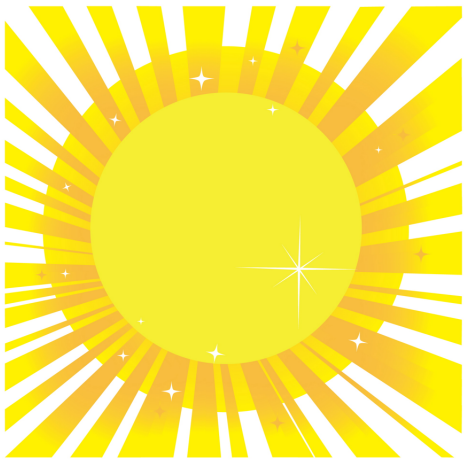
0001001110011000111110100100110
0100100010000001101000110100001
1010010101110100010110000000100
COMPUTER AND INFORMATION SCIENCE
1101000011100110110010000100100
0001000110100010001000000010011
1010001101010001010100100100011
0010010001001111000011100010101

UNIVERSITY OF OREGON

1110001001000011001110011010011
0011100001110100000100101010010

Twenty-second Annual University of Oregon Eugene Luks Programming Competition

2018 March 10, Saturday
am10:00 – pm2:00



Problem Contributors
Ramakrishnan Durairajan, Eugene Luks, Joe Sventek, Chris Wilson

Technical Assistance and Organization
Lauradel Collins, Anton Matschek

A

ANALYZE WEB USAGE

Each Web server routinely logs accesses from other Web servers and browsers. The log is a sequence of text lines; each line contains a date and a hostname, separated by a blank. The date is in mm/dd/yyyy format, and the hostname is in a.b.c.d format. The final token in a hostname is usually called the *top level domain*, or TLD for short. The TLD can be a 2-letter country code, such as *uk* or *fr*, a 3-letter code such as *com* or *edu*, or an even larger number of letters, such as *info*.

A new NSA regulation requires that we be able to track access to our web sites by country, being able to demonstrate the percentage of accesses from each country for a given period. Since there is no correlation between country of origin and TLD's like *com* and *edu*, the politicians have allowed that tracking accesses by TLD is sufficient to satisfy the regulation.

Given a start date, and end date, and the log entries, you are to write a program that determines the percentage of access from each TLD during that period (**including** the start and end dates). The output from your program should be in sorted order by TLD - i.e., *com* must come before *edu*.

Hostnames, and therefore, top level domain names, are case-insensitive. Therefore, accesses by X.Y.UK and a.b.uk are both accesses from the same TLD. When you output your results, your TLDs should be in lower case.

The input file is read from standard input, and is formatted as follows:

```
START_DATE END_DATE
N
LOG_LINE
...
```

`START_DATE` and `END_DATE` are each of the form mm/dd/yyyy. The second line of the input file will be an integer $0 \le N \le 10^6$ giving the number of log entries. Following will be N lines, each containing a log entry in the following format:

```
ACCESS_DATE HOST_NAME
```

`ACCESS_DATE` is in the form mm/dd/yyyy, and `HOST_NAME` is in the form a.b.c.d. The separator is a single space.

Dates in the range 01/01/1990 to 12/31/2017 are legal for start and end dates.

Note that the log entries are not sorted by date, as the file is typically created through the merge of several log files from several servers, which then has the begin and end dates and number of lines inserted at the beginning. Also, as you can see, you are to output the percentage at the beginning of each output line, in a field 5 characters wide, with 2 decimal points. If the percentage is <10%, there should be a leading blank - i.e., `' 0.25'`, not `'0.25'`. The percentage is followed by a single blank and then the TLD.

Sample log file

```
07/01/2015 07/01/2016
19
01/05/2016 host0013.follett.com
12/08/2015 pilspetsen.it.uu.SE
03/27/2014 proxy.dunaweb.hu
10/02/2015 cache4.lankacom.net
10/14/2015 msnbot.msn.COM
02/06/2015 csr043.goo.ne.jp
04/28/2014 host1.abports.co.UK
03/27/2016 b303a14h5.gcal.ac.uk
12/20/2016 kwgt025.gre.ac.Uk
12/20/2016 msnbot.msn.CoM
05/05/2015 zaaijer.xs4all.nl
07/27/2015 pcd49.li.man.ac.uK
12/16/2016 egspd42441.teoma.cOm
08/16/2014 acc9687e.ipt.aol.cOM
11/07/2016 msnbot.msn.COm
08/12/2016 ib216m06.cs.unb.ca
11/11/2014 msnbot.msn.com
12/20/2016 p-1.134.eunet.yu
01/05/2016 msnbot.msn.com
```

Sample output

```
42.86 com
14.29 net
14.29 se
28.57 uk
```

B

GENERATE IP ADDRESSES

ONRG conducts cutting-edge Internet measurements research. One of the classical problems in Internet measurement research is generating IP address targets for tools like traceroute and ping. Basically, the idea is to generate a valid IP address in the form of A.B.C.D, where A, B, C, and D are numbers from 0-255. The numbers cannot be 0 prefixed unless they are 0.

