# Softuniada 24 and 25 March 2018

## Awesome numbers

Joro really likes numbers, but according to him some numbers are more awesome than others, help Joro write a program that can **classify numbers**. According to Joro numbers are classified according to **3 conditions**:

* Whether the number is **odd**
* Whether the number is **negative**
* Whether the **number can be divided by his favorite number without a remainder**

According to the 3 conditions, Joro classifies numbers in this way:

* If the number doesn’t **meet any of the conditions** (is not odd, is not negative and cannot be divided by his favorite number), then Joro thinks that number is “**boring”**
* If the number **meets only one condition**, then Joro thinks it is “**awesome**”
* If the number **meets 2 conditions**, then it is “**super awesome**”
* If the number **meets all 3 conditions**, then it is “**super special awesome**”

### Input

Two lines are read from the console:

* On the first line, there will be **the number Joro wants to classify**
* On the second line, there will be Joro’s **favorite** **number**

### Output

* On the only output line, print how Joro classifies the number:
  + Does not meet conditions -> “**boring”**
  + Meets 1 condition ->“**awesome**”
  + Meets 2 conditions ->“**super awesome**”
  + Meets 3 conditions ->“**super special awesome**”

### Constraints

* **The number that Joro wants to classify will be an whole number in the interval [-1 000 000…1 000 000]**
* **Joro’s favorite number** will bea **whole number** in the interval **[-1 000 000…1 000 000]**
* Allowed time: **100 ms**. Allowed memory: **32 MB**.

### Examples

|  |  |  |
| --- | --- | --- |
| **Input** | **Output** | **Comments** |
| 13 7 | **awesome** | The number **13**: odd **√** negative **X** can be divided by **7** **X** since the number meets only 1 condition we print „**awesome**” |
| **Input** | **Output** | **Comments** |
| -27 9 | **super special awesome** | The number **-27**: odd **√** negative **√** can be divided by **9** **√** since the number meets all conditions, we print „**super special awesome**” |
| **Input** | **Output** | **Comments** |
| 151734 2152 | **boring** | The number **151734**: odd **X** negative **X** can be divided by **2152** **X** since the number doesn’t meet any of the conditions, we print „**boring**” |
| **Input** | **Output** | **Comments** |
| -1158 -6 | **super awesome** | The number **-1158**: odd **X** negative **√** can be divided by **-6** **√** since the number meets 2 of the conditions, we print „**super awesome**” |
| **Input** | **Output** | **Comments** |
| -33 12 | **super awesome** | The number **-33**: odd **√** negative **√** can be divided by **12** **X** since the number meets 2 of the conditions, we print „**super awesome**” |

## Easter egg

Write a program that reads from the console a **whole even number** **n**, as in the examples bellow. The Easter egg has **width** **– 5 \* n** and **height - 2 \* n + 3**. **Your task is to write a program that outputs an Easter egg**.

### Input

* The input is read from the **console** and contains a single line – the number **n**.

### Output

* Print on the console an **Easter egg** as in the examples.

### Constraints

* The number **n** will always be a **whole even number** in the interval **[2…28]**
* Allowed time: **100 ms**. Allowed memory: **32 MB**.

### Examples

|  |  |  |  |
| --- | --- | --- | --- |
| **Input** | **Output** | **Input** | **Output** |
| 6 | ............\*\*\*\*\*\*............  ..........\*++++++++\*..........  ........\*\*++++++++++\*\*........  ......\*\*\*++++++++++++\*\*\*......  .....\*\*================\*\*.....  ....\*\*==================\*\*....  ...\*\*====================\*\*...  ...\*\*~~~~HAPPY EASTER~~~~\*\*...  ...\*\*====================\*\*...  ....\*\*==================\*\*....  .....\*\*================\*\*.....  ......\*\*\*++++++++++++\*\*\*......  ........\*\*++++++++++\*\*........  ..........\*++++++++\*..........  ............\*\*\*\*\*\*............ | 8 | ................\*\*\*\*\*\*\*\*................  ..............\*++++++++++\*..............  ............\*\*++++++++++++\*\*............  ..........\*\*\*++++++++++++++\*\*\*..........  ........\*\*\*\*++++++++++++++++\*\*\*\*........  .......\*\*======================\*\*.......  ......\*\*========================\*\*......  .....\*\*==========================\*\*.....  ....\*\*============================\*\*....  ....\*\*~~~~~~~~HAPPY EASTER~~~~~~~~\*\*....  ....\*\*============================\*\*....  .....\*\*==========================\*\*.....  ......\*\*========================\*\*......  .......\*\*======================\*\*.......  ........\*\*\*\*++++++++++++++++\*\*\*\*........  ..........\*\*\*++++++++++++++\*\*\*..........  ............\*\*++++++++++++\*\*............  ..............\*++++++++++\*..............  ................\*\*\*\*\*\*\*\*................ |

