C++ Fundamentals: Exam 1

The following tasks should be submitted to the SoftUni Judge system, which will be open starting **Sunday, 15 July 2018, 09:00** (in the morning) and will close on **Sunday, 15 July 2018, 15:00**. Submit your solutions here: https://judge.softuni.bg/Contests/Compete/Index/1113.

For this exam, the code for each task should be a single C++ file, the contents of which you copy-paste into the Judge system.

Please be mindful of the strict input and output requirements for each task, as well as any additional requirements on running time, used memory, etc., as the tasks are evaluated automatically and not following the requirements strictly may result in your program's output being evaluated as incorrect, even if the program's logic is mostly correct.

You can use C++03 and C++11 features in your code.

Unless explicitly stated, any integer input fits into **`int`** and any floating-point input can be stored in **`double`**. On the Judge system, a C++ **`int`** is a **`32-bit`** signed integer and a C++ **`double`** is a **`64-bit`** IEEE754 floating point number.

NOTE: the tasks here are NOT ordered by difficulty level.

Task 4 – Antimatter (Exam-1-Task-4-Antimatter)

*Short description: write a program that reads matter/antimatter particles with a 2D position, speed and lifetime. Read number of steps to simulate. On each step: move, reduce lifetime, remove particles outside of coordinates **0** to **65535**, as well as particles with **0** remaining lifetime, detect collisions (two or more particles of different types at the same coordinates) and destroy all at those coordinates. Print the number of remaining matter and antimatter particles, as well as the number of particles destroyed.*

The scientists in an imaginary physics lab (doing imaginary physics) are doing some experiments with matter and antimatter particles and have sensors monitoring a fixed-size area. Each experiment consists of creating a certain amount of **matter** and **antimatter** particles – each of which has a certain **position**, a certain **speed**, and a **lifetime** – and monitoring the interactions between those particles for a certain amount of time. After that amount of time has passed, the scientists count the **remaining matter** particles, the **remaining antimatter** particles, and the **number of particles destroyed** in **collisions between matter and antimatter** particles.

Unfortunately, the equipment and operating costs for conducting these experiments is high, so the scientists have turned to you – they want you to write software that **simulates the experiments** they are doing. The software has to be able to **read information about particles** (matter/antimatter, position, speed per unit of time, lifetime), then a positive integer number representing the **units of time** – aka steps – the simulation has to run. Then the software has to "**simulate**" what happens in the described physical system and – after all the simulation steps have passed – **print the number of remaining matter and antimatter particles**, as well as the **number of particles destroyed in collisions**.

Here's a list of rules for the simulation:

- The simulation runs in a 2-dimensional (2D) coordinates system with integer coordinates from **0** to **65535**. If a particle **exits this area**, it is **no longer counted in the "remaining" particles**, nor does it participate in collisions.
- Each position is a pair of integer values **`(x, y)`** – coordinates in the 2D coordinate system.
- The speed of a particle is also a pair of integer values **`(xSpeed, ySpeed)`**. Each "step", the position of a particles changes by making it's **`x = x + xSpeed`** and **`y = y + ySpeed`**. In other words, **xSpeed** and **ySpeed** are the change in coordinates per unit of **time**.
- Two or more particles can arrive at the **same position** at some point of the simulation.
  If they are all the **same type** – i.e. **all matter or all antimatter** – nothing special happens.
  If they are of **different types** – i.e. we have **matter and antimatter at the same position** – a **collision** happens and all particles at that position are **annihilated/destroyed**.
  - NOTE: it is completely possible for two particles to travel towards each other but "miss" – for example if we have particle **A** at **`(2, 0)`** with a speed of **`(1, 0)`** and particle **B** at **`(3, 0)`** with a speed of **`(-1, 0)`**, on the next step they will "switch" places and not collide (**A** arrives at **`(3,0)`** and **B** arrives at **`(2,0)`**). However, if A was at **`(1, 0)`**, then on the next step they would collide at the position **`(2, 0)`** – assuming they are of different types. That is, a collision can happen only if two particles appear in the exact same position at the exact same time (and the particles are of different types).
- Each step, the remaining **lifetime** of a particle is reduced by **1**. If a particle's lifetime reaches **0**, it **decays/disappears** immediately. So, the **lifetime** indicates **how many units of time** the particle will **exist** before **decaying**. Decayed particles **do not participate in collisions**, nor in the final count of remaining particles – the same way that particles that have exited the **area** do not participate in collisions or in the final count.

- For each step of the simulation:
    - First, the **positions and lifetimes of particles are updated**. Any **decayed** or **exited** particles are **removed**.
    - Then, **collisions** are detected, and any **destroyed** particles are **removed**.
- **After** the **last step** of the simulation – when the specified **units of time** have passed – the **remaining matter and antimatter** particles are **counted** and the result is printed, along with the **number of particles destroyed** in collisions

Write a program that does the above-described simulation.

Input

On the first line of the standard input, a single positive integer number will be entered – the number of particles.

On the following lines, the data for each particle is given in the following format:

```
type positionX positionY speedX speedY lifetime
```

Where:

- **`type`** is a single character, which will be either **`'a'`** or **`'m'`** – respectively indicating whether the particle is **antimatter** or **matter**
- **`positionX`** and **`positionY`** are non-negative integer numbers between **`0`** and **`65535`** (inclusive) and indicate the position of the particle in the 2D coordinate system
- **`speedX`** and **`speedY`** are non-negative integer numbers between **`0`** and **`10`** (inclusive) and indicate the speed of the particle
- **`lifetime`** is a non-negative integer number indicating the remaining units of time the particle will be in the simulation

On the last line of the standard input, there will be a single positive integer number – the units of time/steps the simulation has to be run for.

Output

On the first line of the standard output, print the number of remaining matter particles, followed by the number of remaining antimatter particles, separated by a single space.

On the second line of the standard output, print the number of particles destroyed/annihilated in collisions.

Restrictions

The input will contain no more than **`1500`** particles.

The total time will be no more than **`1000`**.

The maximum particle lifetime will be no more than **`2000`**.

No two particles will be at the same positions in the input. No particle will have invalid coordinates or be outside the described area in the input.

The minimum valid coordinates are **`0 0`** (lower left), the maximum valid coordinates are **`65535 65535`** (upper right). That is, valid coordinates are those for which **`0 <= x <= 65535 && 0 <= y <= 65535`**. Any particle outside this square after the time has elapsed is not counted toward the total matter/antimatter remaining and no collisions are counted for any particle once it leaves these coordinates.

The total running time of your program should be no more than **`0.6s`**

The total memory allowed for use by your program is **`4MB`**

Example I/O

Example Input	Expected Output
2 a 0 0 1 0 10 m 4 0 -1 0 10 4	0 0 2
2 a 1 0 1 0 10 m 4 0 -1 0 10 4	1 1 0
3 a 0 0 1 0 10 a 0 0 1 0 7 m 4 0 -1 0 10 4	0 0 3
3 a 0 0 1 0 10 m 4 0 -1 0 10 a 3 1 0 -1 7 4	0 1 2
4 a 0 0 1 0 10 m 4 0 -1 0 1 a 3 1 0 -1 1 a 65535 0 1 0 10 4	0 1 0
2 a 2 0 1 0 10 m 4 0 -1 0 10 1	0 0 2
2 a 2 0 1 0 1 m 4 0 -1 0 1 1	0 0 0