Contents

1 Secret Message Pad

Hearing stories from the Oracle that a programmer named Neo may possibly be "The One", Trinity risks going into the Matrix to observe Neo. While in the Matrix, her communications are intercepted and Trinity is almost caught by Agent Smith and two other agents.

Safely back in the hovercraft Nebuchadnezzar, Trinity decides to add another level of encryption to her communication using a "one-time pad". The idea is that given a sequence of numbers (a pad), any message can be proven to be securely encrypted if each character in the string is encoded by a number from the pad, *if* each number from the pad is only used once.

Trinity decides to use the one-time pad to encrypt her messages by shifting each letter in the message by k positions later in the alphabet, where k is determined by the one-time pad. Shifts occur in alphabetical order, between the letters "a" and "z". If a letter is shifted past "z", it starts back at "a" and continues shifting. For example, shifting each letter by $k = 2$, the word "car" is transformed into "ect", while the word "yaz" is shifted to "acb".

Input Format

The input will begin with the size of the one-time pad, followed by a sequence of numbers for the pad. The remaining input consists of a series of words to be encrypted using the keyword. The input will be terminated by a line containing only -1.

You may assume that the maximum size of the pad is 100 numbers, all numbers in the pad are between 0 and 25, and that all input will be lowercase letters.

Output Format

For each word to be encrypted, output a line containing the encrypted word.

Example

Input 1:

```
10
 1  2  3  4  5  4  3  2  1  0
aaaaa zzzzz
-1
```

Output 1:

```
bcdef
dcbaz
```

Input 2:

```
40
 1  5  2 21  3  8  4 25 11  9
 6  7  8  9 11 12  3  0 11  4
14 21  9  0  1  3 12  7  2 11
 5  9 20 12  1 19  4  9  8 24
humans make good batteries
gnlr esrf
-1
```

Output 2:

```
izovqa
qzvn
mvwm
mmwtpvwzb
good
luck
```

2 Neo's Leap

On board the hovercraft Nebuchadnezzar, Neo is practicing his skills for the Matrix. Tank loads Neo into the Jump program, where he must jump from the the top of one building to the next. In the Matrix, by concentrating hard Neo can affect gravity to slow the change in the rate of his fall to 5 meters/second. As long as Neo does not land, he will continue to move forward at the same speed he was running before his jump.

Tank can approximate Neo's jump by calculating his speed and position every second. For example, if Neo jumps up with a speed of 10 meters/second while running at 30 meters/second, the results are:

Time	Distance	Height	Speed
0 seconds	0 meters	0 meters	up 10 meters/sec
1 seconds	30 meters	10 meters	up 5 meters/sec
2 seconds	60 meters	15 meters	up 0 meters/sec
3 seconds	90 meters	15 meters	down 5 meters/sec
4 seconds	120 meters	10 meters	down 10 meters/sec
5 seconds	150 meters	0 meters	down 15 meters/sec

As a result, Neo will jump 150 meters and stay in the air for 5 seconds.

Tank starts testing Neo with jumps to buildings that are at different distances away. For simplicity, Tank will calculate Neo's position at the end of every second, even though more accurate calculations are possible by if shorter intervals are used.

Input Format

The input consists of a list of three numbers representing a jump to a target building:

1. the speed Neo is running (in meters/second),
2. the speed Neo jumps up into the air (in meters/second),
3. the distance to the target building (in meters).

Output Format

Print the distance and height of Neo's jump after each second as a pair of numbers on a new line, starting at one second. Repeat each second until Neo lands (height reaches zero or below). Use integers for calculating results. For each building, output "yes" if Neo is within 3 meters of the building when he lands (height $\leq$ zero), else print "no".

Example

Input 1:

```
30 10 150
```

Output 1:

```
30 10
60 15
90 15
120 10
150 0
yes
```

Input 2:

```
10 12 65
```

Output 2:

```
10 12
20 19
30 21
40 18
50 10
60 -3
no
```

3 Cipher's Worms

Cipher is hacking into the matrix and modifying it in small ways that make the crew's job easier. Part of his job is generating trees, bushes, worms, and other critters that look real and that can be used to hide stuff for the crew. We'll focus on worms here. To keep things simple, our worms are represented as character strings (with each character representing some combination of color, texture, etc.). For example, if we use `a` and `b` to denote azure and black then `aaabbbaaabbbaaa` is a worm with azure and black bands. Now, realistic critters tend to have patterns that are more complex than this simple example. Fortunately, Cipher knows of a simple method for creating realistic and complex worms.

