# Problem 1 – Sum to 13

You are given three number **a, b** and **c** as an input. Your task is to tell if the **three numbers can be summed to** **13**, by only **changing their signs**.

### Input

* On the first line, you receive the numbers **a, b** and **c**, separated by spaces.

### Output

* Print "**Yes**" if the numbers can be summed to 13, or "**No**" if it is impossible.

### Constraints

* **a**, **b**, **c** will be whole numbers in the range **[-1 000 000…1 000 000].**
* Allowed time: **100 ms**. Allowed Memory: **16 MB**.

### Examples

|  |  |  |
| --- | --- | --- |
| ****Input**** | ****Output**** | ****Comment**** |
| **-10 2 -1** | **Yes** | We switch the signs of -10 and -1 so that we get 10 + 2 + 1 = 13 and thus print "Yes" |

|  |  |  |
| --- | --- | --- |
| ****Input**** | ****Output**** | ****Comment**** |
| **-1 10 -1** | **No** | Regardless of what sign switching we do, we can't get to 13, so we write "No" |

# Problem 2 – Draw the Halloween Pumpkin

We are given a number **n**, draw a Halloween pumpkin with size **n** as in the examples below.

### Input

* On the first line, we receive the number **n**.

### Output

* Print a Halloween pumpkin with size **n**.

### Constraints

* The number **n** is a whole number in the range **[3…100]**.
* Allowed time: **100 ms**. Allowed Memory: **16 MB**.

### Examples

|  |  |
| --- | --- |
| ****Input**** | ****Output**** |
| **3** | **..\_/\_..**  **/.^,^.\**  **\.\\_/./** |

|  |  |
| --- | --- |
| ****Input**** | ****Output**** |
| **4** | **...\_/\_...**  **/..^,^..\**  **|.......|**  **\..\\_/../** |

|  |  |
| --- | --- |
| ****Input**** | ****Output**** |
| **5** | **....\_/\_....**  **/...^,^...\**  **|.........|**  **|.........|**  **\...\\_/.../** |

# Problem 3 – Duplicated Letters

We are given a string **S**. The objective is to remove all **special duplicates** from it**.** A **special duplicate** represents two consecutive equal letters (example: aa; qq). The removal of a **special duplicate** is considered a **single operation**. After removing all possible special duplicates from the string **S**, print the **remaining string** and the **number of operations** needed to transform it, if all letters are removed, print **“Empty String”** instead of the remaining string.

### Input

* The input comes from the console and consist of a single line containing the string **S**.

### Output

* On the **first line**, we should print the remaining string **S** after removing all special duplicates from it.
* On the **second line**, we should print the number of operations needed for the end result in the following form "**N** operations", where **N** is the number of operations.

### Constraints

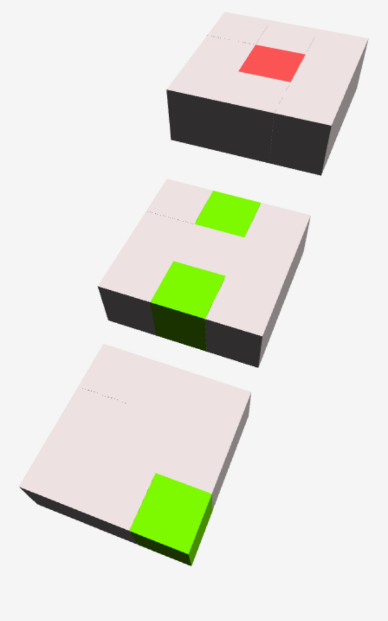
* The letters in the string **S** will always be lowercase letters in the range **[a-z]**.
* **S** will have a length in the range **[1…10000].**
* Allowed time: **100 ms**. Allowed Memory: **16 MB**.

