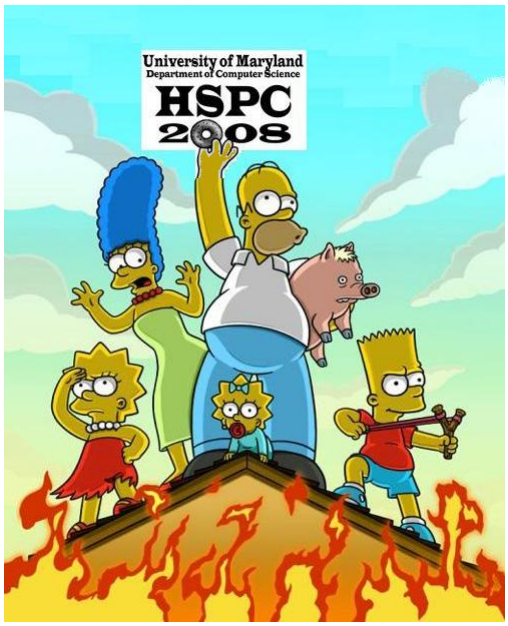
# Contents





# 1 Bart's Skateboard Park

Bart is skateboarding through the town of Springfield and wants to find the 3 block section of town with the most jumps to designate as his own skateboard park.

### Input/Output Format:

Input is a list of block descriptions. On each line is the the block number, followed by the number of jumps on that block. The block numbers start at 1. If descriptions are provided for $n$ blocks, you may assume they are for blocks 1 through $n$. You may assuming $n \geq 3$. Note the block numbers are not necessarily sorted.

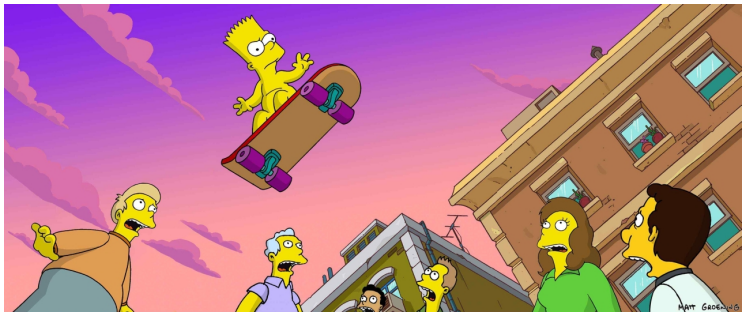
Output is the 3 consecutive blocks with the most jumps (with blocks listed sorted order according to block number). You may assume there are no ties.

### Examples:

| Example 1 Input: | Example 1 Output: |
|---|---|
| 1 2<br>2 6<br>3 3<br>4 4<br>5 1 | 2 3 4 |

| Example 2 Input: | Example 2 Output: |
|---|---|
| 4 4<br>2 6<br>5 1<br>1 2<br>3 3 | 2 3 4 |





## 2 Simpsons' Hidden Talents

**Homer:** Marge, I just figured out a way to discover some of the talents we weren't aware we had.
**Marge:** Yeah, what is it?
**Homer:** Take me for example. I want to find out if I have a talent in politics, OK?
**Marge:** OK.
**Homer:** So I take some politician's name, say Clinton, and try to find the length of the longest prefix in Clinton's name that is a suffix in my name. That's how close I am to being a politician like Clinton
**Marge:** Why on earth choose the longest prefix that is a suffix???
**Homer:** Well, our talents are deeply hidden within ourselves, Marge.
**Marge:** So how close are you?
**Homer:** 0!
**Marge:** I'm not surprised.
**Homer:** But you know, you must have some real math talent hidden deep in you.
**Marge:** How come?
**Homer:** Riemann and Marjorie gives 3!!!
**Marge:** Who the heck is Riemann?
**Homer:** Never mind.

Write a program that, when given strings $s1$ and $s2$, finds the longest prefix of $s1$ that is a suffix of $s2$.

### Input/Output Format:

Input consists of two lines. The first line contains $s1$ and the second line contains $s2$. You may assume all letters are in lowercase.

Output consists of a single line that contains the longest string that is a prefix of $s1$ and a suffix of $s2$, followed by the length of that prefix. If the longest such string is the empty string, then the output should be 0.

The lengths of $s1$ and $s2$ will be at most 50000. Your program should finish in less than one minute.