For each worm type, Cipher has a set of *rules* that determine how a worm grows. For example, the rule `a -> bab` indicates that the string `a` is replaced with `bab`, while `bb -> a` indicates that `bb` is replaced with `a`. Given the current state of any worm (e.g., `bbabb`), the next state is computed by replacing the left-hand-side (LHS) of each rule with its right-hand-side (RHS) at all positions in the worm where the LHS occurs. The replacements are *not* done sequentially (finding one match, performing one replacement, then finding the next match, etc.). Rather, they are done in parallel: All matches to rule LHSs are replaced in unision with the corresponding RHSs. For example, `bbabb` has three matches in all: The first rule matches the `a` in the middle while the second rule matches the `bb` substrings at each end. Substituting the rule RHSs (`a`, `bab`, and `a`) for the LHSs in the worm (respectively, `bb`, `a`, and `bb`) results in `ababa` as the next state of the worm. The initial state of a worm is called the *seed*. Successive states are sequentially numbered *generations*, with the seed being generation 0.

Your task is to write a program that grows worms as described above. Your input is a set of rules, the seed, and the number of generations. Your output is a listing of the states of the worm for each generation from zero through the given number. The rules always match unambiguously. That is, there is never more than one rule with an LHS that matches any given position in a worm. You may assume that worms and rules use only the 52 characters a–z and A–Z, that there are no more than 10 rules, and that worms do not grow longer than 200 characters.

Input Format

The first part of the input is a set of rules, organized one to a line. Each rule is specified by its LHS and RHS, separated by one or more spaces. The end of rules is indicated by a line containing a single period ("." character). Following the rules, the input specifies the seed and the number of generations (each on a line by itself). The input may contain blank lines, which should be ignored. The two-rule example described above is specified as follows:

```
a bab
bb a
.
ab
5
```

The two rules (`a -> bab` and `bb -> a`) are followed by the seed (`ab`) and the number of generations (5). Note that the rules do not include the arrow (`->`).

Output Format

The output consists of exactly $G + 1$ lines, where G is the number of generations specified in the input. Line number x consists of the state of the worm at generation x. Thus, the first line (line 0) is

generation 0, which is the seed; the last line (line G) is the G'th generation. The output should contain nothing else (no extra spaces, no blank lines). The output required on the above input is below:

```
ab
babb
bbaba
ababbbab
babbbababbabb
bbababbabbbababababa
```

Example

In the above example, there is only one match in generation 0: The first character matches the LHS of the first rule (`a -> bab`). Replacing the LHS with the RHS (`bab`) gives us generation 1: `babb`. Now we have two matches. The `a` character matches the first rule and the string `bb` to its right matches the second rule (`bb -> a`). We perform the replacements indicated by the rules (in parallel), giving generation 2: `bbaba`. (In more detail, the first character of `babb` remains unchanged, the second matches the first rule and is replaced with `bab`, while the last two match the second rule and are replaced by `a`.) The rest of the generations are computed by continuing in this manner.

Input 1:

```
a bab
bb a
.
ab
5
```

Output 1:

```
ab
babb
bbaba
ababbbab
babbbababbabb
bbababbabbbababababa
```

4 Helicopter Crash

As one of the artificial intelligence (AI) entities implementing the Matrix, you are responsible for maintaining the appearance of physical reality in response to events in the Matrix. One day, you are on duty when human vermin somehow gain control of a helicopter while attempting to escape from your agents. Due to their limited human reflexes, they crash the helicopter into the side of your glass building. You must calculate the appropriate ripple effects resulting from the impact.

To model the surface of a building, you can use a two-dimensional array of doubles. The value of each point in the array indicates how far that part of the building surface has been stretched. You then apply an *iterative finite-difference* technique to model the ripple effects of the crash as follows.

1. begin with all points at 0.0 (rest state) except the impact point, which is set to the impact value.
2. calculate the new position of a point in the surface as the average of 1) current location and 2) average of neighbors.
3. update all points at once.

The outermost positions of the building are assumed to always remain at 0. For simplicity, you may assume your building has a height of 7 and a width of 5.

Note that it is important to store changes to the array and apply them all at once at the end of the time step. Otherwise, algorithms utilize a mixture of old and new simulation data, which can yield undesirable results.