### Examples

|  |  |  |
| --- | --- | --- |
| ****Input**** | ****Output**** | ****Comment**** |
| abaabccc | ac  3 operations | We start with the string **abaabccc**, we remove the first special duplicate found **aa**. This leaves us with the string **abbccc**, looking through it we remove the next special duplicate found **bb**, getting the string **accc**, removing the last special duplicate **cc**, we get the result **ac.**  **abaabccc -> abbccc -> accc -> ac** To get to the end string we performed **3 operations (removals)**. |

|  |  |  |
| --- | --- | --- |
| ****Input**** | ****Output**** | ****Comment**** |
| **baabccdd** | **Empty String**  **4 operations** | We start with the string **baabccdd**, we remove the first special duplicate found **aa**. This leaves us with the string **bbccdd**, looking through it we remove the next special duplicate found **bb**, getting the string **ccdd**, removing the next special duplicate **cc**, we get the string **dd**, after one last removal we're left with an **Empty string**.  **baabccdd -> bbccdd -> ccdd -> dd -> <empty string>** To get to the end string we performed **4 operations (removals)**. |

# Problem 4 – Snake

 We are given a **cube** of Latin letters of size **n \* n \* n** given as **n layers** (square matrices) of size **n \* n**. The cube is filled with the symbol "**o**", which indicates an empty cell. The symbol "**s**" (the red one in the figure) indicates the spawning point of the snake. With the symbol "**a**" (the green squares in the figure) apples are specified on the map, which the snake can eat. A cube, split into layers, is shown on the right. First an initial direction **d** is given to us, after which we receive commands in the form “**d** in **j** steps”, specifying the path that the snake will have to follow. The goal is to print **the number of apples collected** during the snake's movement. If the snake leaves the map’s bounds during its movement all following movement commands are ignored and we print the extra message **"The snake dies."** on the second line.

### Input

* The first line contains a whole number **n** – the size of the cube.
* At the next **n** lines the layers of the cube are given (from top to bottom) as a sequence of **n** matrices separated by " **|** ".
* At the next line, we are given the initial direction **d**, which the snake will take.
* All lines that are to follow are a total of **m** commands in the form "**d** in **j** steps" indicating the trajectory, which the snake has to follow. At the end, we are given a command in the form of "end in **j** steps", which is the end of the commands.

### Output

* On the first line print the number of apples which the snake eats during its trajectory in this format: “Points collected: **p**”, where **p** is the number of apples.
* If during its trajectory, the snake leaves the bounds of the map, print “**The snake dies.**” at the second line.

### Constraints

* The number **n**, showing us the size of the cube will always be a whole number in range **[2…50].**
* All cells in the cube will contain one of the following small Latin letters: **[o, s, a].**
* The initial direction **d** as well as the direction in the commands will always be one of the following **[up, down, forward, backward, left, right]**
* The number **j**, showing the number of steps before a command will always be a whole number in the range **[1…10 000].**
* The number of commands **m** will be a whole number in range **[1…100]**.
* Allowed time: **100 ms**. Allowed Memory: **16 MB**.

### Examples

|  |  |  |
| --- | --- | --- |
| ****Input**** | ****Output**** | ****Comment**** |
| **3**  **ooo | oao | ooo**  **osa | ooo | ooo**  **ooo | oao | ooa**  **forward**  **down in 1 steps**  **backward in 1 steps**  **down in 2 steps**  **right in 1 step**  **end in 1 step** | **Points collected: 3** | **We spawn the snake at the **s** symbol with the direction forward. The number of steps we have to make in our current direction, we get from the next line. After making 1 step forward, we change our direction to down. After 1 step down (now we’re at the second floor) we change our direction to backwards. We make 2 steps backwards, change the direction to down and continue for 1 step, before changing direction to right. Reading the end line, we see that we should make 1 step in the current direction after which we should stop.** |

|  |  |  |
| --- | --- | --- |
| ****Input**** | ****Output**** | ****Comment**** |
| **2**  **so | oo oa | ao**  **right**  **down in 1 step**  **end in 2 step** | **Points collected: 0**  **The snake dies.** | **We spawn the snake at s. We proceed right for 1 step, after which we change our direction to down. Reading the end command, we see that we should continue for 2 steps before ending the move commands. Moving down, the second step puts us out of the map and the snake dies, collecting 0 apples.** |