### Examples:

| Example 1 Input: | Example 1 Output: |
|---|---|
| `clinton`<br>`homer` | `0` |

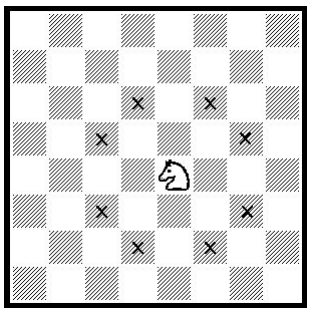
| Example 2 Input: | Example 2 Output: |
|---|---|
| `riemann`<br>`marjorie` | `rie 3` |

## 3 A Knight and a Queen

Marge walks in the house and finds Homer staring at a chessboard with a knight and a queen on it.

**Marge:** Homer, are you OK? What's up with these intellectual endeavors of yours recently?
**Homer:** Oh, I'm OK. And I'm not playing chess anyway.
**Marge:** So what are you doing?
**Homer:** I'm imagining that I'm a knight.
**Marge:** Yeah right.
**Homer:** Well, a knight sitting on the chessboard. And I imagine you are my queen.
**Marge:** I like that.
**Homer:** Of course you are sitting on the chessboard too. Now I wonder if I can reach the square you're sitting on in 16 or fewer moves.
**Marge:** So can you?
**Homer:** I'm still trying to figure it out.

Write a program that, when given a knight and a queen on an $n$ by $n$ chessboard, finds if the knight can reach the queen in $m$ or fewer than moves. One "move" for a knight is defined as 2 squares in one direction, then one square in a perpendicular direction. Knights cannot move along diagonals. For example, if $n = 8$, $kx = 3$, and $ky = 5$, then possible positions for the knight after one move given in ($kx$, $ky$) are: (1,6), (1,4), (2,7), (2,3), (4,7), (4,3), (5,6), and (5,4).



### Input/Output Format:

Input consists of 3 lines. The first line contains 2 numbers: $n$ and $m$. $n$ is the dimension of the board, and $m$ is the number of moves the knight is allowed to do.

The second line contains two numbers: $kx$ and $ky$, s.t. $1 \leq kx \leq n$, $1 \leq ky \leq n$. These two numbers indicate the position of the knight on the board.

The last line again contains two numbers: $qx$ and $qy$, s.t. $1 \leq qx \leq n$, $1 \leq qy \leq n$. They indicate the position of the queen on the board.

If the queen is reachable by the knight with $m$ or less moves, output should contain the following string on a single line ending with a newline:

**Knight can reach Queen within $m$ moves!**

If the queen is not reachable by the knight with $m$ or less moves, output should contain the following string on a single line ending with a newline:

**Knight cannot reach Queen within $m$ moves!**

In the above, $m$ should be the actual value from the input.

No test case will have the queen and the knight sitting on the same square. You can further assume that $n \leq 1000000, m \leq 256$. Your program should finish in less than one minute.

## Examples:

| Example 1 Input: | Example 1 Output: |
|---|---|
| `8 2`<br>`3 5`<br>`7 4` | `Knight cannot reach Queen within 2 moves!` |

| Example 2 Input: | Example 2 Output: |
|---|---|
| `8 3`<br>`3 5`<br>`7 4` | `Knight can reach Queen within 3 moves!` |

# 4 Sorted Trail Map

Volunteers from the World Wildlife Foundation (WWF) have arrived in Springfield in a WWF truck. They are collecting endangered animals living in the nearby Gump Forest, and are asking Lisa for help.

The WWF volunteers have given Lisa a map of the Gump Forest. The forest contains a number of clearings, and trails connecting the clearings. Different types of endangered animals live on each trail, no animals live in the clearings.

The WWF are hoping to collect some endangered animals from Gump Forest. But their trail map is too confusing, since clearings and animals are listed in no particular order. Lisa decides to help the WWF volunteers by making up a sorted trail map of Gump Forest.