Input Format

The input will consist of four integer numbers. The first two numbers indicating the point of impact (height, width). The third number is the impact strength; The final number is how many time steps of the simulation should be displayed.

Output Format

For each time step of the simulation, the program should print "Time $< t >$" (starting at time 0), then the state of the building at that timestep. The building state can be printed as a 7x5 array of numbers, with each floor of the building on a separate line. Even though the values are stored as doubles, for clarity the program should convert it and just print an integer values.

Example

Input:

```
2 2 100 3
```

Output:

```
Time 0
0       0       0       0       0
0       0       0       0       0
0       0       100     0       0
0       0       0       0       0
0       0       0       0       0
0       0       0       0       0
0       0       0       0       0
Time 1
0       0       0       0       0
0       6       6       6       0
0       6       50      6       0
0       6       6       6       0
0       0       0       0       0
0       0       0       0       0
0       0       0       0       0
Time 2
0       0       0       0       0
0       7       7       7       0
0       7       28      7       0
0       7       7       7       0
0       0       1       0       0
0       0       0       0       0
0       0       0       0       0
```

5 Transmit Neo

As you well know, the way to travel back and forth from the Matrix to the real world is via sophisticated communication over telephone lines. Turns out, what is transmitted is the *actual DNA sequences* of the folk who are going back and forth from the Matrix. Clearly, such communication must be error free, otherwise, genes may be modified in transit which can, at the very least, be messy (rumor has it an early transmission error literally caused Morpheus to lose his hair!).

In any case, nowadays these highly sensitive messages are protected using *error detecting and correcting codes*. The messages themselves are DNA sequences, and thus consist only of the letters **A, C, G,** and **T**. When these letters are transmitted over the phone lines, they can be arbitrarily modified because of noise on the circuit. However, each modified letter still is a letter from the original alphabet, e.g. if you transmitted **A**, and it was corrupted in transit, you would get some other letter **C, G,** or **T**.

The error detecting and correcting code takes into account the fact that the telephone lines in and out of the matrix can corrupt at most two letters in any nine letter sequence. This code can detect any error as long as less than three letters in the message are modified in transmission. It can even *correct* the error if only one letter is modified in transit. Each message consists of four data letters from a DNA sequence and five other letters (constructed from four data letters as explained below).

The nine letters in a message are arranged as a 3×3 matrix. The top-left 2×2 matrix of this 3×3 matrix are the four data words. The rest are the check codes. The check codes are computed using the $\odot$ binary function shown in Table 1. The individual entries in 3×3 matrix are computed as shown in Table 2. The $d_{x,y}$ entires are the data letters. The check codes are computed using the letters as shown in the message. You should try to figure out what (and *why*) the entry marked $\times$ in the bottom right of the message should be.

$\odot$	A	C	G	T
A	A	C	G	T
C	C	A	T	G
G	G	T	A	C
T	T	G	C	A

Table 1: The $\odot$ function

$d_{0,0}$	$d_{0,1}$	$d_{0,0} \odot d_{0,1}$
$d_{1,0}$	$d_{1,1}$	$d_{1,0} \odot d_{1,1}$
$d_{0,0} \odot d_{1,0}$	$d_{0,1} \odot d_{1,1}$	$\times$

Table 2: Message format

Example

Consider the example message in Table 3. Here $d_{0,0} =$ **T**, $d_{0,1} =$**G**, etc. Using Tables 1 and 2, you should be able to check that all five check code entries in this message are correct.

Now suppose the message is corrupted during transmission and (say) the first letter (**T**) is changed to an **A**. In this case, the received message is shown in Table 4.

Note that two check codes in the first column and the first row , (corresponding to the incorrect letter), are invalid, because **A** $\odot$ **G** $\neq$ **C** and **A** $\odot$ **C** $\neq$ **G**. Given the check codes and the corrupt message, can you reconstruct the original letter?

In our last example (shown in Table 5), we consider the case when two letters of message 3 are corrupted in transit.

T	G	C
C	T	G
G	C	T

Table 3: Example Message (no errors)

A	G	C
C	T	G
G	C	T

Table 4: Message with one error

In this case, the first two data letters **T** and **G** have been corrupted to **A** and **C**. Consider the first and second columns of the transmission — since $\mathbf{A} \odot \mathbf{C} \neq \mathbf{G}$ and $\mathbf{C} \odot \mathbf{T} \neq \mathbf{C}$ we know there were errors in the transmission. However, we can only detect this error and not correct it. (Why?) (The problem is that both the original message and the message in Table 6 can be *re-constructed* by changing two letters from the message in Table 5, and there is no way of telling which one is the original message!)