## Bingo generator

Misho and Deni thought up a new game called “Bingo”, but you have to help them and create a generator, which checks whether an entered **4-digit number** **wins**. The rules of the game are as follows:

1. A 4 digit number is entered, from which we have to create two new **2-digit numbers**.
2. The two **2-digit numbers** are created in the following way:
   1. **The first number** consists of the **thousands** digit and the **tens digit** in the **4 digit number**.
   2. **The second number** consists of the **hundreds digit** and the **units digit**.
3. After we get the **two 2-digit numbers**, we need to calculate the **ceiling**. The **ceiling** is calculated by adding together the **two** **2-digit numbers.** For example **1234** – the first **2-digit number** is **13**, while the second is **24**, their sum is 13 + 24 = **37**, so the **ceiling** is equal to **37**.
4. As a next step, we have to create a new 4-digit number – **the starting element**. **The starting element** is created by **appending the second 2-digit number to the back of the first 2-digit number**. For example, if the first 2-digit number is **13** and the second is **24**, the starting element will be **1324**.
5. The next step is to generate all 4-digit numbers, which meet the following rules. The order of generation should be the following – we start from the **starting** **element**, first we **increase** **the** **second** **2-digit** **number** to its limit, after which we **increase** **the** **first** **2**-**digit** **number**, **reset** the second 2-digit number and **repeat** until there is no more viable numbers to generate. For example, with a **starting** **element** of **1123** and a **ceiling** of **37**, we generate them in the order 1123 -> 1124 -> 1125… 1137->1223->1224->1225…->1237->1323…:
   1. They should be **equal or bigger** than **the starting element**
   2. The **2-digit number** generated from the first 2 digits of the newly generated number, should be **less than or equa**l to the **ceiling**.
   3. The **2-digit** **number** generated from the last 2 digits of the newly generated number, should be **less than or equa**l to the **ceiling**.

If we take the number **1334** as an example, with a **ceiling** of **37** and a **starting** **element** **1324**, we can see that it meets the requirements**:**

* The number is bigger than the starting element => **1334 > 132**4
* The 2-digit number created from the first 2 digits is less than the ceiling => **13 < 37**
* The 2-digit number created from the last 2 digits is less than the ceiling => **34 < 37**

If we take the number **1342** as an example, it **is NOT valid**, because it does not meet the requirement that the second two digits should be less than the ceiling => **42 > 37**

1. After creating the new **4-digit numbers** we need to check whether they **can be divided by 12 or 15** and to print them in two groups. If the amount of numbers that can be divided by 12 **is equal** to the amount of numbers that can be divided by 15, then the player who entered the 4-digit number is a winner.

**\*IMPORTANT: If any of the newly generated 4-digit numbers happens to be divisible by both 12 and 15 it should be printed in both groups.**

### Input

* **A single line** – the entered 4-digit number

### Output

**Three lines:**

* **First line :** Dividing on 12: {the newly generated 4-digit numbers that can be divided by 12 in ascending order}
* **Second line :** Dividing on 15: {the newly generated 4-digit numbers that can be divided by 15 in ascending order}
* **Third line:** If the amountof numbers divisibleby 12 is equal to the amount of numbers divisible by 15 print **“**!!!BINGO!!!”, alternatively print **“**NO BINGO!”

### Constraints

* The 4-digit number will always be a whole number in the interval **[1000…9999]**
* If the ceiling is **bigger than or equal to 99**, then all **2-digit numbers** are valid.
* Allowed time: **100 ms**. Allowed memory: **32 MB**.