### Input/Output Format:

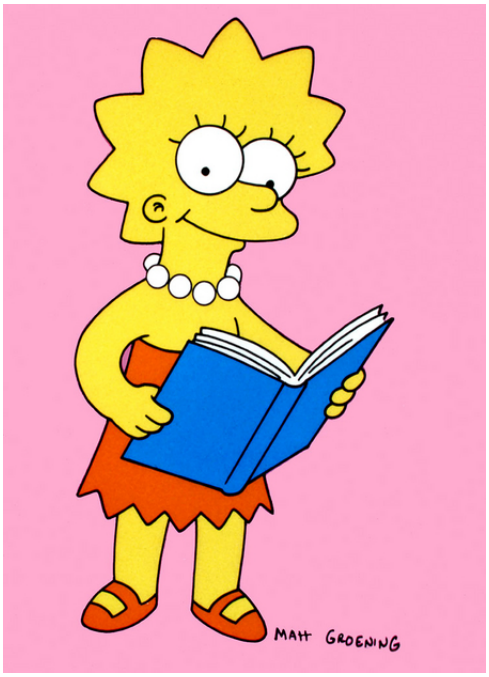
Animals are given as words. Clearings are given as numbers. There may be up to 500 clearings. Clearing 0 is always the entrance to Gump Forest. The input is the unsorted Gump Forest trail map. Each line describes a trail between two clearings as a pair of numbers, followed by a list of animals living on the trail.

The program output is a sorted trail map, where clearing numbers go up from 0, and animal names go from a to z. Trails are sorted by the number of the clearings at either end of the trail. All animals living on the trail are sorted in alphabetical order.

### Examples:

| Example 1 Input: | Example 1 Output: |
|---|---|
| `0 1 puma` | `0 1 puma` |
| `2 3 toad` | `1 2 frog` |
| `1 2 frog` | `2 3 toad` |

| Example 2 Input: | Example 2 Output: |
|---|---|
| `1 0 puma lynx` | `0 1 lynx puma` |
| `2 0 puma` | `0 2 puma` |
| `1 2 vole` | `1 2 vole` |



# 5 Collecting Forest Wildlife

Volunteers from the World Wildlife Foundation (WWF) have arrived in Springfield in a WWF truck. They are collecting endangered animals living in the nearby Gump Forest, and are asking Lisa for help.

The WWF volunteers have given Lisa a map of the Gump Forest. The forest contains a number of clearings, and trails connecting the clearings. Different types of endangered animals live on each trail, no animals live in the clearings. **Every trail must have at least one animal. Two trails connected to the same clearing cannot have the same animal.**

Trails in the Gump Forest are narrow, so the WWF truck can only turn once it reaches the end of the trail and reaches a clearing. Exactly one endangered animal **must** be collected each time the truck drives along a trail. A trail may be traveled many times to pick up multiple animals.

The WWF truck is filled with a number of cages, where each cage is designed for a single endangered species. Unfortunately, the truck is so filled with cages that the cages can only be filled in a certain order. Lisa has been asked to go with the WWF volunteers in their truck and guide the truck through the Gump Forest and fill all the cages in the truck. The truck starts at the clearing at the entrance, goes through the forest, and returns to the entrance.

## Input/Output Format:

Animals are given as words. Clearings are given as numbers. There may be up to 500 clearings. Clearing 0 is always the entrance to Gump Forest.

The input begins with the Gump Forest trail map. Each line describes a trail between two clearings as a pair of numbers, followed by a list of animals living on the trail. Remaining lines begain with list of animals to be collected, in the order they are to be collected.

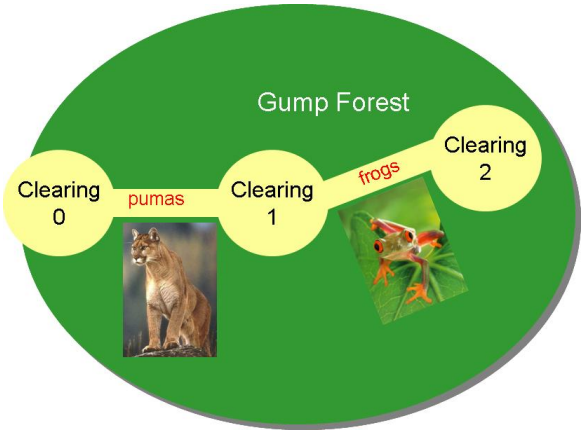
For each list of animals to be collected, your program should output the successful path through the Gump Forest as a sequence of numbers in the order clearings are visited, starting and ending with 0. If multiple successful paths are possible, print any successful path. If there is no successful path through the Gump Forest that can fill up the truck and return it to the entrance, your program should output "Failed" for that list.

**Hint:** Don't try to solve the problem in a brute-force manner by enumerating all possible paths through Gump Forest. (It will take too long.)