# Problem 5 – Parking Zones

You want to find the best place to park your car, in order to quickly go to the nearby store. We are given **n** parking zones, each parking zone has **x** and **y** (the coordinates of the top left corner of the parking zone), **width**, **height** and a price **p –** the price for **a minute stay** at the zone. We are also provided **s** free parking spaces in the form of their top left **x** and **y** coordinates, every parking space belonging to a parking zone. We also know that each block on the coordinate system is a square with dimensions exactly **1 x 1**. Additionally, we are given a target block **t** (given as its top left **x** and **y** coordinates)and a constant **k** which tells us the number of seconds needed to traverse **one block** (1 unit on the coordinate system). For the purposes of calculating the distance to **t**, movement is considered possible only in the four **major directions** (north, south, east, west), **moving diagonally is** **NOT allowed**.

The goal is to find the parking space, from which the journey to the target block **t** and backcosts the **least amount of money**, if multiple such spaces exist then we are to find the one that takes us the **least amount of time** to reach **t.** For the purpose of calculating the price for the stay, the minutes needed to reach the point and come back have to be **rounded up**, because for 2 minutes and 5 seconds we are taxed for 3 minutes

### Input

* On the **first line,** we receive the number **n** that shows us the number of parking zones that we will receive.
* On the next **n** lines, we receive the zones in the format "**[Name of Zone]: [x], [y], [width], [height], [price/min]**"
* On the next line, we receive a list of the free parking spots in the format "**x, y**" separated by "**;**"
* On the next line, we receive the target point **t**, again in format "**x, y**"
* On the last line, the constant **k** is given, showing us **the time in seconds needed to traverse a single block** on the coordinate system.

### Output

* On a single line print the string “**Zone Type: [Zone Name]; X: [x]; Y: [y]; Price: [Price]**”, where **[Zone Name]** is the name of the zone in which the parking space with the best price and time to **t** is, **[x]** and **[y]** are the top left coordinates of the parking space and **[price]** is the price we’ll have to pay for the stay **formatted to 2 decimal places** **(pay attention that the number of minutes needed to reach the point and come back has to be rounded up, because for 2 minutes and 5 seconds we are taxed for 3 minutes).**

### Constraints

* Each parking space will always be part of a defined parking zone.
* Parking zones will never overlap.
* There will never be 2 parking spaces with equal distance from the target in the same zone.
* There will never be 2 parking spaces with the best price and time from the target point.
* The number of zones **n** will always be a whole number in the range **[1…10].**
* The price for staying a minute at a parking zone **p** will always be a real number in the range **[1…1000].**
* All input values except **p** and **n** will always be integers in the range **[1…1 000 000].**
* Allowed time: **100 ms**. Allowed Memory: **16 MB**.

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| **2 Blue: 0, 0, 10, 10, 1.0 Green: 0, 10, 10, 10, 0.5 1, 1; 5, 5; 1, 11; 5, 16 15, 10 20** | **Zone Type: Green; X: 1; Y: 11; Price: 5.00** |
| **Comments** | |
| We find the closest parking spot to t from each zone.  For the **Blue zone** this is the parking spot with coordinates **5, 5**. The total blocks we need to walk from it to the target point is **14**. We multiply the blocks by **2** to get the number of blocks needed **to reach t and return**, then we multiply by the time needed for each block **k** (in this case **20**) to receive the number of seconds that we have to stay. **14 \* 2 \* 20 = 560** / 60 (to transform the seconds into minutes) = **9 minutes and 20 seconds** (since we crossed the 9 minutes’ mark, we need to pay for 10 minutes). 10 \* price **p** for stay/min **1** = **10lv.**  For the **Green zone** this is the parking spot with coordinates **1, 11** (we don't check **[1,1]** and **[5, 16]** because the distance from them is obviously bigger). We do the same calculations, the total number of blocks is again **14**, so we calculate the time **14** **\*** **2** for a round trip **\* 20** for the number of seconds = **560** / 60 **= 9m20s**, which we round up to **10**, this time however the price per minute for the zone is lower, so we get **10 \* 0.5** (p) **= 5lv**.  **[1, 1]**  **t [15, 10]**  **[5, 5]**  **[1, 11]**  **[5, 16]**  **15 blocks**  **14 blocks**  **14 blocks**  **[0, 0]**  **[0, 10]** | |