### Examples

|  |  |  |
| --- | --- | --- |
| **Input** | **Output** | **Comments** |
| 1213 | Dividing on 12: 1128 1224 1332 1428 1524 1632 1728 1824 1932 2028 2124 2232 2328 2424 2532 2628 2724 2832 2928 3024 3132 3228 3324 3432  Dividing on 15: 1125 1230 1425 1530 1725 1830 2025 2130 2325 2430 2625 2730 2925 3030 3225 3330  NO BINGO! | We take the **first digit** (the thousands digit **1**) and the **third digit** (the tens digit **1**) from the 4-digit number => we get the **first** **2-digit number**, which is 11. After that we create the **other 2-digit number**, which is 23 (the hundreds digit **2** and the units digit **3**). The **maximum value** that any of the **two 2-digit numbers** can take is their sum => **11 + 23 = 34**. After we generate the numbers (1123, 1124, … , 1224, … , 3434), we check each of them and place it in one of the two groups – those divisible by 12 or those divisible by 15.  **Because the amount of numbers divisible by 12 (which is 24) is different than the amount of numbers divisible by 15 (which is 16) we print** NO BINGO! |
| 1214 | Dividing on 12: 1128 1224 1332 1428 1524 1632 1728 1824 1932 2028 2124 2232 2328 2424 2532 2628 2724 2832 2928 3024 3132 3228 3324 3432 3528  Dividing on 15: 1125 1230 1335 1425 1530 1635 1725 1830 1935 2025 2130 2235 2325 2430 2535 2625 2730 2835 2925 3030 3135 3225 3330 3435 3525  !!!BINGO!!! | |

## Wrong results

We are given a cube (three dimensional array) with integer numbers. The cube represents the results of an experiment in a lab, but during the gathering of the data, a problem has occurred and **elements with incorrect values** were added in the cube. We know the **coordinates** of one of the incorrect values in the cube (**element**). We need to change all values in the cube equal to the **incorrect** **element’s** value.

The task is to **find in the cube all incorrect values and to rewrite them with the correct ones**. The correct value is the result of the **sum of all direct neighboring elements** – those elements that are in distance of one index (from the incorrect element), where there are none – nothing should be added. The values of the elements to the **left**, **right**, **up**, **down**, **front** and **back** should be taken. If an element has no neighbors with correct values, it should get the value - 0.

It is possible to have a neighboring element, whose value is also incorrect – in this case the value of the incorrect neighbor is NOT added to the sum made from the neighboring nodes (it should essentially be skipped), regardless if it has already calculated its own correct value or not.

### Input

* The first line contains a whole number **size** – the size of the cube.
* On the next **size** lines, we are given **size** amount of matrices, split with " **|** ", where each matrix represents a layer of the cube and contains **size** **\* size** elements.
* On the last line, we are given coordinates **x, y, z –** the indexes of the element, which we know has an incorrect value.

### Output

* On the first line, we have to print the number of found and corrected elements in the format **"**Wrong values found and replaced: elementsCount**"**, where **elementsCount** is the number of elements, which we have changed.
* On the next **size \* size** lines we need to print all **size** amount of elements in the matrix, with the correct values separated by a space **" "**.

### Constraints

* The **size** of thecube will always be a whole number in the interval **[3...25].**
* All cells of the cube will contain whole numbers in the interval **[-1000…1000]**.
* The coordinates of the element with incorrect value will always be whole numbers and will be inside the constraints of the cube.
* There will never be a cube consisting of the same value.
* Allowed time: **200 ms.** Allowed memory: **32 MB**.
* .

### Examples

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| ****Input**** | ****Output**** | ****Comments**** | | | |
| **3**  **1 2 4 | 6 7 8 | 1 1 4**  **4 6 7 | 4 9 3 | 4 4 4**  **8 9 4 | 4 2 4 | 7 9 4**  **0 1 0** | **Wrong values found and replaced: 11**  **1 2 17**  **15 6 7**  **8 9 16**  **6 7 8**  **15 9 3**  **17 2 5**  **1 1 9**  **8 19 3**  **7 9 9** | **According to the given coordinates, the incorrect value is equal to 4. We are looking for all elements equal to 4. We find the first element and calculate the sum of his neighbors 15 = (1 + 6 + 8). The current element doesn’t have neighbors in the front and left side, in his back side he has an element equal to 4, so we don’t add it to the sum. Analogously we find all elements in the cube, calculate the sums of their neighbors and correct their values.** | | | |
| **4**  **4 0 0 0 | 12 12 10 1 | 14 10 6 12 | 11 12 14 12**  **11 1 9 11 | 9 10 2 7 | 6 8 5 7 | 10 7 6 12**  **5 0 12 12 | 12 0 3 0 | 12 1 12 7 | 12 8 4 12**  **7 4 1 12 | 12 11 9 12 | 12 8 8 12 | 12 2 12 12**  **0 2 3** | | | **Wrong values found and replaced: 21**  **4 0 0 0**  **11 1 9 11**  **5 0 13 11**  **7 4 1 1**  **27 30 10 1**  **9 10 2 7**  **14 0 3 0**  **18 11 9 9**  **14 10 6 14**  **6 8 5 7**  **7 1 28 7**  **8 8 8 15**  **11 42 14 14**  **10 7 6 13**  **18 8 4 11**  **2 2 14 0** | **The incorrect value is equal to 12. The element at indexes 0 2 3 we have only one neighbor, whose value we can take and that is the element from the top side 11, all other neighbors contain an incorrect value as well, so we don’t add their values to the sum. At the last element, there is no neighbor, whose value can be added to the new sum, so the element on this index gets the value 0.** |