Program

The input to your program will be a set of messages. Each line will contain three characters separated by a single space each and there will be a newline between each different message. Your program should read a message (three lines) and output whether the message had any errors. If only one error is detected in the data part, it should point out where the error is can correct it (and output a new correct message). If more than one error is detected, the program should just say that errors were detected (and not try to correct them). If a single error is detected in the check codes, the program should note this. The expected output of the program is shown on a few sample messages (using some of the messages we have already talked about).

A	C	C
C	T	G
G	C	T

Table 5: Message with two errors

A	C	C
G	A	G
G	C	T

Table 6: Another valid message that can be constructed by changing two letters in Table 5.

Input:

```
T G C
C T G
G C T

A G C
C T G
G C T

A C C
C T G
G C T

A C C
G A G
G C T

A C C
G A G
A C T
```

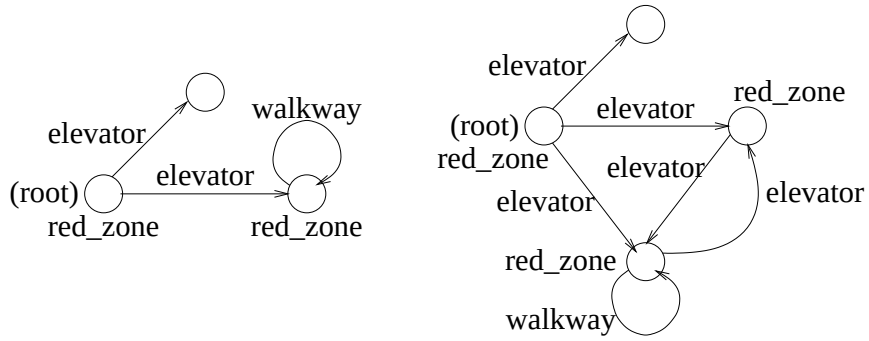
Output:

```
Read:
T G C
C T G
G C T
Data transmission fine...

Read:
A G C
C T G
G C T
Error in pos: (0,0) ... corrected, new code:
T G C
C T G
G C T

Read:
A C C
C T G
G C T
Can't correct more than one error

Read:
A C C
G A G
G C T
Data transmission fine...

Read:
A C C
G A G
A C T
Check code error at position (2, 0)
```

6 A Glitch in the Matrix

Cipher has succeeded in hacking into the matrix and has introduced several friendly critters, such as the worms in Question 3. Now, he is getting more ambitious and wants to modify the topology of the matrix. This task is delicate, since agents patrol the matrix looking for glitches caused by imperfectly done modifications.

Cipher knows that the essential topology of the matrix is represented using a simple directed graph. Nodes in this graph represent locations (e.g., a building basement) and have optional labels (e.g., "basement132"). The labeled edges in this graph represent connections between locations. For example, an edge with label "elevator2" from a node (with label) "basement132" to node "lobby" indicates that the elevator leads from the basement to the lobby. The graph has a special node, called the *root*, that serves as the main entry point into this world. The following figure suggests two such graphs. Note that all edges are labeled, but some nodes are not.



Cipher also knows a little about how the patrolling agents operate: Since determining whether two graphs are identical is an expensive task, agents use a simpler method based on only local comparisons. More precisely, Cipher has determined that agents will be unable to detect a glitch in the matrix if there exists a relation (say, R) between the nodes of the old and new graphs (G_1 and G_2) with the following properties. (If node x in G_1 is related to node y in G_2, we write xRy.)

1. The roots of G_1 and G_2 are related to each other and to no other nodes.
2. If xRy then x and y must either have the same label or must both be unlabeled. (Recall that node labels are optional.)
3. If xRy and (x, l, x') is an edge in G_1, then there must be an edge (y, l, y') in G_2 such that $x'Ry'$. Similarly, if xRy and (y, l, y') is an edge in G_2, then there must be an edge (x, l, x') in G_1 such that $x'Ry'$.

Cipher has a number of graphs he would like to use interchangeably. For a pair of such graphs, if we can find some relation R with the above properties, the agents will be unable to tell them apart (and thus will be unable to detect a glitch). Your task is to write a program that determines when such a relation R exists between two given graphs. We will only consider graphs in which each non-root node is reachable from the root by following one or more directed edges. You may assume that the graphs will have no more than 64 nodes each and that there are no more than 16 edges out of any node. Node and edge labels contain only the characters a–z, A–Z, and underscore ("_") and are at most 31 characters long.