## Examples:

| Example 1 Input: | Example 1 Output: |
|---|---|
| `0 1 puma`<br>`1 2 frog`<br>`puma puma`<br>`puma frog frog puma` | `0 1 0`<br>`0 1 2 1 0` |

| Example 2 Input: | Example 2 Output: |
|---|---|
| `1 0 puma lynx`<br>`1 2 vole`<br>`2 0 toad`<br>`toad vole lynx puma lynx`<br>`toad puma lynx` | `0 2 1 0 1 0`<br>`Failed` |



# 6 Crowded Forest Wildlife

The WWF volunteers are back again to collect more wildlife from Gump Forest! Animals have moved around the forest, and some trails are getting more crowded!. **Lisa finds that the same type of animal may now be found living on multiple trails connected to the same clearing. Elsewhere in the forest trails may have no animals living on it.** If a trail has no animals, the WWF truck can drive along it any time without collecting any animals. Can Lisa still guide the WWF volunteers through the forest and fill up their truck?

## Input/Output Format:

The same as before. Animals are given as words. Clearings are given as numbers. There may be up to 500 clearings. Clearing 0 is always the entrance to Gump Forest.

The input begins with the Gump Forest trail map. Each line describes a trail between two clearings as a pair of numbers, followed by a list of animals living on the trail. Remaining lines began with list of animals to be collected, in the order they are to be collected.

For each list of animals to be collected, your program should output either "Succeeded" if there is a path through the Gump Forest that can fill up the truck and return it to the entrance. Otherwise your program should output "Failed" for that list.

**Hint:** Don't try to solve the problem in a brute-force manner by enumerating all possible paths through Gump Forest. (It will take too long.)

## Examples:

| Example 1 Input: | Example 1 Output: |
|---|---|
| `1 3 puma`<br>`2 3 lynx`<br>`0 3 toad`<br>`0 1`<br>`0 2`<br>`puma lynx`<br>`lynx toad`<br>`puma lynx toad` | `Succeeded`<br>`Succeeded`<br>`Failed` |

| Example 2 Input: | Example 2 Output: |
|---|---|
| `1 0 puma`<br>`1 2 puma`<br>`2 0 puma`<br>`puma puma puma`<br>`puma` | `Succeeded`<br>`Failed` |

# 7 Homer's Broken Remote

After a hard day at the nuclear power plant, Homer loves to enjoy an evening on the couch watching television. Unfortunately, after years of abuse, Homer's remote control unit is not fully functional. Homer wants to switch channels, but some of the digit keys on his remote control are broken. He has a working up-channel key ('`+`'), a working down-channel key ('`-`'), but some of the digits '`0`'–'`9`' may be broken. (They may all be broken, some may be broken, and none may be broken.) The remote also has a working "Enter" key ('`E`'), which must be hit after each sequence of digits. Your job is, given the state of Homer's remote, help Homer determine the minimum of key presses to get from one given channel to another.

For example, suppose that Homer only has buttons 4 and 5 working, and he wants to go from start channel 40 to target channel 50. He could hit the '`+`' key 10 times. Better would be to use the sequence "`44E++++++`", since this would only involve 9 key presses. The best sequence, however, is "`54E----`", since this would only involve 7 key presses.

The minimum channel is 1 and the maximum channel is given in the input. If you are at the maximum channel and hit '`+`', you loop around to channel 1, and similarly, if you are at channel 1 and hit '`-`', you loop around to the maximum channel. Thus, in the previous example, if the maximum channel had been 330, and you wanted to go to channel 329, you could use "`4E-----`".

### Input/Output Format:

We will provide a main program that will read the input for you. The first line of input contains the maximum channel, the number of working digit keys, and a list of the working digits. (For example, "`330 2 4 5`" means that 330 is the maximum channel, and there are 2 working digits, '4' and '5'.) You may assume that the minimum channel is always 1, and the maximum channel cannot exceed 9999.