## Circumscribed Circle

You are given a sequence of shapes – **a triangle** and a **circle** in a two-dimensional Cartesian coordinate system. Find whether **the circle is circumscribed around the triangle (a circle is circumscribed around a triangle when the triangle’s vertices lie on the circle)** and **whether its center is inside the triangle, or outside (points on some of the triangle sides are assumed to be inside the triangle)**.

Each **triangle** is defined by three points: lower left , lower right, and upper . Each **circle** is defined by a center and radius.

### Input

* The input is read from the console.
* On the first line, there is the number of shape pairs – **n**.
* On the next **2** **\*** **n** lines there are the shape descriptions. Every line contains exactly one figure in one of the following formats:
  + **circle Ox, Oy, R**
  + **triangle Ax, Ay, Bx, By,Cx, Cy**
* The parameters will always be passed in this sequence.

### Output

* The output always consists of one line for each pair of shapes.
* For every combination, you have to print one of the following, on a single line:
  + The circle **is circumscribed** around the triangle and the circle **center is inside** the triangle, on a single line 🡪 **“The circle is circumscribed and the center is inside.”**
  + If the circle **is circumscribed** around the triangle but **the center is not inside it**, print

🡪 **“The circle is circumscribed and the center is outside.”**

* + The circle **is not circumscribed** around the triangle **but the center is** inside the triangle, on a single line 🡪 **“The circle is not circumscribed and the center is inside.”**
  + If the circle **is not circumscribed** around the triangle and **the center is not inside**, print 🡪 **“The circle is not circumscribed and the center is outside.”**

### Constraints

* **n** will be an integer in the interval **[1…1000]**.
* , , , , , , , , and are real numbers in the interval **[-1000...1000]** with no more than 5 digits after the decimal point.
* will always be a positive number.
* For all real (floating-point) numbers in the input, the decimal separator is “.”, for example “1.45” and “2.5”.
* When you perform calculations, assume that two values are considered the same if the difference between them is less than 0.01 units.
* Allowed time: **200 ms**. Allowed memory: **32 MB**.

### Examples

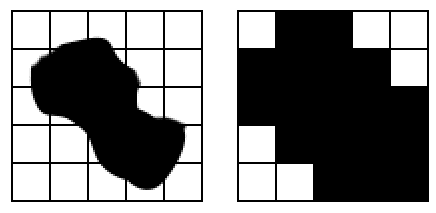
|  |  |  |
| --- | --- | --- |
| **Input** | **Output** | **Comments** |
| 1  circle -13.5, 19.5, 22.63  triangle -36.13, 19.5, 6, 8, -13.5, 42.13 | The circle is circumscribed and the center is inside. | The circle center is **at the same distance from all triangle vertices, therefore it’s circumscribed around it**. The sum of the areas of the triangles between the circle center and the triangle vertices **is equal** to the area of the triangle, therefore the center of the circle is inside the triangle. |
| 1  circle 3.5, 3, 2.69  triangle 1, 2, 6, 2, 1, 4 | The circle is circumscribed and the center is inside. | The circle center is **at the same distance from all triangle vertices, therefore it’s circumscribed around it**. The sum of the areas of the triangles between the circle center and the triangle vertices **is equal** to the area of the triangle, therefore the center of the circle is inside the triangle. |
| 1  circle -2, 1.5, 6.9  triangle -6, 4.5, 3.2, 1, 2, 6 | The circle is not circumscribed and the center is outside. | The circle center is **not at the same distance from all triangle vertices, therefore it’s not circumscribed around it**. The sum of the areas of the triangles between the circle center and the triangle vertices **is not equal** to the area of the triangle, therefore the center of the circle is not inside the triangle. |
| 1  circle 3.5, 3, 4.58  triangle 1, 4, 3.5, 0.31, 6, 4 | The circle is not circumscribed and the center is inside. | The circle center is **not at the same distance from all triangle vertices, therefore it’s not circumscribed around it**. The sum of the areas of the triangles between the circle center and the triangle vertices **is equal** to the area of the triangle, therefore the center of the circle is inside the triangle. |

