50 0.10 0.00 0.00
0.00 0.00 0.42 0.00 0.00 0.00
0.00 0.00 0.00 0.30 0.00 0.00
0.65 0.63 0.00 0.60 0.00 0.00
1.00 1.00 0.78 1.00 0.95 1.00
1.00 1.00 1.00 1.00 0.95 1.00
```

Output:

```
.;.
EFL
```

Input:

```
7 13
1 1
0.0 0.0 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5
0.0 0.0 0.0 0.0 0.0 0.5 0.6 0.5 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.6 0.5 0.6 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.6 0.9 0.6 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.6 0.9 0.6 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.9 0.6 0.9 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.9 0.9 0.9 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
```

Output:

```
..lllllllllll
.....lLl.....
....LlL......
...L$L.......
..L$L........
.$L$.........
$$$..........
```

5 String Similarity

Finding similarities in character strings is an important problem in text processing and data mining. It has applications in genetics research as well, since strands of DNA can be expressed as very long strings of the characters C, G, T, and A.

A string s is said to be a *subsequence* of string t if the characters of s appear in order within t, but possibly with gaps between occurrences of each character. For example, "COAT" is a subsequence of "ACROBATS" since these characters occur in the same order within this string "ACROBATS". "COBRA" is *not* a subsequence of "ACROBATS", because even though all the characters appear, they do not appear in the proper order, and "COBBS" is *not* a subsequence because a character cannot be used twice.

To make this formal, break s and t into their individual characters as: $s = s_1 s_2 \ldots s_m$ and let $t = t_1 t_2 \ldots t_n$. We define s to be a *subsequence* of t if there exists a sequence of integers $1 \leq j_1 < j_2 < \ldots < j_m \leq n$ such that for $1 \leq i \leq m$, we have $s_i = t_{j_i}$.

One common way to measure the similarity of two character strings is to determine the length of their *longest matching subsequence*. For example, the sequences "ABRACADABRA" and "YABBADABBADOO" share the subsequence "ABADABA", whose length is 7.

ABRACADABRA YABBADABBADOO

For this problem you will input two character strings and output their longest matching subsequence and its length. In some cases there may be more than one longest subsequence, in which case you may output any of them. (For example, given the strings "ABCD" and "ACBD" your program could output *either* "ABD" or "ACD".)

Input format

The input consists of two lines, each containing a string. Each string will contain from 0 to at most 15 characters.

Output format

The output consists of four lines. On the first and second lines, output the two input strings after the headers "First string: " and "Second string: ", respectively. On the third line output "Similarity: " and the length of the longest matching subsequence. On the second line output "LMS: " and the longest matching subsequence.

Example

Input:

```
ABRACADABRA
YABBADABBADOO
```

Output:

```
First string: ABRACADABRA
Second string: YABBADABBADOO
Similarity:  7
LMS: ABADABA
```

6 Campaign Trip Planning

As a presidential candidate, you need to cover the most territory in the least amount of time, blazing through miles of backwoods America on your way from one big city to another in an effort drum up enough support to win the election and make the massive amounts of money you showered on your campaign ads! Your staff is competent enough to find out what cities are connected to what others and how much time you are likely to have to spend traveling both within and between the cities, but they aren't quite competent enough to figure out what the best way for you to go would be.

However, since you've been well educated in the ways of data structures and algorithms, you are wise about these sorts of things. You know programming. To circumvent your staff, you decide to take matters into your own hands and write a program to find the shortest routes between two cities between which you need to travel, including time spent in all the cities on the route. Beware, it may not always be possible to travel from city 1 to city n.

Good luck with the campaign!

Input Format

The first line of input will be n, the number of cities. The second line will be n integers showing the delay for city $1 \dots n$. During the trip, delays apply to cities 1 and n as well as all intermediate cities. The remaining lines with be a list of ordered triples (x, y, d) indicating a path from city x to city y with delay d. You may also travel from city y to city x with the same delay. All delays are integers. The end of the list will be a triple where $x = 0$.

Your goal is to find the quickest time to get from city 1 to city n. You do not have to visit all cities. You may assume all delays within a city or between cities will be less than or equal to 100, and the maximum number of cities is 100.

Output Format

The output consists of two lines. If no path exists from city 1 to city n, output "No path found". Otherwise, on the first line output "Shortest time: " and the shortest time from city 1 to city n. On the second line output "Shortest path: ", followed by the list of cities on the shortest path in the order visitied, starting from city 1 and ending at city n. If there are multiple shortest paths, you may output any such path.

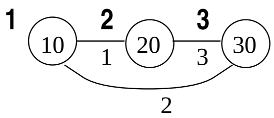
Examples

Input:

```
3
10 20 30
1 2 1
2 3 3
3 1 2
0 0 0
```

Output:

```
Shortest time: 42
Shortest path: 1 3
```

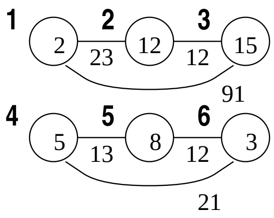


Input:

```
6
2 12 15 5 8 3
1 2 23
1 3 91
3 2 12
6 4 21
6 5 12
4 5 13
0 0 0
```

Output:

```
No path found
```



7 Thread Partitioning

A technique for making programs run faster is to first divide the work in the program into a number of *threads* (tasks), then to assign the threads to a number of *nodes* (computers) to be executed in parallel. A problem arises because threads need to communicate with each other. When two threads are assigned to different computers, the communication takes place across a computer network and will incur a *communication cost.*

Your problem is to minimize communication cost between threads across the network. Each node must be assigned the same number of threads. You must find an optimal partition (assignment) of threads to nodes such that the total communication cost is minimized. The communication cost of a partition of threads to nodes is defined as the sum of the communication costs of each pair of threads that are assigned to different nodes. To balance the network load, you must assign equal numbers of threads to each node.

For example, your problem may involve partitioning 4 threads to 2 nodes, where the communication cost between each pair of threads is shown in the table below. Since the same number of threads must be assigned to each node, each node must be assigned 2 threads. If we put threads 1&3 on the first node, and threads 2&4 on the second node, the communication cost for the partition will be:

cost(1,2) + cost(1,4) + cost(2,3) + cost(3,4) = 14 + 2 + 30 + 39 = 85

The costs for 1,3 and 2,4 were not included because these pairs of threads are assigned to the same node. In comparison, if we put threads 1&2 on the first node, and threads 3&4 on the second node, the communication cost for the partition will be:

cost(1,3) + cost(1,4) + cost(2,3) + cost(2,4) = 27 + 2 + 30 + 11 = 70

The second partition is thus less expensive.

Input format

The first line of integers consists of a pair of integers (t, n), where t is the number of threads and n is the number of nodes. There will be an additional t lines of input, providing a $t \times t$ grid of communication costs between threads, where the entry in position $[x, y]$ represents the communication cost between thread x and y if they are placed on separate nodes. All costs will be positive integers, and the grid will be symmetric (i.e., cost of $[a, b]$ will be same as $[b, a]$). You may assume that there are at most 10 threads and 10 nodes, and the number of nodes always evenly divides the number of threads.

Output format

The output consists of one line. Output "Communication cost: " and the communication cost of the best thread mapping.

Example

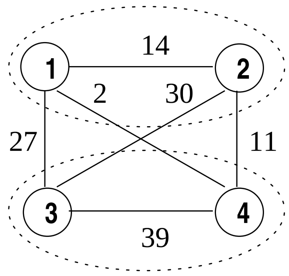
Input:

```
4 2
 0 14 27  2
14  0 30 11
27 30  0 39
 2 11 39  0
```

Output:

```
Communication cost: 70
```

	Threads			
Tds	1	2	3	4
1	0	14	27	2
2	14	0	30	11
3	27	30	0	39
4	2	11	39	0



8 Asteroid Tracking

(Basis for J. Robert Dorfman Prize for Numerical Computation)

N-body simulation is an important technique used by scientists and engineers to iteratively compute the motions of a number of interacting objects. Objects may range from stars (galaxy formation) to molecules (chemical reactions). We adapt a similar method to track the path of an asteroid as it approaches Earth.

Panic is spreading in world governments as an asteroid approaches the Earth. Your task is to take the position and velocity of the asteroid, along with the gravitational attraction of the Earth, and compute the course of the asteroid. To refresh your memory, you ask your White House science advisor for a quick physics lesson. The key points of motion in a two-dimensional space are:

- Consider two points A, B in space at locations (x_1, y_1) and (x_2, y_2). The separation from A to B can be represented by the vector $\vec{D} = (x_2 - x_1, y_2 - y_1)$ containing both the direction and the distance as the magnitude of the vector $|\vec{D}| = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$.
- Motion in a two-dimensional plane will have components along both the x-axis and the y-axis. Velocity can thus be described as a vector $\vec{V} = (x, y)$, where x, y are the speeds along the x-axis and y-axis, respectively. The overall speed is the magnitude of the vector $|\vec{V}| = \sqrt{x^2 + y^2}$.
- Earth will pull the asteroid towards itself with an attraction g proportional to its mass divided by the square of their distance $g = \frac{M}{|\vec{D}|^2}$. The gravitational force will be composed of a vector $\vec{G} = (x, y)$ pointing from the asteroid to Earth such that its magnitude $\sqrt{x^2 + y^2} = g$.
- The change in asteroid's velocity can be approximated as $\vec{G} \times t$, where the t is the amount of time the gravitational force $\vec{G}$ acts on the asteroid. The change in the asteroid's position can be approximated as $\vec{V} \times t$, where t is the amount of time the asteroid travels with velocity $\vec{V}$.

To make matters clearer, the science advisor gives you the following example. The asteroid is at position (0,0) traveling with velocity (0,1). The Earth is at position (5,0) with a mass of 50. You now begin your calculations. The asteroid and the Earth are separated by the distance vector $\vec{D} = (5, 0)$, resulting in a distance $\sqrt{(5-0)^2 + (0-0)^2} = 5$. The gravitational attraction $g = 50/5^2 = 2$. The gravitation force $\vec{G} = (x, y)$ must point from the asteroid to the earth (i.e., match $\vec{D}$) and have a magnitude $\sqrt{x^2 + y^2} = g$, so it is calculated to be (2,0).

Now suppose we decide to simulate the position of the asteroid after 0.2 time units. The change in velocity for the asteroid is $(2, 0) \times 0.2 = (0.4, 0)$, resulting in the new velocity $(0, 1) + (0.4, 0) = (0.4, 1)$. The change in position for the asteroid (if we use the average of the old and new velocities) is $(0.2, 1) \times 0.2 = (0.04, 0.2)$, resulting in the new position $(0, 0) + (0.04, 0.2) = (0.04, 0.2)$. To continue tracking the asteroid, you simply need to repeat the process.

With the refresher course, you feel confident you can calculate the path of the asteroid as it approaches the earth. If at any point of your calculations you find the distance between the asteroid and Earth is less than or equal to 1, you can output the position and stop, since a collision will occur and further computation is pointless!

Unfortunately, just as you are about to begin, a serious problem arises. Power outages caused by gravitational disturbances from the approaching asteroid have severely limited the computing power you have available. As a result, you can only make a small number of position computations.

The process of tracking the asteroid must thus follow this sequence of steps:

1. calculate distance and force between asteroid and Earth
2. choose simulated time interval for next calculation
3. calculate new position and velocity at end of time interval
4. repeat steps 1–3 (together comprising one position calculation) until one of
   (a) collision (distance ≤ 1)
   (b) end of simulation
   (c) limit on number of calculations reached

You now need to decide how to space out your calculations over the entire span of the simulation to achieve the most precise result. An acceptable approach is to simply spread out your calculations evenly by making the interval between calculations the total simulation time period divided by the number of calculations allowed, but other heuristics may yield more precise answers. No extra position calculations are allowed, even if you do not output their results. Your program will be examined to enforce this restriction.

Remember that the precision of your answer is vital. The world is waiting for your results. Good luck!

Input format

The input consists of four lines. The first line consists of a pair of reals (x, y) marking the position of the asteroid. The second line consists of a pair of reals (x, y) marking the current velocity of the asteroid. The third line consists of an ordered triple of reals (m, x, y) giving m, Earth's mass, and x, y, Earth's location. The fourth line consists of pair of real and integer (t, c), where t is the time period you need to cover in your simulation, and c is the number of times you are allowed to calculate the position and velocity of the asteroid. Any position coordinate x, y may be negative, but all other values are positive.

Output format

The output consists of multiple lines in the same format. On each line, output three reals (t, x, y) presenting the position x, y you computed for the asteroid at time t. The numbers must be output up to three decimal places. If the position indicates a collision has occurred (distance ≤ 1), you may stop. Otherwise you may output the position calculated up to c calculations.

Example

Input:

```
0 0
0 1
50 5 0
3 15
```

Output:

```
0.200   0.040   0.200
0.400   0.161   0.398
0.600   0.364   0.592
0.800   0.655   0.776
1.000   1.042   0.945
1.200   1.538   1.091
1.400   2.166   1.200
1.600   2.963   1.245
1.800   4.006   1.158
2.000   5.480   0.653
```