Input Format

For the purpose of describing an input graph, we will number the nodes of the graph sequentially $0, 1, 2, \ldots, N-1$, where N is the number of nodes in the graph. We will use the convention that the root of the graph is always numbered 0. (Note that these numbers are for the purpose of data input only; they have no significance for identifying a relation R as described above.)

The input describes two graphs, one after the other. For each graph, the input first lists the edges, one per line, followed by a line containing only "." (the period character), followed by a listing of node labels, one per line, followed by another line containing only a period. An edge is described by listing its label, source node identifier, and target node identifier (separated by white space). A node label is described by listing the label followed by the node identifier (separated by white space). Node labels are optional; therefore, some or all of the nodes may have no labels specified. Blank lines in the input should be ignored. For example, the two graphs depicted above would be input as follows:

```
elevator 0 1
elevator 0 2
walkway 1 1
.
red_zone 0
red_zone 1
.
elevator 0 1
elevator 0 2
elevator 0 3
walkway 1 2
walkway 2 2
walkway 2 1
.
red_zone 0
red_zone 1
red_zone 2
.
```

Output Format

If it is possible to find a relation R as described above (and thus fool the agents), your program should output the string "`No glitch`"; otherwise, it should output the string "`Glitch`" to warn Cipher that the agents may detect the glitch. For the two graphs in our example, the relation R given by $\{(0,0),(1,1),(1,2),(2,3)\}$ satisfies the requirements (items 1–3 above) and therefore the output indicates no glitch:

```
No glitch
```

Note that the task of identifying the required relation R in this example was simplified a by the presence of the node labels and the small number of nodes. However, since node labels are optional your program cannot rely solely on labels for relating nodes. Further, when the number of nodes in the input graphs is large, a strategy based on trying out all possible relations between nodes is likely to run out of time before it finds an answer.

7 Find Neo

Neo is inside the Matrix and he needs to urgently get back to the ship. Tank must find Neo inside the Matrix using messages that Neo is sending. However, Neo can't just say where he is — otherwise, the agents will get to him first! Instead, Neo uses an ingenious scheme that Morpheus has devised to help Tank find him.

Instead of sending one message, Neo sends a set of messages over the Matrix's communication network. Each router has an unique address and Tank has to figure out which router Neo is closest to.

If we consider the comm. network to be a graph, then the routers in the network are vertices and the links between the routers are edges, and Tank's job is to figure out the id (address) of the Neo's vertex. Each edge in the graph has a weight. As messages are transmitted over these edges, the weight of the link is *multiplied* to a product carried with the message. Neo always sets the initial value of the product to 1, and as messages traverse through the network, the number carried in the messages is the product of the edge weights of all the edges a message has been forwarded on.

The messages Neo sends are routed randomly throughout the network, and travel arbitrary distances. However, each message can only travel over a single link once (and can only visit a given vertex once). The network topology is already known to Tank. For each message that Neo sends, Tank receives the id of the vertex that the message *ends* at (i.e. the node from which the message was not forwarded any more) and the product of the weights of edges the message took to get there. Given sufficient number of messages, using just this information, Tank can track Neo and get him back safely.

Wait a minute...

You are probably already worried that a really large number of messages must be needed in order for this technique to work. Further, it is not clear how any number of messages would be sufficient for arbitrary topologies. Morpheus was stumped with these problems as well, when Trinity pointed out two very important properties of the Matrix communication graph:

- The communication graph is not an arbitrary graph, but has a lot of structure. Specifically, it is a set of same sized cliques arranged as a tree. (Recall that a clique is a graph in which every vertex has an edge to every other vertex, and a tree is a connected graph without any cycles).
- Lastly (and extremely important), the edges inside the clique all have weights chosen from the set $\{2, 3, 5, 7\}$, and the edges between the cliques are chosen from the set $\{11, 13, 17, 19, 23, 29\}$.

An example graph of this type is shown in Figure 1. In this case, the cliques all have four vertices (and correspondingly six edges). There are five of these cliques and so the total number of edges in the graph is 34 (= 6x5 + 4).

Consider an example in which Neo is on vertex 16 and sends the a message that travels five hops — vertices 14, 5, 6, 4 and ends up at node 7. In this case, the information Tank gets is (node 7, product 12350).