## Asteroids

You are the captain of a spaceship travelling in a distant planetary system. You’re about to embark on the most important and most dangerous part of the journey: travelling through the densely populated asteroid belt. On the front monitor you see a picture very similar to this one:



You have a program which preprocesses the raw picture. It generates a rectangular grid (field) of monochromatic pixels. If a part of an asteroid fills in a part of a pixel, the pixel has a value **1**, otherwise – **0**. An example of how this program works is given below:



Find the **total number** of asteroids. **Group them by area** (total number of pixels on the screen) – from largest (which are most dangerous) to smallest. The asteroid from the example above has area 18.

### Input

You get a number of screens to process.

Each screen starts with a number containing the numbers **N** and **M**, separated by a single symbol „**x**“. This is the screen size.

The following line contain exactly binary values: **0** or **1**.

The last input line contains the string “**end**” only.

### Output

For each screen, print the groups of asteroids in the format **GxS**, where is the **asteroid area** and is **the number of asteroids of area S** in the field (read as “**G**” asteroids of area “**S**”). **Sort the groups from largest to smallest area**.

At the end of each screen, output the total number of asteroids in that screen in the format **Total: T**.

### Constraints

* Each field will contain at least one asteroid
* Allowed time: **350 ms**. Allowed memory: **32 MB**.

### Examples

|  |  |  |
| --- | --- | --- |
| **Input** | **Output** | **Comments** |
| 5x5  01100  11110  11111  01111  00111  end | 1x18  Total: 1 | The example from the problem statement. |

|  |  |  |
| --- | --- | --- |
| **Input** | **Output** | **Comments** |
| 3x3  100  000  001  5x4  1011  0011  1000  1111  0000  2x2  10  01  end | 2x1  Total: 2  1x5  1x4  1x1  Total: 3  2x1  Total: 2 | Three screens of different sizes, displaying asteroids. |

## Car

Kolyo bought a new second hand car for the eighth time in a row and he decided to try it out on the streets of Sofia, during the night. He has to be careful after each turn and to increase or decrease the speed of his car, before every street.

Your program will receive a list of **C** whole numbers, where the **i-th** **number** expresses how much of a change in speed Kolyo wants to make before the **i-th** street. Kolyo can choose to either **increase** or **decrease** **his current** **speed** with the given number. In addition to that, your program will receive a whole **B** – the **initial speed** of the car and another number **M** – the **maximum speed** possible speed of the car. In other word, Kolyo **cannot drive with a speed of less than 0 or more than M** (but he can drive with exactly 0 or exactly M). Your program should calculate what the **maximum speed** of the car is, which Kolyo can use for **the last street**. If there is no way to make all the consecutive changes in the speed, without it becoming larger than **M** or less than **0**, your program should print -1.

### Input

The input data will be read from the console and will always be correct:

* On the first line of the input will be the number **C** – the amount of numbers, with which Kolyo wants to change the speed of his car.
* On the second line, there will be **C** amount of whole number – where each consecutive number expresses the **change**, which Kolyo wants to make in the speed of his car, before the next street.
* On the next line, there will be the number **B** – the **initial speed** of Kolyo’s car**.**
* On the next line will be the number **M** – the maximum speed of Kolyo’s car.

### Output

* The output should be printed on the console:
* **On the only line of the output, your program needs to print either -1 or the maximum speed, with which Kolyo can pass through the last street.**

### Constraints

* The number of attempts to change the speed - **C** will be a whole number between **[1...50]**
* In **95%** of the test cases **C** will be less than **34**
* Each change of the speed will be a whole number between **[1...M]**
* The maximum speed **M** will be a whole number between **[1...1000]**
* The initial speed **B** will be a whole number between **[0…M]**
* Allowed time: **100 ms**. Allowed memory: **32 MB**.