# Problem 6 – Shop Keeper

You are the shop keeper of a store, you will be given a sequence of numbers - **S** representing the **products** **you have in stock** (each number representing a certain product ex. 1 = bread, 2 = flour and so on). You will also be given a **list of orders** placed by customers – **O**, represented as a sequence of numbers (each number in the list represents an order for a product of that type ex. 1 = a customer has ordered a bread, 2 = customer has ordered flour and so on). Fulfilling an order does not remove the desired product from your stock (imagine we have infinite amounts of all products in stock). After each order, you are allowed to **change a single product** in your stock to any product you wish, however the total amount of product types in your stock should remain constant. Knowing all orders in advance, your task is to calculate the **minimum amount of changes** you need to do in your stock, in order to service all of them.

### Input

* On the first line, we receive the sequence **S**, each element of **S** will be separated from the others by a single space.
* On the second line, we receive the list of orders **O**, each element of **O** will be separated from the others by a single space.

### Output

* On a single line print the **minimum number of changes needed to service all orders**, if it's impossible to service all of them write "**impossible**" instead.

### Constraints

* The possible product types will be integer numbers in the range **[1…3000].**
* The sequence of products in stock - **S** will contain only **unique** products.
* The sequence of the products in stock - **S** will contain between **[1…1500]** products.
* The sequence of the orders - **O** will contain between **[1…12000]** orders.
* Allowed time: **140 ms**. Allowed Memory: **32 MB**.