Similarly, consider another Neo sends (again starting from vertex 16) that goes through nodes 14, 12, 13, and ends up at node 15. In this case, Tank gets the pair (node 15, product 532).

In both these cases, the only possible node that could have sent that message with those products happen to be node 16. However, consider a message that starts at node 16, and traverses nodes 14, 5, 7, 4, 6, 9, 10, 8, and finally ends up at node 11. In this case, the information Tank gets is (node 11, product 15042300). Tracing back, it turns out this message could have been originated from any of the nodes 16, 17, 18, or 19! So in general, a single message points back to a *set* of possible originating nodes, and in order to figure out exactly where Neo is, Tank has to keep intersecting the possible originating node sets till only one remains.

Clearly, writing an intersection algorithm is no problem to an experienced programmer like Tank, but he needs your help in figuring out the originating set.

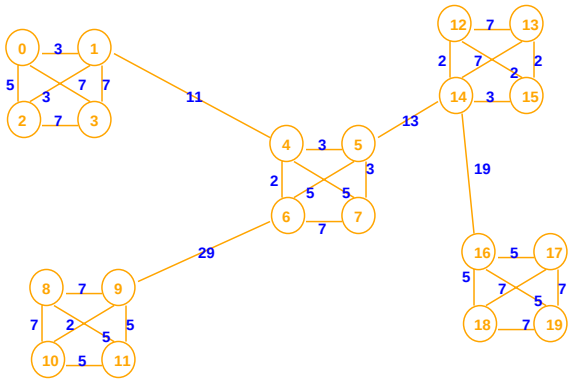


Figure 1: Example Graph

A further detail Morpheus noticed a while back that for long paths, the product grows large quickly and can easily overflow the 32-bit registers in the ship's computers. Therefore, when the routers compute the product, they send back four integers. These integers contain unique prime factors of the real product. Specifically, the first integer is used if (and only if) an edge with weight 29, 5, or 3 is crossed. The second integer is used only for edges with weights 23, 7, and 2. The third integer is used for the weights 17 and 13, while the last integer is used for the weights 11 and 19.

Thus, in our first three examples, the input product integers were 25 2 13 19 ($25 \times 2 \times 13 \times 19 = 12350$), 1 28 1 19 ($28 \times 19 = 532$), and 2175 28 13 19 ($2175 \times 28 \times 13 \times 19 = 15042300$).

Program description

The input to your program is a graph description file. The output should be the set of vertices reachable from vertex i using exactly and only the product integers (in any order).

The graph description file has the following format:

```
#clique size #num cliques
4 5
#Graph
20 34
#Edge Set
0 3 7
0 2 5
0 1 3

...

5 14 13
14 16 19

#Paths
15 1 28 1 19
7 25 2 13 19
11 2175 28 13 19
0 75 49 13 209
2 9 49  13 209
```

The first line is a comment. The second line says what the clique size is and how many cliques are there in the graph. The third line is a comment. The fourth line lists the number of vertices and the number of edges. The fifth line is again a comment. The subsequent lines are either blank or is an edge description. An edge is defined by two vertices (specified by the first two integers) and a weight. The entire graph description file for the graph in Figure 1 is given at the end of this description.

At the end of the list of edges is another comment, following which are a list each containing five integers. These integers are the terminating vertex id and the four product integers. The output of your program should be the set of vertices reachable from each terminating vertex using the product integers.

Example Output

```
Terminating vertex 15, product set [1, 28, 1, 19]
Found a path from    15 ->    16

Terminating vertex 7, product set [25, 2, 13, 19]
Found a path from     7 ->    16

Terminating vertex 11, product set [2175, 28, 13, 19]
Found a path from    11 ->    16
Found a path from    11 ->    17
Found a path from    11 ->    18
Found a path from    11 ->    19

Terminating vertex 0, product set [75, 49, 13, 209]
Found a path from     0 ->    16
Found a path from     0 ->    17
Found a path from     0 ->    18
Found a path from     0 ->    19

Terminating vertex 2, product set [9, 49, 13, 209]
Found a path from     2 ->    16
```

Here are another set of example runs assuming Neo is sending messages from node 7.

```
Terminating vertex 15, product set [3, 28, 13, 1]
Found a path from    15 ->     4
Found a path from    15 ->     6
Found a path from    15 ->     7

Terminating vertex 3, product set [75, 7, 1, 11]
Found a path from     3 ->     6
Found a path from     3 ->     7

Terminating vertex 2, product set [45, 98, 1, 11]
Found a path from     2 ->     7
```

We will test your code on graphs somewhat larger than this toy example. Our test graphs will have cliques of size up to 10 and graphs with no more than 10,000 vertices. We have provided you a test graph with clique size 6 and 128 cliques.