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| 3  5 3 7  5  10 | 10 |

|  |  |
| --- | --- |
| **Input** | **Output** |
| 4  15 2 9 10  8  20 | -1 |

|  |  |
| --- | --- |
| **Input** | **Output** |
| 14  74 39 127 95 63 140 99 96 154 18 137 162 14 88  40  243 | 238 |

## Logistics

You own a courier company that delivers packages, on the first line you will receive the number **N** – the amount of packages you will receive. On the next **N** lines, the **packages** will be given – each **package** has a:

* **price** (the sum you will receive for delivering the package)
* **deadline** (the number of the day after which the package becomes invalid and cannot be delivered).

If a package has a deadline of **day 4** for example, then that means it can be delivered on days **1** to **4** – so it can be delivered on **day 1**, **day 2**, **day 3** and **day 4** (but not on day 5 and up). Packages are **indexed** in the order of receiving them from the input (the first package has index **1**, the second – index **2** and so on).

The company has a **single minivan** with which to deliver the packages and it can only deliver a **single package a day**. The driver is still new though and from time to time he gets in a car crash, if the driver crashes, **the company receives no money for the delivery** and has to **pay a price of 50** for the repair of the van, as well as reimburse the client for an **amount equal to the price of the package with which the minivan crashed**.

The goal of the task is to create a list of packages, which the minivan has to deliver each day, so that the company has the best **total balance** (maximum gain or least expenses, if the driver crashed too often).

If there exist multiple optimal distributions, the following rules should be followed:

* For example if we look at the following packages:
  + Package 1 – price 20, deadline 2
  + Package 2 – price 10 deadline 2

There are 2 optimal distributions – Distribution 1:

* + Day 1 -> Package 1 (price 20)
  + Day 2 -> Package 2 (price 10)

Or Distribution 2

* + Day 1 -> Package 2 (price 10)
  + Day 2 -> Package 1 (price 20)
* In case of packages which will be **delivered successfully**, the **ascending** **order** should be used (Distribution 2).
* Alternatively if we knew that the minivan **will crash in both days**, the packages should be ordered in **descending** **order** (Distribution 1)

### Input

* On the first line of input, we receive the number **N** – the number of packages.
* On each of the next **N** lines we receive a package in the format **„{price} {deadline}”**.
* On the last line, **all the days in which the minivan will crash** are written, separated from each other with spaces - **„{day of crash } { day of crash } … { day of crash }”,** if there will be no crashes – the string **„**none**”** is written on the line instead.

### Output

* On the first line of the output the **total balance** should be printed (the sum of money received from clients, minus the expenses for repairs and reimbursements).
* On the second line of the output, the sequence of packages which the minivan has to deliver is written in the format **„{index of package to be delivered on day 1} { index of package to be delivered on day 2} … { index of package to be delivered on the last day}”**

### Constraints

* The **number of packages** will be a whole number between **[1…100 000].**
* There will never be 2 packages with the same price.
* The **price** will be a whole number between **[1…1 000 000]**.
* The **deadline** will be a whole number between **[1…100 000].**
* The number of crashes will be a whole number between **[0…100 000],** but not more than the **highest deadline.**
* Allowed time: **450 ms**. Allowed memory: **32 MB**.

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| 4  25 3  15 1  10 1  50 3  none | 90  2 1 4 |
| **Comments** | |
| The indexed packages look like this: #1 25 3  #2 15 1  #3 10 1  #4 50 3  We check all packages, we immeadiately notice that **#4** and **#1** are the most expensive ones, but if we choose to deliver one of them on day **day 1**, we won‘t be able to deliver 2 packages on the next 2 days (packages **#2** and **#3** cannot be delivered after **day 1**, because their deadline would have passed). We look at the 2 packages with deadline **day 1**, with prices **15** and **10**. **#2** has the bigger price, so we choose it. For **day 2** and **day 3** we have a choice between 2 packages – **#1**, with price **25** and **#4** with price **50**. Because both packages have a deadline of **day 3**, we have 2 distribution, following the rules for distribution from the task description, we use the ascending order, thus on **day 2** we choose **#1** and on **day 3** - **#4**.The end result is the following: **Total balance** = 15(#2) + 25(#1) + 50(#4) = **90.00.**  **Distribution of the packages in the minivan**  2 1 4 | |

