# More Exercises: Lists

Problems for exercises and homework for the [“Programming Fundamentals Extended” course @ SoftUni](https://softuni.bg/courses/programming-fundamentals).

You can check your solutions here: <https://judge.softuni.bg/Contests/425>.

## Distinct List

You will be given a list of **integers** on the **first line** of the input (space-separated). **Remove** all **repeating elements** from the list.

### Examples

|  |  |  |
| --- | --- | --- |
| **Input** | **Output** | **Comments** |
| 1 2 3 4 | 1 2 3 4 | No repeating elements |
| 7 8 9 7 2 3 4 1 2 | 7 8 9 2 3 4 1 | 7 and 2 are already present in the list 🡺remove them |
| 20 8 12 13 4 4 8 5 | 20 8 12 13 4 5 | 4 and 8 are already present in the list 🡺remove them |

## Integer Insertion

You will receive a list of **integers** on the same line (separated by **one** space). On the next lines, you will start receiving a list of **strings**, until you receive the string “end”. Your task is to insert each string (**converted** to integer) at a specific index in the list. The index is determined by the **first digit** of the **number**.

Example: 514 🡺 first digit – 5 🡺 insert 514 at the **5th** index of the list.

After you insert all the elements, print the list, separated by single spaces.

*The input will* ***always*** *be valid and you don’t need to explicitly check if you’re inserting an element into a valid index.*

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| 1 2 3 4 5 6 7 8 9  25  716  2772  1982  8534  2  end | 1 1982 2 2 2772 25 3 4 5 8534 6 716 7 8 9 |
| 3 12 66 243 8766  12  33  56  end | 3 12 12 33 66 56 243 8766 |
| 9 9 9 9 9 9 9 9 9 9  9  9  9  9  9  end | 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 |

## Camel’s Back

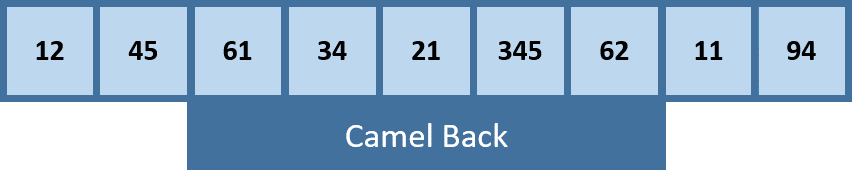
The city is breaking down on a camel back. You will receive a sequence of **N** **integers**, (space-separated), which will represent the **buildings** in the city. You will then receive an integer **M**, indicating the camel back's **size**.

The camel back is a **linear structure** standing **below** the city, in such a way that it has an **equal amount** of buildings to its **left** and **right**. The idea is, if every round – **one** building falls from the **left** **side** of the city, and **one** from the **right side**, how many **rounds** will it take for the city to stop breaking down?

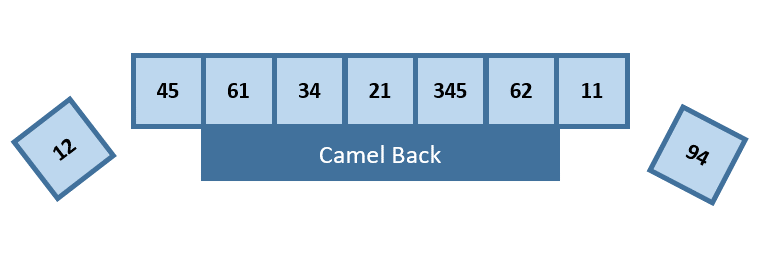
As output you must print how many rounds it took before the city stopped breaking down as “{rounds} rounds”. On the next line, print what’s left of the city (space-separated). Format: “remaining: {buildings (space-separated)}”

If no buildings have fallen, print “already stable: {buildings (space-separated)}”

