Stacks and Queues – Lab

Write C++ code for solving the tasks on the following pages.

Code should compile under the C++03 or the C++11 standard.

I. Working with Stacks

1. Reverse Strings

Write a program that:

- **Reads** an **input string.**
- **Prints** the result back at the terminal in a reversed order. The words are reverted and the letters inside each word are reverted as well.

Examples

Input	Output
`I Love C++`	`++C evoL I`
`Stacks and Queues`	`seueuQ dna skcatS`

Hints

- Use a **`std::stack<T>.`**
- Use the methods **`push()`**, **`pop()`**.

2. Stack Sum

Write a program that:

- **Reads** an **input of integer numbers** and **adds** them to a **stack.**
- **Reads command** until **"end"** is received.
- **Prints** the **sum** of the remaining elements of the **stack.**

Input

- On the **first line,** you will receive **an array of integers.**
- On the **next lines**, until the "**end**" command is given, you will receive **commands** – a **single command** and **one** or **two** numbers after the **command, depending** on what **command** you are given.
- If the **command** is "**add**", you will **always** receive **exactly two** numbers after the command which you need to **add** to the **stack.**
- If the **command** is "**`remove`**", you will **always** receive **exactly one** number after the command which represents the **count** of the numbers you need to **remove** from the **stack.** If there are **not enough elements** skip the command.

Output

- When the **command** "**end**" is received, you need to print the sum of the remaining elements in the stack.

Examples

Input	Output
1 2 3 4 adD 5 6 REmove 3 end	Sum: 6
3 5 8 4 1 9 add 19 32 remove 10 add 89 22 remove 4 remove 3 end	Sum: 16

Hints

- Use a **std::stack<T>.**
- Use the methods **push()**, **pop()**.

3. Simple Calculator

Create a simple calculator that can **evaluate simple expressions** with only addition and subtraction. There will not be any parentheses.

Solve the problem **using a Stack**.

Examples

Input	Output
2 + 5 + 10 - 2 - 1	14
2 - 2 + 5	5

4. Matching Brackets

We are given an arithmetic expression with brackets. Scan through the string and extract each sub-expression.

Print the result back at the terminal.

Examples

Input	Output
1 + (2 - (2 + 3) * 4 / (3 + 1)) * 5	(2 + 3) (3 + 1) (2 - (2 + 3) * 4 / (3 + 1))
(2 + 3) - (2 + 3)	(2 + 3) (2 + 3)

Hints

- Scan through the expression searching for brackets.
- Use a **std::stack<T>.**

II. Working with Queues

5. Print Even Numbers

Write a program that:

- **Reads** an array of **integers** and **adds** them to a **queue.**
- **Prints** the **even** numbers **separated** by **", "**.

Examples

Input	Output
1 2 3 4 5 6	2, 4, 6
11 13 18 95 2 112 81 46	18, 2, 112, 46

Hints

- Use a **`std::queue<int>`**
- Use the methods **`push()`**, **`pop()`, `front()`**

6. Supermarket

Reads an **input** consisting of a **name** and **adds** it to a **queue** until "**End**" is received. If you receive "**`Paid`**", **print** every customer name and empty the queue, otherwise, we receive a client and we have to add him to the queue. When we receive **"End"** we have to print the count of the remaining people in the queue in the format: "**`{count} people remaining.`**".

Examples

Input	Output
Liam Noah James Paid Oliver Lucas Logan Tiana End	Liam Noah James 4 people remaining.
Amelia Thomas Elias End	3 people remaining.

7. Hot Potato

Hot potato is a game in which **children form a circle and start passing a hot potato**. The counting starts with the first kid. **Every nth toss the child left with the potato leaves the game**. When a kid leaves the game, it passes the potato along. This continues **until there is only one kid left**.

Create a program that simulates the game of Hot Potato. **Print every kid that is removed from the circle**. In the end, **print the kid that is left last**.

Examples

Input	Output
Alva James William 2	Removed James Removed Alva Last is William
Lucas Jacob Noah Logan Ethan 10	Removed Ethan Removed Jacob Removed Noah Removed Lucas Last is Logan
Carter Dylan Jack Luke Gabriel 1	Removed Carter Removed Dylan Removed Jack Removed Luke Last is Gabriel

Advanced Problems

1. Traffic Jam*

Create a program that simulates the **queue** that forms during a **traffic jam**. During a traffic jam, only **N** cars can **pass** the crossroads when the **light goes green**. Then the program reads the **vehicles** that **arrive** one by one and **adds** them to the **queue**. When the light **goes green N** number of cars **pass** the crossroads and **for each,** a **message** "**`{car} passed!`**" is displayed. When the "**end**" command is given, **terminate** the program and **display** a **message** with the **total number** of cars that **passed** the crossroads.

Input

- On the **first line,** you will receive **N** – the number of cars that can pass during a green light.
- On the **next lines**, until the "**end**" command is given, you will receive **commands** – a **single string**, either a **car** or "**green**".