|  |  |
| --- | --- |
| **Input** | **Output** |
| 5  30 2  14 2  13 2  43 1  17 2  2 1 | -127  2 3 |
| **Comments** | |
| The indexed packages look like this: #1 30 2  #2 14 2  #3 13 2  #4 43 1  #5 17 2  From the input we see that the minivan will crash on **day 1** and on **day 2**. Because the minivan will crash, we choose to deliver the cheapest packages, those are **#2** and **#3**. Because both packages have a deadline of **day 2**, we have 2 ways to distribute them, following the rules for distribution from the task description, we use descending order, thus we deliver **#2** on **day 1** and **#3** on **day 2**.  The minivan will crash while delivering **#2** on **day 1**, resulting in us having to pay 50 for repairs and 14 to the client as reimbursement, making the total balance for the day 0 – 50 – 14 = **-64**  On **day 2** we have another crash, this time with package **#3**, as a result we pay another 50 for repairs and 13 for reimbursements. So finally our total balance after **day 2** is the following: **Total balance** = -64(day 1) - 50(crash on day 2) - 13(reimbursement for #3) = **-127.**  **Distribution of the packages in the minivan**  2 3 | |

|  |  |
| --- | --- |
| **Input** | **Output** |
| 7  21 2  18 2  13 4  2 1  6 1  4 3  33 4  2 | 13  1 6 3 7 |
| **Comments** | |
| The indexed packages look like this: #1 21 2  #2 18 2  #3 13 4  #4 2 1  #5 6 1  #6 4 3  #7 33 4  We look at the input and see that we have a crash on **day 2**, thus we choose the cheapest package that’s available on **day 2** – package **#6**. Looking at the packages, the most expensive one we can choose on **day 1** is obiously **#7**, if we deliver package **#7** on **day 1** though, we won’t be able to deliver packages **#1** or **#2** (because **day 1** will be occupied by package **#7** and during **day 2** there will be a crash). If we test all the possibilities, we can see that the most correct distribution would be: **Distribution of packages in the minivan**  1 6 3 7  Giving us a **Total balance** = 21(#1) – 50(crash 2) – 4(reimbursement for #6) + 13(#3) + 33(#7) = **13**  Let us check:  On **day 1** we will deliver **#1** with deadline **day 2**.  On **day 2** we will deliver **#6** with deadline **day 3**.  On **day 3** we will deliver **#3** with deadline **day 4**.  On **day 4** we will deliver **#7** with deadline **day 4**. | |

## Mafia

Your task is to bust global cyber mafia organizations. In each organization people are connected: each connection describes a pair of people who know each other.

The organization has computers. Each of them is labelled with a **unique identifier** from to . The computer with ID belongs to the **boss**, and the computer with ID – to his **closest subordinate**. You have only partial control over the remaining computers. You don’t know exactly who uses the computers but you do know the following:

* The computers in the entire organization are connected in a network
* Each direct connection between two computers in the network has a capacity : number of packets which can travel between them in one second.

You want to perform a DDoS attack to break the connection between the boss and his closest subordinate. Using a given model of the network, you have to estimate the maximum capacity of the network between the boss and his subordinate (in packets per second).

### Input

The input contains information about criminal organizations.

The description of each organizations starts with the number **N** on its own line – the total number of computers in that organization. The next rows contain the connections between computers in the format **A-B P**, where **A** is the identifier of the first computer, **B** is the identifier of the second computer and **P** is the capacity of the connection . The dash in the input data means “there is a connection”, not negation. **The links are bidirectional**.

The last input line contains the string “**end**” only.

### Output

For every cyber mafia organization, output its number and the capacity of the network between the mafia boss and his closest subordinate. Output the result in the format „**Group G: T**“, where is the number of the group (**1** for the first group from the input, **2** for the second group, etc.) and is the required capacity.

### Constraints

* Allowed time: **350 ms** Allowed memory: **32 MB**.

### Examples

|  |  |  |
| --- | --- | --- |
| **Input** | **Output** | **Comments** |
| 4  1-2 14  3-4 14  1-3 6  2-4 6  1-4 5  end | Group 1: 17 | The direct connection 1-4 has capacity 5. The indirect connections 1-2-4 and 1-3-4 have capacity 6.  The total capacity is 5 + 6 + 6 = 17. |

|  |  |  |
| --- | --- | --- |
| **Input** | **Output** | **Comments** |
| 3  1-2 5  2-3 4  5  1-2 4  4  1-2 14  3-4 14  1-3 6  2-4 6  2-3 5  end | Group 1: 4  Group 2: 0  Group 3: 17 | The example shows the behaviour of the program with several networks. In the first network, there’s only one possible way: 1-2-3 with capacity 4. In the second, only the computers 2 and 3 are connected; the mafia boss has no link to his subordinate. The third network is the same as the above. |

|  |  |  |
| --- | --- | --- |
| **Input** | **Output** | **Comments** |
| 6  1-2 10  2-3 20  1-4 15  4-5 7  3-6 13  5-6 22  4-3 19  end | Group 1: 20 |  |