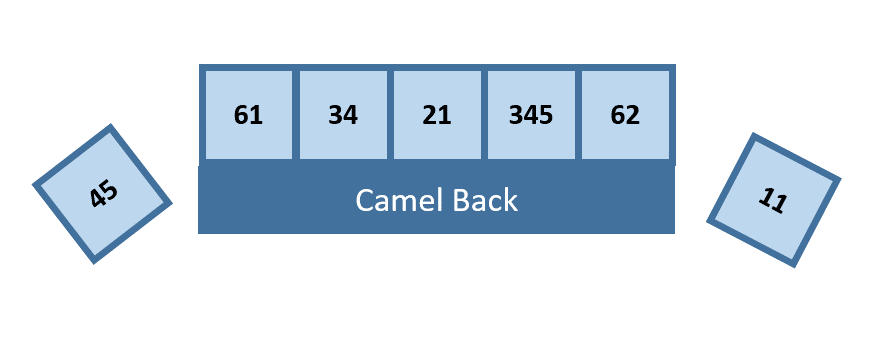
Example: city with **9 elements**; Camel back size: **5**



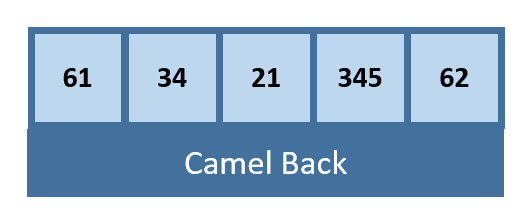
Round 1: first and last elements fall off:



Round 2: first and last elements fall off:



Remaining list:



### Constraints

**N** and **M** will always be **odd** numbers. [1-1000] and **M** will always be **smaller or equal** to **N**.

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| 12 45 61 34 21 345 62 11 94  5 | 2 rounds  remaining: 61 34 21 345 62 |
| 9 8 7 8 2  5 | already stable: 9 8 7 8 2 |
| 8 7 8 1 2 3 8 7 2 44 3 212 11  5 | 4 rounds  remaining: 2 3 8 7 2 |

## Ununion Lists

You will be given a sequence of integers, separated by a space. This is the **primal list**. Then you will receive an **integer N**. On the next **N lines**, you will receive sequences of integers. Your task is to add all elements that the primal list **does not** contain from the **current sequence** to the **primal list** and then **remove** from the primal list, all elements which **are** contained in the current sequence of integers. If there are several occurrences, remove **all** occurrences you find.

After you are done receiving lists and manipulating the primal list, **sort** the primal list and **print** it.

### Examples

|  |  |  |
| --- | --- | --- |
| **Input** | **Output** | **Comments** |
| 1 2 3 4 5  3  3 4 5 20 1  16 2 1  14 | 1 14 16 20 | Key:  🡺 remove from primal list  🡺 add to primal list  Primal: 1 2 3 4 5  List 1: 3 4 5 20 1  Primal after List 1: 2 20  List 2: 16 2 1  Primal after List 2: 20 16 1  List 3: 14  Primal after List 3: 20 16 1 14  Sorted primal list: 1 14 16 20 |
| 3 3 4 5 18 20  2  1 2 3  18 20 | 1 2 4 5 | Primal: 3 3 4 5 18 20  List 1: 1 2 3  Primal after List 1: 4 5 18 20 1 2  List 2: 18 20  Primal after List 2: 4 5 1 2  Sorted primal list: 1 2 4 5 |
| 99 88 77 99 66 55  5  99  88  77  66  14 11 12 13 55 | 11 12 13 14 | Primal: 99 88 77 99 66 55  List 1: 99  Primal after List 1: 88 77 66 55  List 2: 88  Primal after List 2: 77 66 55  List 3: 77  Primal after List 3: 66 55  List 4: 66  Primal after List 4: 55  List 5: 14 11 12 13 55  Primal after List 5: 14 11 12 13  Sorted primal list: 11 12 13 14 |

## \* Note Statistics

In music, certain frequencies correspond to certain musical notes (example: A -> 440hz, C# -> 554.37hz).

You will be given frequencies as **real numbers** on the first line of the input (space-separated). Your task is to **convert** the numbers to their **musical note representation**, then **separate** them into **naturals** (C, B, F and etc.) and **sharp** notes (C#, F#, A#, etc.). After that, print both lists in the console in the format described in the examples. After you print them, **sum** each list’s frequencies and **print** it on the console, rounded to the second decimal place.

