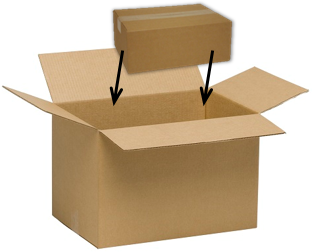
**Problem 1 – Fit Box in Box**

Nakov likes boxes. Ha has many boxes at home. To save space he often puts his boxes one inside another. A **box can fit inside another box** if the size of the smaller box is smaller than the size of the bigger box **in all dimensions** at the same time. Of course, boxes can be rotated to fit one inside another. Here are few examples:

* (5, 1, 6) fits naturally in (6, 2, 9), because 5 < 6 and 1 < 2 and 6 < 9
* (5, 1, 6) fits in (2, 7, 6) after rotating the second box to (6, 2, 7)
* (7, 8, 1) cannot fit in (12, 7, 3) even after rotating
* (6, 2, 9) cannot fit in (5, 1, 6) even after rotating

You are given the sizes of **two boxes** (width, height and depth). Write a program to check whether the boxes can fit one inside another and how exactly. Print the **smaller box first**, exactly as it is given in the input (without rotating), followed by "**<**" and the **larger box** (eventually rotated) to fit each of the dimensions. **Print all possible ways to put the smaller box into the larger box**. In case there is no smaller box, print nothing. See the examples below to catch the idea.

**Input**

The input data comes from the console. It holds exactly **6 different numbers**, each at a separate line:

* The first 3 lines hold the dimensions of the first box (width, height and depth).
* The second 3 lines hold the dimensions of the second box (width, height and depth).

The input data will always be valid and in the format described. There is no need to check it explicitly.

**Output**

The output should be printed on the console. It should consist of zero or more lines:

* If a smaller box exists, print **all possible ways** to fit the smaller box (without rotation) in the larger box (eventually rotated) in format: **(w1, h1, d1) < (w2, h2, d2)**. Note that the first box is smaller if **w1** **<** **w2** and **h1** **<** **h2** and **d1** **<** **d2**. Beware of **whitespaces** in the output format!
* Print each possible fit into a separate line. The lines **order is not important**.
* In case of no smaller box exists, **print nothing**.

|  |  |
| --- | --- |
| **Input** | **Output** |
| 10  12  11  3  5  2 | (3, 5, 2) < (10, 12, 11)  (3, 5, 2) < (10, 11, 12)  (3, 5, 2) < (12, 10, 11)  (3, 5, 2) < (12, 11, 10)  (3, 5, 2) < (11, 10, 12)  (3, 5, 2) < (11, 12, 10) |

**Constraints**

* All input numbers are **unique** integers in [1 … 1000].
* Time limit: 0.1 seconds.
* Memory limit: 16 MB.

**Examples**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Input** | **Output** |  | **Input** | **Output** |  | **Input** | **Output** |
| 9  6  2  5  1  7 | (5, 1, 7) < (6, 2, 9) | 12  4  8  9  16  14 | (12, 4, 8) < (16, 9, 14)  (12, 4, 8) < (16, 14, 9)  (12, 4, 8) < (14, 9, 16)  (12, 4, 8) < (14, 16, 9) | 12  4  8  4  16  9 |  |

**Problem 2 – Sequence of K Numbers**

Write a program to **remove all sequences of k equal elements** from a sequence of integers. For example, if we have the sequence **3 3 3 8 8 2 5 1 7 7 7 4 4 4 4 3 4 4** and we remove all sequences of **k = 2** elements, we will obtain **3 2 5 1 7 3**. For **k = 3**, we will obtain the following result: **8 8 2 5 1 4 3 4 4**. For **k = 1**, the result will be empty.

**Input**

The input data comes from the console. It should consist of a **two lines**:

* The first line holds the **input numbers**, separated one from another by a **space**.
* The second line holds the number **k**.

The input data will always be valid and in the format described. There is no need to check it explicitly.

**Output**

The output should be printed on the console. It should consist of a **single line** holding the obtained sequence of numbers. Separate each number from the next number by a space.

**Constraints**

* The input sequence numbers are integers in the range [-1000 … 1000].
* The **count** of the input numbers is in the range [1 … 1000].
* The number **k** is integer in the range [1 … 1000].
* Time limit: 0.1 seconds.
* Memory limit: 16 MB.