## Durin’s Labyrinth

You are a famous archeologist, who found the lost labyrinth of Durin, in the **antechamber** you see a huge detailed map of the labyrinth. The labyrinth is devised of a series of **rooms** connected together by narrow **tunnels**, where each room has a number (the **index** of the room), on the map you also see 2 specifically marked rooms – the **antechamber** in which you are right now and the **treasury**. Underneath the map, a riddle is inscribed which states that to open the **treasury**, you must know the **secret code**, in the riddle it’s also written how the **secret code** can be calculated. According to the riddle, regardless of which path you take from the **antechamber** to the **treasury**, there will be some **rooms through which you will always have to pass**, the riddle calls these rooms - **key rooms**. In the riddle it is also mentioned that the **antechamber** **and** **the** **treasury are NOT considered key rooms**. According to the text the **secret** **code** can be calculated by adding together the numbers (**indexes**) of all key rooms.

### Input

* On the first line we receive the number **N** – the number of rooms in the labyrinth.
* On the second line, we receive the number **M –** the number of tunnels in the labyrinth.
* On each of the next **M** lines, we receive information about a **tunnel** which connects 2 **rooms** in the format **„{index of room 1} {index of room 2}“**.

### Output

* On the first line of the output, we need to print the **secret** **code**.
* On the second line of the output, we need to print the **indexes** of all **key** **rooms** in the order in which we traversed them from the **antechamber** to the **treasury** in the format **„{index of key room 1}->{ index of key room 1}->…{ index of last key room}”**.

### Constraints

* The **Indexes** oftheroomswillalwaysbethenumber **[0…N-1]**.
* The **antechamber** will always be the room with index **0**.
* The **treasury** will always be the room with index **N-1**.
* The number of **rooms** will be between **[3…20 000]**.
* The number of **tunnels** will be between **[2…30 000]**.
* There will always be a path between the **antechamber** and the **treasury**.
* There will always be at least one **key** **room**.
* All **tunnels** can be traversed both ways.
* Allowed time: **350 ms** Allowed memory: **32 MB**.

### Examples

|  |  |  |
| --- | --- | --- |
| **Input** | **Visualization** | **Output** |
| 6  6  0 1  0 2  1 2  1 3  3 4  3 5 | **0**  **2**  **5**  **1**  **4**  **3** | 4  1->3 |
| **Comments** | | |
| We start from the **antechamber** (**#0**) and want to reach the **treasury** (**#5**), when we look at the map, it is obvious that no matter what path we choose, we will have to pass through rooms **#1** and **#3** – therefore it is obvious that the key rooms in this labyrinth are rooms **#1** and **#3**. Let’s look at the possible paths from the **antechamber** to the **treasury**.  Path 1:  **0**->**1**->**3**->**5**  Path 2:  **0**->2->**1**->**3**->**5**  After we know the **key** **rooms**, it’s time to find the **secret** **code** – we add up the **indexes** of the **key** **rooms** and we get 1 + 3 = **4**.We see that regardless of the path, we will always have to first go through room **#1**, then room **#3**, therefore we already have the correct order of the **key** **rooms** **1->3**, with which we finish the task. | | |

|  |  |  |
| --- | --- | --- |
| **Input** | **Visualization** | **Output** |
| 10  11  0 6  6 7  6 1  7 4  7 8  1 2  1 4  2 3  2 5  3 4  5 9 | **0**  **7**  **9**  **5**  **2**  **4**  **3**  **1**  **6**  **8** | 13  6->2->5 |

|  |  |  |
| --- | --- | --- |
| **Input** | **Visualization** | **Output** |
| 12  15  0 1  0 3  0 4  1 2  1 10  2 10  3 4  4 5  4 8  8 9  5 6  5 7  6 7  11 6  7 11 | **0**  **1**  **2**  **3**  **4**  **7**  **11**  **8**  **9**  **5**  **6**  **10** | 9  4->5 |