### Examples

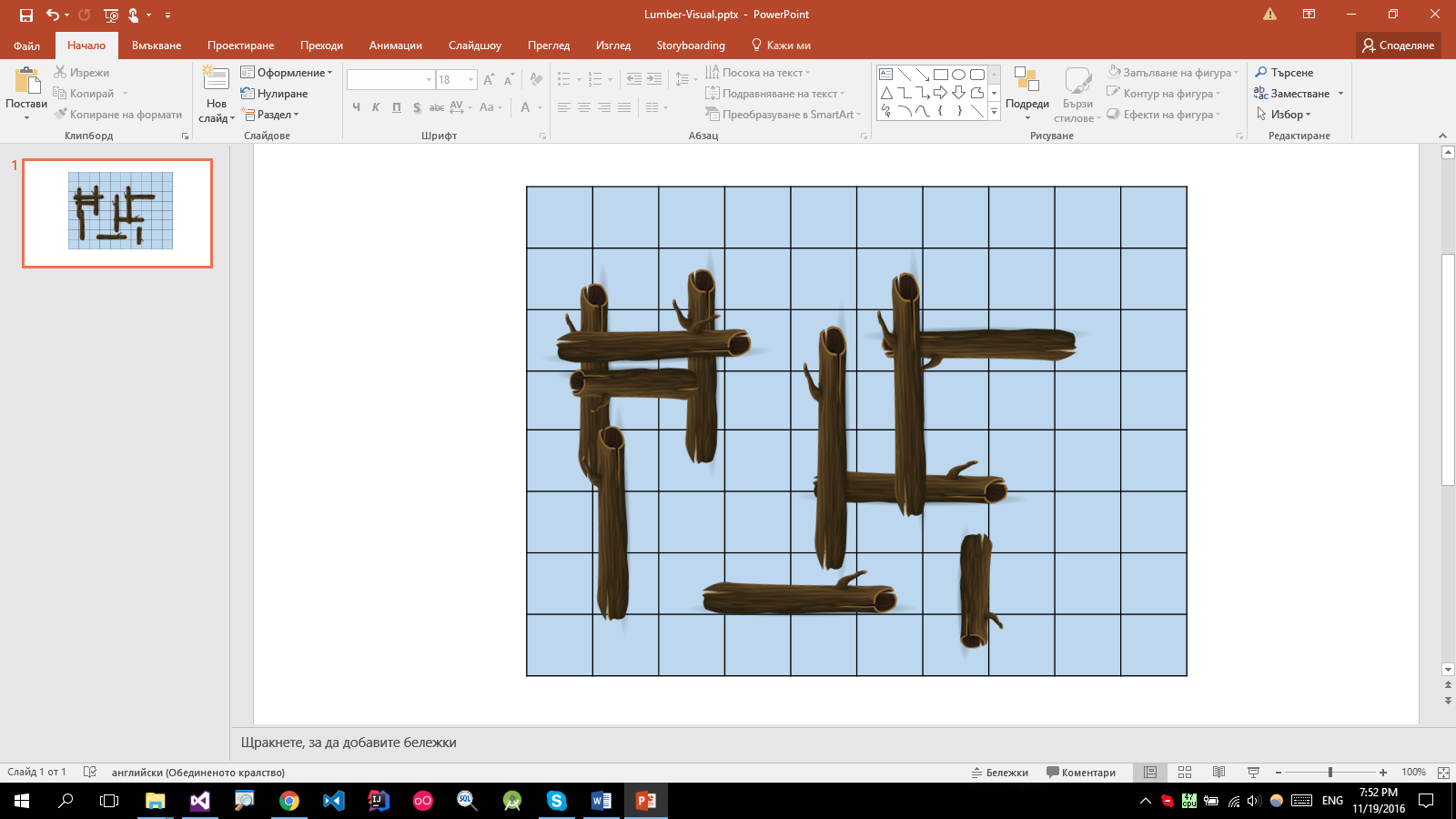
|  |  |
| --- | --- |
| **Input** | **Output** |
| 1 2 3 4  1 2 2 5 1 3 | 1 |
| **Comments** | |
| We start with item types 1, 2, 3 and 4 in stock, the first order comes and it wants an item type **1**, checking our stock we have the item so we fulfill the order. After fulfilling an order we have the option to change an item type in our stock, looking ahead in the list of orders we see that the next order is for an item type **2**, as we already have that item type in stock we decide not to change anything.  **After Order 1 -> No exchange**  The second order comes in, since we have **2** in stock, we fulfill that order. We are again offered the chance to change an item type in stock, looking ahead we see the next order is for a **2** as well, having the **2** in stock we decide not to change anything again.  **After Order 2 -> No exchange**  The third order comes in, it is for item type **2** again, so we fulfill it. Looking ahead in the order list we see the next order is for a **5**, since we don't have it in stock we have to exchange an item type for it. In this case we have two good choices, we can either change the **2** or the **4**, as we won't need them anymore. We arbitrarily choose to exchange the **2**.  **After Order 3 -> Exchange 2 for 5**.  **Stock after exchange:** **1 5 3 4**  The fourth order comes – it's for a **5**, since we exchanged the **2** for it we manage to service the order. Looking ahead, the next item is a **1**, since we have it we don't change anything.  **After Order 4 -> No exchange**  The fifth order comes in for a **1**, we have it in stock so we fulfill it. Looking ahead the last order is for a **3**, as we also have that in stock we skip any changes again.  **After Order 5 -> No exchange**  The sixth and last order comes in for a **3**, we have it in stock so we fulfill it. Having fulfilled all orders we write the number of changes we did **1**. | |

|  |  |
| --- | --- |
| **Input** | **Output** |
| 5 3 1 8  8 10 13 13 5 3 1 13 3 8 14 10 | 5 |

|  |  |
| --- | --- |
| **Input** | **Output** |
| 5 6 7 8  1 6 6 6 8 | impossible |
| **Comments** | |
| We take a look at the first order – 1, we don't have it in stock so we immediately fail the order, thus we print "**impossible**". | |

# Problem 7 – Lumber

The Iskar river has **lumber logs** floating in its waters (see image below). Habibi is a beaver who wants to know if he can move from log **X** to log **Y**.



A log is a **rectangle**. It is defined by two corners: **top-left A**(**Ax**; **Ay**) and **bottom-right B**(**Bx**; **By**). Habibi can travel between two logs if they touch each other (their coordinates intersect).

Write a program that tells Habibi if he can travel between two arbitrary logs.