**Examples**

|  |  |
| --- | --- |
| **Input** | **Output** |
| 3 3 3 8 8 2 5 1 7 7 7 4 4 4 4 3 4 4  2 | 3 2 5 1 7 3 |
| 3 3 3 8 8 2 5 1 7 7 7 4 4 4 4 3 4 4  3 | 8 8 2 5 1 4 3 4 4 |
| 3 3 3 8 8 2 5 1 7 7 7 4 4 4 4 3 4 4  4 | 3 3 3 8 8 2 5 1 7 7 7 3 4 4 |
| 1 1 100 1 1  2 | 100 |

**Problem 3 – Disk**

In geometry, a **disk** is the region in a plane bounded by a circle (it also **includes** the circle itself). Your task is to **print a disk on the console** by given **radius R** in a square **field of size** **N** x **N** (see the examples below).

**Input**

The input data should be read from the console.

* On the first line of the input you will be given the size of the field **N**. On the second line of the input you will be given the radius of the disk **R**.
* The disk’s center **is the center point** of the field (it will always exist, because N is odd).

The input data will always be valid and in the format described. There is no need to check it explicitly.

**Output**

The output should be printed on the console. You should print the disk on the console following the examples below.

* Your output must consist of **N** rows, each containing **N** characters. Each character represents a point in the field. For every point you must output one of two possible states – dot ‘**.**’ if the point lies outside of the disk and asterisk ‘**\***’ if the point lies within the disk.

**Hint:** In order to check whether a point is inside or outside of a circle, you may calculate the distance from the point to the center of the field by the **Pythagorean’s** **theorem** (see <http://goo.gl/HwqOuU>).

**Constraints**

* The number **N** is a positive **odd** integer in the range [3…39], inclusive.
* The number **R** is a positive integer between 1 and floorNby2 (floor (N/2)), inclusive. This means that the disk will always fit in the field, without crossing its sides.
* Allowed working time for your program: 0.1 seconds.
* Allowed memory: 16 MB.

**Examples**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Input** | **Output** |  | **Input** | **Output** |  | **Input** | **Output** |  | **Input** | **Output** |
| 5  2 | ..\*..  .\*\*\*.  \*\*\*\*\*  .\*\*\*.  ..\*.. | 9  3 | .........  ....\*....  ..\*\*\*\*\*..  ..\*\*\*\*\*..  .\*\*\*\*\*\*\*.  ..\*\*\*\*\*..  ..\*\*\*\*\*..  ....\*....  ......... | 11  1 | ...........  ...........  ...........  ...........  .....\*.....  ....\*\*\*....  .....\*.....  ...........  ...........  ...........  ........... | 19  6 | ...................  ...................  ...................  .........\*.........  ......\*\*\*\*\*\*\*......  .....\*\*\*\*\*\*\*\*\*.....  ....\*\*\*\*\*\*\*\*\*\*\*....  ....\*\*\*\*\*\*\*\*\*\*\*....  ....\*\*\*\*\*\*\*\*\*\*\*....  ...\*\*\*\*\*\*\*\*\*\*\*\*\*...  ....\*\*\*\*\*\*\*\*\*\*\*....  ....\*\*\*\*\*\*\*\*\*\*\*....  ....\*\*\*\*\*\*\*\*\*\*\*....  .....\*\*\*\*\*\*\*\*\*.....  ......\*\*\*\*\*\*\*......  .........\*.........  ...................  ...................  ................... |

**Problem 4 – Poker Straight**

The classical **playing cards** have **face** and **suit**. Card **faces** are: 2, 3, 4, 5, 6, 7, 8, 9, 10, J, Q, K, A. Cards **suits** are: Clubs (**C**), Diamonds (**D**), Hearts (**H**) and Spades (**S**). We denote a card in short as **card face + card suit letter**, e.g. **5C** (Five Clubs), **10S** (Ten Spades), **AH** (Ace Hearts), **2D** (Two Diamonds).

In some poker games, we have a hand called "**straight**" which consists of a **sequence of five cards of increasing sequential rank**. The card **ranks** are as follows: A, 2, 3, 4, 5, 6, 7, 8, 9, 10, J, Q, K, A. Note that the Ace (**A**) is a special card: it works both as the **smallest** and as the **biggest** card.

**Examples** of "straight": (AC 2H 3H 4C 5H), (2H 3S 4H 5H 6D), (5C 6C 7C 8C 9C), (7C 8D 9S 10S JS), (9C 10H JD QD KD), (10D JD QS KH AH). The following hands are not "straight": (9D JD QS KH AH) – not sequential; (10C 7C 9D 8S JS) – sequential, but not ordered; (9H 8S 7H 6H 5D) – sequential, but incorrectly ordered.