Complete Graph Description File

```
#clique size #num cliques
4 5
#Graph
20 34
#Edge Set
0 3 7
0 2 5
0 1 3
1 3 7
1 2 3
2 3 7

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

8 11 5
8 10 7
8 9 7
9 11 5
9 10 2
10 11 5

12 15 2
12 14 2
12 13 7
13 15 2
13 14 7
14 15 3

16 19 5
16 18 5
16 17 5
17 19 7
17 18 7
18 19 7

1 4 11
6 9 29
5 14 13
14 16 19

#Paths
15 1 28 1 19
7 25 2 13 19
11 2175 28 13 19
0 75 49 13 209
2 9 49  13 209
```

8 Sentinel Detection

After rescuing Neo, Morpheus is piloting the hovercraft Nebuchadnezzar back to Zion, the last human city. To recapture them, the AI entities controlling the Matrix have sent out several robot *sentinels*, giant octopus-shaped robots which can chase down and tear apart the hovercraft. Nebuchadnezzar's sensors have located several sentinels sentinels around the ship. Morpheus needs to calculate the range of their detection systems in order to define safe zones for travel.

You are to help Morpheus by writing the code to accurately detect the proximity of the sentinels and appropriately sound the warning alarms. The latest generation sentinels that the Matrix has designed can be approximated by regular hexagons (each side of the hexagon is equal). Your ship can be approximated by a circle. Write the code to detect the closest sentinel to the ship. You can assume that all the hexagons are oriented alike with one pair of edges parallel to the x-axis. The distance has to be reported from the ship's outer hull to the outer boundary of the sentinel (not from ship's center to the sentinel's center). If the distance to the closest sentinel is less than zero (i.e. the sentinel has already breached the ship's hull), the distance should be reported as zero (not a negative number).

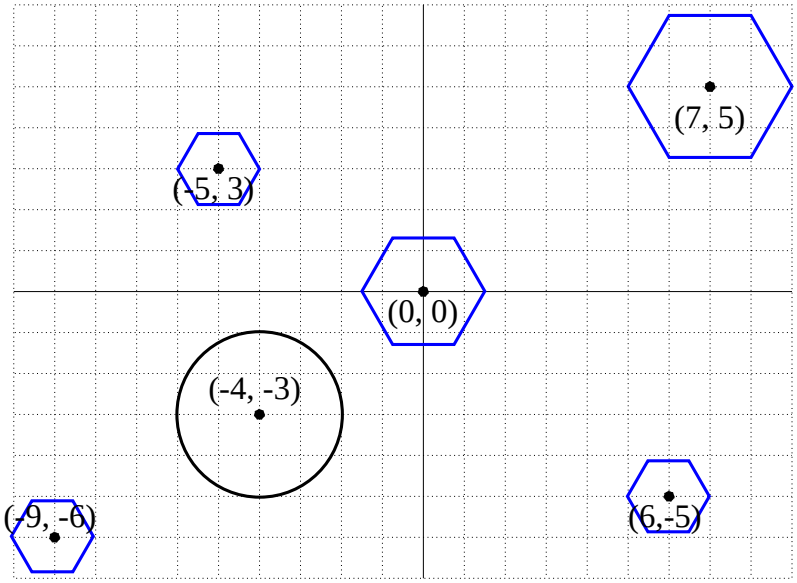


Figure 2: Example Scenario

Input Format

The first line contains the number of lines you need to further read. This is an integer. All the other values should be read as doubles. The second line contains the center of the ship's position and radius:
$ship_x ship_y ship_{radius}$
The third and subsequent lines contain the centers and radii of the sentinels:
$sentinel_{i_x} sentinel_{i_y} sentinel_{i_{radius}}$

Output Format

List the distance of the closest sentinel to 2 decimal places :

```
The distance of the closest sentinel is 21.91
```

Example

Input:

```
6
-4 -3 2
-9 -6 1
-5 3 0.5
0 0 1.5
6 -5 1
7 5 2
```

Output:

```
The distance of the closest sentinel is 1.67
```