Output

- Every time the "**green**" command is given, **print out** a message for **every car** that **passes** the crossroads in the format "**`{car} passed!`**"
- When the "**end**" command is given, **print out** a message in the format "**`{number of cars} cars passed the crossroads.`**"

Examples

Input	Output
4 Hummer H2 Audi Lada Tesla Renault Trabant Mercedes MAN Truck green	Hummer H2 passed! Audi passed! Lada passed! Tesla passed! Renault passed! Trabant passed! Mercedes passed! MAN Truck passed! 8 cars passed the crossroads.

green Tesla Renault Trabant end	
3 Enzo's car Jade's car Mercedes CLS Audi green BMW X5 green end	Enzo's car passed! Jade's car passed! Mercedes CLS passed! Audi passed! BMW X5 passed! 5 cars passed the crossroads.

2. Crossroads*

Our favorite super-spy action hero Sam is back from his mission in the previous exam, and he has finally found some time to go on a **holiday**. He is taking his wife somewhere nice and they're going to have a really good time, but first, they have to get there. Even on his holiday trip, Sam is still going to run into some **problems** and the first one is, of course, getting to the airport. Right now, he is stuck in a traffic jam at a **very active crossroads** where a lot of **accidents** happen.

Your job is to keep track of traffic at the crossroads and report whether a **crash happened** or everyone **passed** the **crossroads safely** and our hero is one step closer to a much-desired vacation.

The road Sam is on has a **single lane** where cars queue up until the **light goes green**. When it does, they start passing one by one during the **green light** and the **free window** before the **intersecting road's light** goes **green**. During **one second** only **one part** of a **car** (a **single character**) passes the crossroads. If a car is still at the crossroads when the **free window** ends, it will get hit by the **first character** that is still at the crossroads.

Input

- On the **first line**, you will receive the duration of the **green light** in seconds – an **integer in the range [1-100].**
- On the **second line**, you will receive the duration of the **free window** in seconds – an **integer in the range [0-100].**
- On the **following lines**, until you receive the "**END**" command, you will receive one of two things:
  - A **car** – a **string** containing any ASCII character, or
  - The command "**green**" indicates the **start** of a **green light cycle**

A **green light cycle** goes as follows:

- During the **green light,** cars will enter and exit the crossroads one by one
- During the **free window,** cars will only exit the crossroads

Output

- If a **crash happens**, **end the program** and print:
  "A crash happened!"
  "{car} was hit at {characterHit}."
- If everything **goes smoothly** and you receive an "**END**" command, print:
  "Everyone is safe."
  "{totalCarsPassed} total cars passed the crossroads."

Constraints

- The input will be **within the constraints** specified above and will **always be valid**. There is **no need** to check it explicitly.

Examples

Input	Output	Comments
10 5 Mercedes green Mercedes BMW Skoda green END	Everyone is safe. 3 total cars passed the crossroads.	During the first green light (**10** seconds), the **Mercedes** (**8**) passes safely. During the second green light, the **Mercedes** (**8**) passes safely and there are **2 seconds left**. The **BMW enters** the crossroads and when the green light ends, it still has **1 part** inside ('**W**') but has **5 seconds** to leave and passes successfully. The **Skoda never enters** the crossroads, so **3 cars passed successfully**.
9 3 Mercedes Hummer green Hummer Mercedes green END	A crash happened! Hummer was hit at e.	Mercedes (**8**) passes successfully and Hummer (**6**) enters the crossroads but only the 'H' passes during the green light. There are **3** seconds of a free window, so "umm" passes and the Hummer gets hit at 'e' and the program ends with a **crash**.

3. Key Revolver*

Our favorite super-spy action hero Sam is back from his mission in another exam, and this time he has an even more difficult task. He needs to **unlock a safe**. The problem is that the safe is **locked** by **several locks in a row**, which all have **varying sizes**.

Our hero possesses a special weapon though, called the **Key Revolver**, with special bullets. Each **bullet** can unlock a **lock** with a **size equal to or larger than** the **size** of the **bullet**. The bullet goes into the keyhole, then explodes, completely **destroying** it. Sam **doesn't know the size** of the locks, so he needs to just shoot at all of them until the safe run out of locks.

What's behind the safe, you ask? Well, intelligence! It is told that Sam's sworn enemy – **Nikoladze**, keeps his **top-secret Georgian Chacha Brandy** recipe inside. It's valued differently across different times of the year, so Sam's boss will tell him what it's worth over the radio. One last thing, every bullet Sam fires will also cost him money, w**hich will be deducted from his pay** from the price of the intelligence.

Good luck, operative.

Input

- On the **first line** of input, you will receive the price of each **bullet** – an **integer in the range [0-100].**
- On the **second line**, you will receive the **size of the gun barrel** – an **integer in the range [1-5000].**

- On the **third line**, you will receive the **bullets** – a **space-separated integer sequence** with **[1-100] integers.**
- On the **fourth line**, you will receive the **locks** – a **space-separated integer sequence** with **[1-100] integers.**
- On the **fifth line**, you will receive the **value of the intelligence** – an **integer in the range [1-100000].**

After Sam receives all of his information and gear (**input**), he starts to **shoot the locks front-to-back**, while going through the bullets **back-to-front**.