**Card faces** have the following **weights**: A 🡪 1 (as first card) or 14 (as last card); 2 🡪 2; 3 🡪 3; 4 🡪 4; 5 🡪 5; 6 🡪 6; 7 🡪 7; 8 🡪 8; 9 🡪 9; 10 🡪 10; J 🡪 11; Q 🡪 12; K 🡪13. **Card suits** have the following weights: Clubs 🡪 1; Diamonds 🡪 2; Hearts 🡪 3; Spades 🡪 4. **Hands weight** is calculated as follows:

10 \* weight(first card face) + weight(first card suit) +

20 \* weight(second card face) + weight(second card suit) +

30 \* weight(third card face) + weight(third card suit) +

40 \* weight(fourth card face) + weight(fourth card suit) +

50 \* weight(fifth card face) + weight(fifth card suit)

**Examples** of straight hands and their **weights**:

* weight(AC 2H 3H 4C 5S) = (10\*1 + 1) + (20\*2 + 3) + (30\*3 + 3) + (40\*4 + 1) + (50\*5 + 4) = 562
* weight(10H JS QD KS AD) = (10\*10 + 3) + (20\*11 + 4) + (30\*12 + 2) + (40\*13 + 4) + (50\*14 + 2) = 1915

Your task is to write a program that **calculates the number of straight hands that have given weight w**.

**Input**

The input data should be read from the console. It holds a single number **w** (the target weight). The input data will always be valid and in the format described. There is no need to check it explicitly.

**Output**

Print on the console the **number of straight hands of given weight** **w**.

**Constraints**

* The number **w** is an integer in the range [0…5000].
* Allowed time: 0.2 seconds. Allowed memory: 16 MB.

**Examples**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Input** | **Output** | **Hands** |  | **Input** | **Output** | **Hands** |
| 555 | 1 | (AC 2C 3C 4C 5C) | 1907 | 15 | (10C JC QC KC AH) (10C JC QC KD AD) (10C JC QC KH AC) (10C JC QD KC AD) (10C JC QD KD AC) (10C JC QH KC AC) (10C JD QC KC AD) (10C JD QC KD AC) (10C JD QD KC AC) (10C JH QC KC AC) (10D JC QC KC AD) (10D JC QC KD AC) (10D JC QD KC AC) (10D JD QC KC AC) (10H JC QC KC AC) |
| 200 | 0 | (no hands) |
| 1912 | 155 | (155 hands) |
| 856 | 5 | (3C 4C 5C 6C 7D)  (3C 4C 5C 6D 7C)  (3C 4C 5D 6C 7C)  (3C 4D 5C 6C 7C)  (3D 4C 5C 6C 7C) |

**Problem 5 – Friend Bits**

"**Friend bits**" are sequences of bits with the same value. All other bits are "**alone bits**". For example in the sequence **11101010011011010000111010110001** the friend bits (marked in grey, from left to right) are: **111**, **00**, **11**, **11**, **0000**, **111**, **11** and **000**. In the same sequence alone bits are: **0101**, **0**, **01**, **010** and **1**.

We are given a 32-bit integer **n**. Write a program that **extracts all friend bits and all alone bits from n**. Append all friend bits in the same order in which they appear in **n** and print their decimal representation **f**. Respectively, append all alone bits in the order in which they appear in **n** and print their decimal representation **a**.

**Input**

The input data should be read from the console. It holds the number **n** at the first line.

The input data will always be valid and in the format described. There is no need to check it explicitly.

**Output**

The output should be printed on the console. It should consist of exactly 2 lines: the numbers **f** and **a**.

**Constraints**

* All input and output numbers should be **unsigned 32-bit integers** in the range [0 to 4 294 967 295].
* Time limit: 0.1 seconds.
* Memory limit: 16 MB.

**Examples**

|  |  |  |
| --- | --- | --- |
| **Input** | **Output** | **Explanation** |
| 3933015729 | 1896696  661 | 3933015729 = **11101010011011010000111010110001**(2)  friend bits = **111 00 1111 0000 111 11 000(2)** = 1896696  alone bits = **0101** **0** **01** **010** **1**(2) = 661 |
| 3822674602 | 466840  2730 | 3822674602 = **1110001111**0**1100**10**11000**1010101010(2)  friend bits = 1110001111 1100 11000 (2) = 466840  alone bits = 0 **10** 1010101010(2) = 2730 |
| 21 | 0  21 | 21 = 10101(2)  friend bits = (empty) = 0  alone bits = 10101(2) = 21 |
| 15 | 15  0 | 15 = **1111**(2)  friend bits = 1111(2) = 15  alone bits = (empty) = 0 |