This is followed by a sequence of channel pairs, one pair per line, consisting of the start channel $s$ and desired target channel $t$. You may assume that $s$ and $t$ will be valid channels and that $s \neq t$. To simplify the interface, our main program initializes the following global variables for you. The boolean array `isWorking` has 10 elements, where `isWorking[i]` is true if the digit $i$ is working and false otherwise.

```
static int maxChannel;              // maximum channel number
static int nWorkingDigits;          // number of working digit keys
static boolean[] isWorking;         // true if i-th digit key works
static int startChannel;            // starting channel number
static int targetChannel;           // desired target channel number
```

All you need to do is to provide the contents of the following procedure, which returns the optimal key sequence.

```
private static String generateSequence()
```

Because some of the optimal sequences may have very long runs of '`+`' or '`-`', we have provided an output procedure `makePrintable(sequence)`, which will take the sequence you provide and produce

something easier to read by replacing long strings of either '`+`' or '`-`' (having 6 or more repeats) with a repetition count. For example if your program returned the sequence "`55E+++++++`" (meaning enter channel 55 followed by 7 repetitions of '`+`'), it would be printed as "`55E[7x+]`". Note that there may be two sequences of equal length that tie for being the shortest. In such a case you may output either.

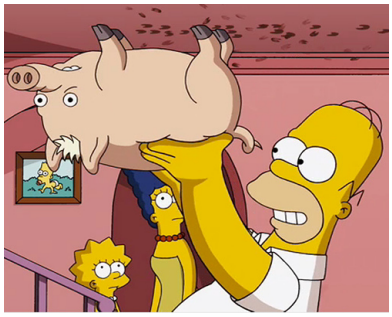
**Examples:**

| Example 1 Input: | Example 1 Output: |
|---|---|
| `330 2 4 5`<br>`40 50`<br>`40 329`<br>`40 62` | `Max Channel: 330  Working Digits: 4 5`<br>`Start: 40  Target: 50  Sequence: 54E----`<br>`Start: 40  Target: 329  Sequence: 4E-----`<br>`Start: 40  Target: 62  Sequence: 55E[7+]` |

| Example 2 Input: | Example 2 Output: |
|---|---|
| `9999 5 0 6 4 2 8`<br>`1234 6789`<br>`1234 9997`<br>`9997 1234`<br>`777 790` | `Max Channel: 9999  Working Digits: 0 2 4 6 8`<br>`Start: 1234  Target: 6789  Sequence: 6800E[11-]`<br>`Start: 1234  Target: 9997  Sequence: 2E----`<br>`Start: 9997  Target: 1234  Sequence: 888E[346+]`<br>`Start: 777  Target: 790  Sequence: [13+]` |

# 8 Spider Pig's Doughnut-Eating Spree

Homer's new pet, *Spider Pig*[1], is making a tour of the backyard, where Homer has left a number of doughnuts. Your job is to help Spider Pig eat all the doughnuts by taking the shortest possible tour, returning to his starting point. There is a restriction, however. Spider Pig starts at the leftmost doughnut (the one with the smallest $x$-coordinate) and walks strictly from left to right until he hits the rightmost doughnut, and then he goes back strictly from right to left. In other words, on the outbound portion of his trip his $x$-coordinate increases and along the return portion the trip his $x$-coordinate decreases. An example of such a doughnut path is shown in the figure below.



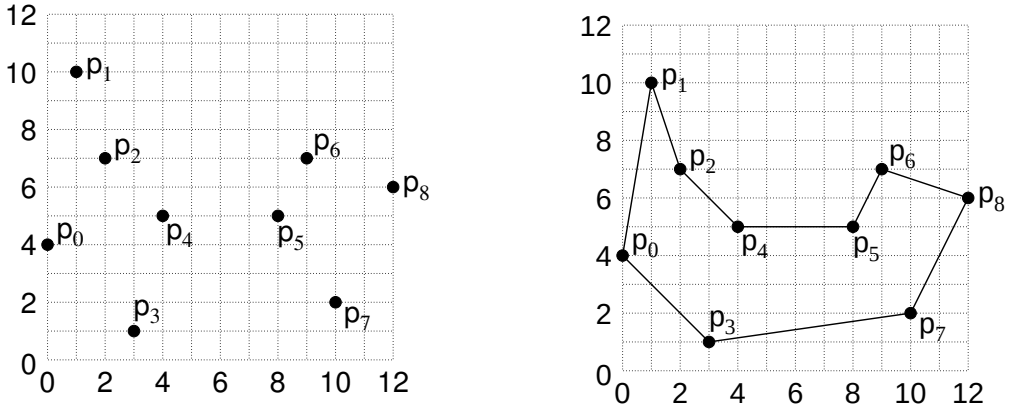



Figure 1: Sample input (right) and the optimal solution (left) for Spider Pig.