If the **bullet** has a **smaller or equal** size to the **current lock**, print **"Bang!"**, then **remove the lock**. If not, print **"Ping!"**, leaving the lock **intact**. The bullet is removed in **both cases**.

If Sam runs out of bullets in his barrel, print **"Reloading!"** on the console, then continue shooting. If there aren't any bullets left, **don't** print it.

The program ends when Sam **either runs out of bullets**, or the safe **runs out of locks**.

Output

- If Sam **runs out of bullets** before the safe runs out of **locks**, print:
  "Couldn't get through. Locks left: {locksLeft}"
- If Sam manages to **open the safe**, print:
  "{bulletsLeft} bullets left. Earned ${moneyEarned}"

Make sure to account for the **price of the bullets** when calculating the **money earned**.

Constraints

- The input will be **within the constraints** specified above and will **always be valid**. There is **no need** to check it explicitly.
- There will **never** be a case where Sam breaks the lock and ends up with a **negative balance**.

Examples

Input	Output	Comments
50 2 11 10 5 11 10 20 15 13 16 1500	Ping! Bang! Reloading! Bang! Bang! Reloading! 2 bullets left. Earned $1300	**20** shoots lock **15** (**ping**) **10** shoots lock **15** (**bang**) **11** shoots lock **13** (**bang**) **5** shoots lock **16** (**bang**) Bullet cost: 4 * 50 = $200 Earned: 1500 – 200 = $1300
20 6 14 13 12 11 10 5 13 3 11 10 800	Bang! Ping! Ping! Ping! Ping! Ping! Couldn't get through. Locks left: 3	**5** shoots lock **13** (**bang**) **10** shoots lock **3** (**ping**) **11** shoots lock **3** (**ping**) **12** shoots lock **3** (**ping**) **13** shoots lock **3** (**ping**) **14** shoots lock **3** (**ping**)
33 1 12 11 10 10 20 30	Bang! Reloading! Bang! Reloading!	**10** shoots lock **10** (**bang**) **11** shoots lock **20** (**bang**) **12** shoots lock **30** (**bang**)

100	Bang! 0 bullets left. Earned $1	Bullet cost: 3 * 33 = $99 Earned: 100 – 99 = $1

4. Cups and Bottles*

You will be given a **sequence of integers** – each indicating a **cup's capacity**. After that, you will be given **another sequence of integers** – a **bottle with water** in it. Your job is to try to **fill up** all of the cups.

The filling is done by picking **exactly one** bottle at a time. You must start picking from **the last received bottle** and start filling from **the first entered cup**. If the current bottle has **N** water, you **give** the **first entered cup N** water and **reduce** its integer value by **N**.

When a cup's **integer value** reaches **0 or less**, it **gets removed**. The current cup's value may be **greater** than the current bottle's value. **In that case,** you **pick bottles until** you reduce the cup's integer value to **0 or less**. If a bottle's value is **greater or equal to** the cup's **current** value, you fill up the cup, and **the remaining water becomes wasted**. You should **keep track of the wasted litters of water** and **print it at the end of the program**.

If you **have managed** to **fill up all of the cups**, print the **remaining water bottles**, from the **last entered – to the first**, otherwise you must print the **remaining cups**, by **order of entrance** – from the **first entered – to the last**.

Input

- On the **first line** of input, you will receive the integers, representing the **cups' capacity**, **separated** by a **single space**.
- On the **second line** of input, you will receive the integers, representing the **filled bottles**, **separated** by a **single space**.

Output

- On the first line of output, you must print the remaining bottles, or the remaining cups, depending on the case you are in. Just **keep** the **orders of printing exactly as specified**.
  - **`"Bottles: {remainingBottles}"`** or **`"Cups: {remainingCups}"`**
- On the second line print the wasted litters of water in the following format: **`"Wasted litters of water: {wastedLittersOfWater}".`**

Constraints

- All of the given numbers will be valid integers in the range [1, 500].
- It is safe to assume that there will be **NO** case in which the water is **exactly as much** as the cups' values so that in the end there are no cups and no water in the bottles.
- Allowed time/memory: 100ms/16MB.

Examples

Input	Output	Comment
4 2 10 5 3 15 15 11 6	Bottles: 3 Wasted litters of water: 26	**We take the first entered cup and the last entered bottle, as it is described in the condition.** **6 – 4 = 2 – we have 2 more so the wasted water becomes 2.** **11 – 2 = 9 –** again, it is more, so we add it to the previous amount, which is 2 and it becomes 11.

		15 – 10 = 5 – wasted water becomes 16. **15 – 5 = 10 – wasted water becomes 26.** **We've managed to fill up all of the cups, so we print the remaining bottles and the total amount of wasted water.**
1 5 28 1 4 3 18 1 9 30 4 5	Cups: 4 Wasted litters of water: 35	
10 20 30 40 50 20 11	Cups: 30 40 50 Wasted litters of water: 1	