### Input

* On the first input line you will be given the number of logs **N** and the number of queries **M** as 2 space-separated integers.
* On the next **N** lines you will be given the coordinates of each log in the format "**Ax Ay Bx By**".
* On the next **M** lines you will be given queries in the format "**X Y**" where **X** and **Y** correspond to logs in the **order** they were given in the input (starting from **1**).

### Output

* For each query print "**YES**" if the two logs are connected. Otherwise, print "**NO**".

### Constraints

* The number of logs **N** will be an integer in the range **[2..1000]**.
* The number of queries **M** will be an integer in the range **[1..10000]**.
* All log **coordinates** will be valid integer numbers in the range **[-100..100]**.
* Time limit: **100 ms**. Allowed memory: **16 MB**.

### Examples

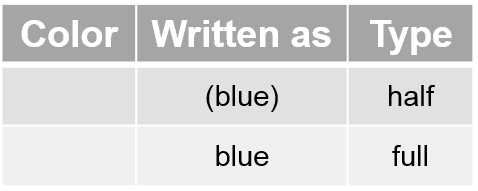
|  |  |  |
| --- | --- | --- |
| **Input** | **Output** | **Visual** |
| 4 3  -10 30 60 10  -50 20 -30 -20  -35 60 -20 15  -40 -10 50 -30  4 2  3 4  4 1 | YES  YES  NO |  |

|  |  |  |
| --- | --- | --- |
| **Input** | **Output** | **Visual** |
| 3 3  0 50 30 40  30 50 60 40  40 40 60 1  1 2  2 3  3 1 | YES  YES  YES |  |

# Problem 8 – Color Coding

You are given **n** pairs of **color sequences**. Each pair consist of **2** sequences of colors. A color can be **full** (represented as its normal text ex. "red", "blue") or **half** (written in brackets ex. "(red)", "(blue)").

The first sequence can contain both **half** and **full** colors, while the second will contain only **full** colors. With only the ability to **turn half colors into full** colors and to **delete all half colors**, you are asked if you can turn the first color sequence into the second one (keep in mind that the ordering of the elements matters). For each pair of sequences - if the first one can be transformed into the second one write “**true**”, otherwise write “**false**”.



### Input

* On the first line, we receive the number **n**.
* On the next **n \* 2** lines we receive a pair of color sequences in the format “**color1 color2 color3… colorN**”, each color separated by a space (check the examples below).

### Output

* For each pair of color sequences, print either “**true**” or “**false**” depending if the first sequence can be transformed into the second one.

### Constraints

* You are **NOT allowed to shuffle the elements** in any of the sequences – the ordering must be kept.
* A color can be one of the following **[red, green, blue, yellow, orange, black, purple, brown].**
* The number **n** is a whole number in the range **[1…100]**.
* Each sequence will consist of between **[1…100]** colors.
* Allowed time: **100 ms**. Allowed Memory: **16 MB**.

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| 1  (purple) (green) (purple) red blue  green red blue | true |
| **Comments** | |
| We look at the first sequence, we see we have a full red and blue and we only need a full green, so we make the half green full:  After that we delete all half colors:  In the end we’re left with the desired sequence, so we write **true**. | |

|  |  |
| --- | --- |
| **Input** | **Output** |
| 2  (blue) green (red) (blue)  blue red blue  (purple) red (blue) (yellow) blue (purple) (green) black  red blue green black | false  true |
| **Comments** | |
| Taking look at the first pair:  We look at the first sequence, while we see we can turn the half colors into full and get the desired 3 colors, we have no way of getting rid of the full green, thus it’s impossible to get the second sequence so we just write **false**.  Looking at the second pair:  We look at the first sequence, we see that we have a full red, blue and black and we only need a full green between the blue and black. Luckily we have a half green to turn into full, so we can complete the sequence.  Notice that while we have two ways to get the blue in the target sequence, we need to use the full one, because we have no way of otherwise getting rid of it.  Having found a way to get the target sequence we write **true**. | |

# Problem 9 – Merchants of Novigrad

The merchants of the city Novigrad want to know all possible routes to the city of Oxenfurt.

You are given **N** cities and **M** **one-directional** roads connecting them. Write a program that finds all distinct paths between the two cities. A path is considered unique if it does not use the same sequence of roads as any other path.