You will be provided with the coordinates of the doughnuts as input and write a procedure that will determine the path of minimum length, subject to the above requirements. To simplify distance computations, we will assume the so-called *Manhattan*, or *checkerboard*, distance function. The distance between two points $p = (p_x, p_y)$ and $q = (q_x, q_y)$ is given by the formula

$$\text{distance}(p, q) = |p_x - q_x| + |p_y - q_y|.$$

## Input/Output Format:

We have provided a main program for you that handles all the input and output. The quantities it inputs are listed below and are stored as global variables. The integer variable `n` stores the number of doughnuts. The integer arrays `dx[]` and `dy[]` store the $x$,and $y$-coordinates, respectively, of the doughnuts. You may assume that all coordinates are nonnegative integers, no two doughnuts have the same $x$-coordinates, and they are given in the array in increasing order of $x$-coordinates.

```
static int n;           // number of doughnuts
static int[] dx;        // doughnut x-coordinates
static int[] dy;        // doughnut y-coordinates
```

[1] "Does whatever a Spider Pig does."

All you need to provide is the body for the procedure `generatePath()`, which computes the minimum length path (given the above global values and subject to the left-right constraints). Note that any valid path consists of two portions, and outbound portion from doughnut 0 to $n-1$ and an inbound portion from doughnut $n-1$ back to 0. The path can be specified by indicating which portion each doughnut lies on. Your procedure will store its answer a global $n$-element array `position[]`, declared:

```
static int[] position;  // doughnut's position on final path (0=outbound, 1=inbound)
```

where `position[i] = 0` indicates that the $i$th doughnut is visited on the outbound portion of the trip, and `position[i] = 1` indicates that it is visited on the inbound portion. Since the first and last doughnuts are visited on both the outbound and inbound portions, the values `position[0]` and `position[nd-1]` are ignored. Given this array, our main program will print the path for you and compute its length. (There is an obvious source of ambiguity, since a path and its reversal have the same length. By convention, the doughnut with index 1 should go on the outbound portion. If you fail to do this, however, don't worry. Our printing program will reverse your path as needed so this is true.)

For example, given the above input, doughnuts 3 and 7 are on the inbound portion, and so your procedure would set the path array as follows, where '`*`' indicates that the value can be either 0 or 1:

```
      i:  0  1  2  3  4  5  6  7  8
path[i]:  *  0  0  1  0  0  0  1  *
```

## Examples:

| Example 1 Input: | Example 1 Output (as generated by our main program): |
|---|---|
| 9<br>0 4<br>1 10<br>2 7<br>3 1<br>4 5<br>8 5<br>9 7<br>10 2<br>12 6 | Doughnuts: (0,4) (1,10) (2,7) (3,1) (4,5) (8,5) (9,7) (10,2) (12,6)<br>Path: 0 1 2 4 5 6 8 7 3 0<br>Length: 46 |

**Hint:** Don't try to solve the problem in a brute-force manner by enumerating all possible paths. (It will take too long.) Scan left to right, maintaining the optimal cost of the two parts of the path.

## Practice 1 Lisa's Zoo

After a school trip to the Springfield zone, Lisa decides she wants to redesign the zoo so that each animal species can live in its own rectangular *zone*. To aid her, she needs to write a program that can decide whether any zones overlap.

Two zones are considered to overlap if one contains part of the other, if they share a portion of a common side, or if they share a common corner.

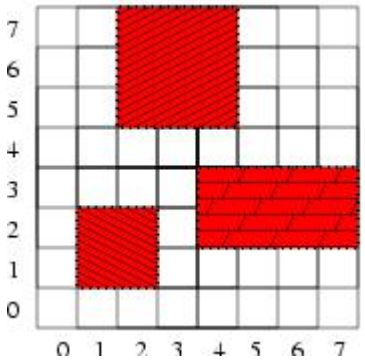
Each zone will be described by the (x,y) coordinates of its lower left and top right corners, where (0,0) indicates the bottom left corner of the zoo. You may assume that all zones fit within a 20 by 20 area (coordinates range between 0..19).

### Input/Output Format:

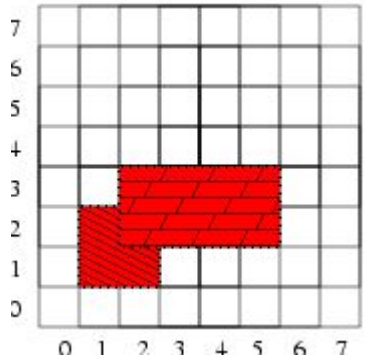
The input will consist of a line containing the number of animal zones, followed by one line per zone in the format: x1 y1 x2 y2 where (x1,y1) is the lower left-hand corner of the rectangle and (x2,y2) is the upper right-hand corner of the rectangle. The output will be "Overlap" if any zones overlap, otherwise "No Overlap".