### Constraints

The frequencies in the input will be constrained to this chart:

|  |  |
| --- | --- |
| **Note** | **Frequency** |
| C | 261.63hz |
| C# | 277.18hz |
| D | 293.66hz |
| D# | 311.13hz |
| E | 329.63hz |
| F | 349.23hz |
| F# | 369.99hz |
| G | 392.00hz |
| G# | 415.30hz |
| A | 440.00hz |
| A# | 466.16hz |
| B | 493.88hz |

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| 293.66 311.13 293.66 311.13 349.23 349.23 | Notes: D D# D D# F F  Naturals: D, D, F, F  Sharps: D#, D#  Naturals sum: 1285.78  Sharps sum: 622.26 |
| 329.63 329.63 329.63 261.63 329.63 392.0 | Notes: E E E C E G  Naturals: E, E, E, C, E, G  Sharps:  Naturals sum: 1972.15  Sharps sum: 0 |
| 392.0 440.0 349.23 392.0 466.16 440.0 349.23 392.0 | Notes: G A F G A# A F G  Naturals: G, A, F, G, A, F, G  Sharps: A#  Naturals sum: 2754.46  Sharps sum: 466.16 |

*Bonus: you can feed the list of frequencies into Console.Beep(freq, duration) to hear the tests* ☺

## \* Winecraft

You will be given a sequence of integers, which will represent grapes. On the next line, you will be given **N** - an **integer**, indicating the **growth days**. You must increment every integer in the list by 1 **N times**. However, if one of the grapes’ **value** is **greater** than the grape to its **left** and is **also** **greater** than the one to his **right**, it becomes **Greater grape**.

**The Greater grape** steals the values, which would have been incremented to its **neighbors**, and adds them to itself, instead of being incremented by **1** like **normal**. On top of that the grapes, which are **neighbors** of the Greater grape are decremented by **1** (note: if any of the **greater grapes’** neighboring grapes have a value of **0**, **DON’T** decrement it and **DON’T** add its **value** to the **greater grape**).

**Example**: If we the list **1 12 4**. The element at position 1 is greater grape, because it is **bigger** than the **elements** on the **left** and on the **right**:

* **First** iteration: The **Greater** grape increases with **1** by default and takes **2** from its **neighbors**. The new list look like: **0 15 3**
* **Second** iteration: The **Greater** grape increases with **1** by **default** and takes **only** **1** from its **neighbors**. This is because the grape on **left** is **0** and the **Greater** **grape** takes only from the **left** one. The list now looks like this: **0 16 2**

Lesser grapes **don’t** get **incremented** whenthey **have** as **neighbor Greater grape** , but instead they have their values **decremented** by **1** by their **neighboring** **Greater grapes** (if there are such), therefore their **values** get **added** to the Greater grapes.

After you're done with the **growing** (processed the grapes **N** times), every grape which has a **value**, **lower** than **N** should be **set to a value of 0 and** you should **not increment** them or **steal values** from them.

The process should then repeat, again incrementing everything **N** times, where the **Greater grapes** steal from the **lesser grapes**, until your list contains **less** than **N** grapes with **value** **more** than **N**.

After that, print the **remaining** grapes on the console (**one** **line**, **space-separated**).

### Examples

|  |  |  |
| --- | --- | --- |
| **Input** | **Output** | **Comments** |
| 4 4 15 4  3 | 7 24 | Key:  🡺 normal grapes  🡺 lesser grapes  🡺 greater grapes  Round 1: 5 3 18 3  Round 2: 6 2 21 2  Round 3: 7 1 24 1  Remove grapes less or equal to 3 🡺 1 and 1 |
| 10 11 12 13 19 13  5 | 20 35 44 | Round 1: 11 12 13 12 22 12  Round 2: 12 11 16 10 25 11  Round 3: 13 10 19 8 28 10  Round 4: 14 9 22 6 31 9  Round 5: 15 8 25 4 34 8  Remove grapes less or equal to 5 🡺 4  Round 1: 16 7 27 0 36 7  Round 2: 17 6 29 0 38 6  Round 3: 18 5 31 0 40 5  Round 4: 19 3 33 0 42 4  Round 5: 20 3 35 0 44 3  Remove grapes less or equal to 5 🡺 3 and 3 |
| 6 7 6 2  3 | 16 5 | Round 1: 5 10 5 3  Round 2: 4 13 4 4  Round 3: 3 16 3 5  Remove grapes less or equal to 3 🡺 3 and 3 |