**Note**: Two roads can connect the same cities, but they are considered different.

### Input

* On the first input line you will be given the numbers **N** (cities) and **M** (roads)**.**
* On the next **M** lines you will be given 2 space-separated integers in the format "**U V"**, representing a **one-way road** from the city **U** to the city **V**.
  + Cities will be numbered from **1** to **N**. 1 will always represent Novigrad and N will represent Oxenfurt.

### Output

* There are two cases:
  + If there exists an infinite number of different paths from Novigrad to Oxenfurt, print "**infinite**".
  + Otherwise, print the **number of paths** modulo 109 and "**yes**" or "**no**" depending on if there exist infinite paths from Novigrad to **any other city** on the map

### Constraints

* The number **N** (cities) will be in the range **[2..10 000]**.
* The number **M** (roads) will be in the range **[1..100 000]**.
* Time limit: **300 ms**. Allowed memory: **30 MB**.

### Examples

|  |  |  |  |
| --- | --- | --- | --- |
| **Input** | **Output** | **Visual** | **Comments** |
| 7 7  1 3  4 2  3 4  3 2  2 7  6 5  5 6 | 2 no |  | There are 2 paths from **1** (Novigrad) to **7** (Oxenfurt):   * 1->3->4->2->7 * 1->3->2->7   There are no infinite paths to any other city so we print "**no**". |

|  |  |  |  |
| --- | --- | --- | --- |
| **Input** | **Output** | **Visual** | **Comments** |
| 6 8  1 3  1 2  1 4  2 6  3 6  3 2  4 5  5 4 | 3 yes |  | There are 3 paths from **1** to **6**:   * 1->2->6 * 1->3->2->6 * 1->3->6   There exist infinite paths from 1 to 5 (returning from 5 to 4 can extend the path forever). We print "**yes**". |

|  |  |  |  |
| --- | --- | --- | --- |
| **Input** | **Output** | **Visual** | **Comments** |
| 5 5  1 3  3 4  2 3  4 2  4 5 | infinite |  | The circular path  3->4->2->3 creates an **infinite** number of unique paths from 1 to 5. |

# Problem 10 – Shelter

You are Commander Shepard and you have **S** soldiers who are located in a warzone. When the enemy fires a missile, your soldiers must hide in one of the **M** nearby shelters. However, shelters have a limited capacity **C** (they cannot hold more than **C** soldiers).

Soldiers and shelters will be given as **2D points**. Assume all soldiers can move **1 unit** of distance per second.

Write a program that finds the **shortest time** it will take for all your soldiers to hide in a shelter.

### Input

* On the first input line you will be given **S** (number of soldiers), **M** (number of shelters) and   
  **C** (shelter capacity), separated by a single space**.**
* On the next **S** lines you will be given the coordinates of each soldier in the format "**X Y**".
* On the next **M** lines you will be given the coordinates of each shelter in the format "**X Y**".

### Output

* On the only ouput line print the **shortest time** it takes for all soldiers to take shelter (in seconds, rounded to the 6th digit after the decimal point).

### Constraints

* The numbers **S** (number of soldiers), **M** (number of shelters) and **C** (capacity) will be in the range **[1..500]**.
  + In 50% of the tests **S, M, K <= 15**.
  + There will always be enough shelters for all soldiers to hide.
* All given coordinates will be integers in the range **[-1000…1000]**.
  + There may be multiple soldiers/shelters with the same coordinates.
* Time limit: **1000 ms**. Allowed memory: **50 MB**.

### Examples

|  |  |  |  |
| --- | --- | --- | --- |
| **Input** | **Output** | **Visual** | **Comments** |
| 5 3 2  3 14  10 8  5 9  14 8  12 4  3 12  11 7  5 13 | 7.071068 |  | One of the optimal solutions.  Soldiers with coordinates (3,14) and (5,9) go to shelter (3,12). (12,4) and (14,8) go to shelter (11,7) and soldier (10,8) goes to shelter (5,13).  Notice that (10,8) can get to shelter (11,7) more quickly, but the capacity does not allow it. |