### Examples:

| Example 1 Input: | Example 1 Output: |
|---|---|
| 3<br>1 1 2 2<br>2 5 4 7<br>4 2 7 3 | No Overlap |




| Example 2 Input: | Example 2 Output: |
|---|---|
| 2<br>1 1 2 2<br>2 2 5 3 | Overlap |



## Practice 2 Bart's Palindromes

Bart is trying to come up with funny palindromes, and decides to write a program to check whether a string is a palindrome. The program should read strings on separate lines and determine whether the entire string is a palindrome. The program should only check letters (i.e., a-z, A-Z), and ignore all blanks, numbers, and punctuation.

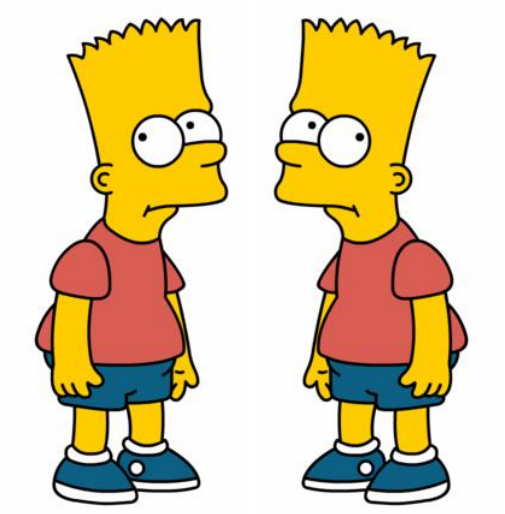
### Input/Output Format:

The input will consist of a sequence of lines, each of which should be tested separately. Each line will be no greater than 100 characters long. A blank line signals the end of input. After each line, the program should print on a separate line, either YES (if the input was a palindrome) or NO (otherwise).

### Examples:

| Example 1 Input: | Example 1 Output: |
|---|---|
| `radar` | `Yes` |
| `rAd. Ar` | `Yes` |
| `radiar` | `No` |
| `able was I ere I saw elba!` | `Yes` |

| Example 2 Input: | Example 2 Output: |
|---|---|
| `b` | `YES` |
| `bb` | `YES` |
| `ab` | `NO` |
| `aabaa` | `YES` |
| `aabbaa` | `YES` |
| `aab2aa` | `YES` |
| `aabaaa` | `NO` |
| `This is a palindrome` | `NO` |
| `radar is a palindrome` | `NO` |
| `a man, a plan, a canal, panama!` | `YES` |
| `a bell was eye, ear I saw elbow` | `NO` |
| `go hang a salami, bub, i'm a lasagna hog!!` | `YES` |



## Practice 3 Homer Plays Poker

Homer saw a poker tournament on TV, and decides he wants to start playing as well. He and Bart start playing poker at home, when Bart teaches him how to play a game called Razz. In Razz, each player receives 7 cards. Instead of trying to get the highest 5-card poker hand, the winner is the player with the *lowest* 5-card hand (using any 5 cards out of the 7 cards the player holds). Straights (5 cards in sequence) and flushes (5 cards of the same suit) do not affect the low hand, only pairs. Aces are treated as low cards.

Since Homer is just learning how to play Razz, he decides to bet only when he has the best possible Razz hand, which is Ace, 2, 3, 4, and 5 (in any order). Your job is to help Homer determine whether he has this 5-card hand out of his 7 cards.

### Input/Output Format:

Cards are treated as numbers, with Aces, Kings, Queens, and Jacks represented as 1, 13, 12, 11, respectively. 7-card hands are read in as 7 numbers on each line. For every hand, your program should output on each line "Bet" if the lowest possible 5-card hand is possible, and "Fold" if it is not.

### Examples:

| Example 1 Input: | Example 1 Output: |
|---|---|
| `1 2 3 4 5 6 7` | `Bet` |
| `1 2 3 4 6 7 8` | `Fold` |
| `1 2 3 3 4 4 4` | `Fold` |
| `11 1 5 4 12 2 3` | `Bet` |
| `5 1 3 2 6 7 9` | `Fold` |