The target generation problem is challenging due to the fact that the seeds used are typically strings collected from e.g. webpages and does not follow the aforementioned formatting. Specifically, a seed collected from abc.com can be "25011255255" and can have three possibilities:

- 25.011.255.255 is not valid as 011 is not valid.
- 25.11.255.255 is not valid either as you are not allowed to change the string.
- 250.11.255.255 is valid.

The input will start with a number N (N≤100), followed by N lines, each containing a single seed containing only digits. For each given seed, generate all possible valid IP addresses. If the seed has no valid addresses, print "None". The list of addresses for a seed should be on a single line, separated by commas, in ascending order of digits. For instance, 255.255.11.135 should precede 255.255.111.35 since 11 is before 111.

Sample Input

```
4
25525511135
25505011535
25011255255
010011135
```

Sample Output

```
255.255.11.135, 255.255.111.35
None
250.11.255.255, 250.112.55.255
0.100.11.135, 0.100.111.35
```

C

Semigroup Sudoku

As in traditional Sudoku, a *Semigroup*-Sudoku puzzle requires completion of 9×9 arrays where each entry is an integer between 1 and 9. However, entries are subject to a different sort of constraint.

A semigroup consists of a set $\mathcal{S}$ together with an *associative* binary operation $\circ$ on $\mathcal{S}$; that is, for any i, j, k in $\mathcal{S}$, $(i \circ j) \circ k = i \circ (j \circ k)$. We now fix the set $\mathcal{S} := \{1,2,3,4,5,6,7,8,9\}$ and consider various operations that make it into a semigroup.

A binary operation on $\mathcal{S}$ can be specified by a 9×9 array where the entry in the i^{th} row and j^{th} column is $i \circ j$. Each of the following arrays defines a semigroup.

1	2	3	4	5	6	7	8	9
2	2	3	4	5	6	7	8	9
3	3	3	4	5	6	7	8	9
4	4	4	4	5	6	7	8	9
5	5	5	5	5	6	7	8	9
6	6	6	6	6	6	7	8	9
7	7	7	7	7	7	7	8	9
8	8	8	8	8	8	8	8	9
9	9	9	9	9	9	9	9	9

1	1	1	1	1	1	1	1	1
2	2	2	2	2	2	2	2	2
3	3	3	3	3	3	3	3	3
4	4	4	4	4	4	4	4	4
5	5	5	5	5	5	5	5	5
6	6	6	6	6	6	6	6	6
7	7	7	7	7	7	7	7	7
8	8	8	8	8	8	8	8	8
9	9	9	9	9	9	9	9	9

7	8	1	6	1	8	1	6	8
7	8	2	2	2	8	1	6	8
1	2	5	4	5	6	7	8	9
1	2	4	4	4	6	7	8	9
1	2	5	4	5	6	7	8	9
7	8	6	6	6	8	1	6	8
1	6	7	8	7	6	7	8	6
1	6	8	8	8	6	7	8	6
7	8	9	6	9	8	1	6	8

In the first one, $i \circ j = \max(i, j)$, and in the second, $i \circ j = i$. A rule for the third operation is less obvious but here is an associativity illustration:

$$6 \circ (7 \circ 2) \;=\; 6 \circ 6 \;=\; 8 \;=\; 1 \circ 2 \;=\; (6 \circ 7) \circ 2.$$

In this problem, you are given 9×9 arrays that resulted from attempts to fill in blanks in semigroup-sudoku puzzles. In each case, a *single place* had been filled incorrectly so that the binary operation is not associative. You are to locate that correctible error. For the arrays you will encounter, there will be only one way to create a semigroup with a single change.

The first line of the input will declare the number N of nearly-correct semigroup instances to be considered. The second line will be blank. There will also be a blank line between successive instances.

Each instance will be given over 9 consecutive lines, with each line consisting of 9 integers between 1 and 9. For each instance, you should indicate the position of the error and the correct fill. The three numbers

row, column, correct-entry

should be entered on a single line with one-space separations.

See reverse for sample I/O.

Sample Input

```
3

1 2 3 4 5 6 7 8 9
2 3 4 5 6 7 8 9 1
3 4 5 6 7 8 9 1 2
4 5 6 7 8 9 1 2 3
5 6 7 8 9 1 2 3 4
6 7 8 9 1 2 3 4 5
7 8 9 1 2 3 4 5 6
8 9 1 2 3 4 5 6 7
9 1 2 3 3 5 6 7 8

8 8 8 8 8 8 8 8 8
8 8 8 8 8 8 8 8 8
8 8 8 8 8 8 8 8 8
8 8 8 8 8 8 8 8 8
8 8 8 8 8 8 8 8 8
8 8 8 8 8 8 8 8 8
8 8 8 8 8 8 8 8 8
4 8 8 8 8 8 8 8 8
8 4 8 8 8 8 8 8 8

1 2 2 4 1 2 2 1 4
2 2 2 4 2 2 2 2 4
2 2 3 4 2 3 2 3 4
2 2 4 4 2 4 2 4 4
1 2 2 4 5 2 2 5 4
2 2 3 4 2 6 2 6 4
2 2 2 4 7 2 7 7 4
1 2 3 4 5 6 2 8 4
2 2 4 4 2 4 7 4 9
```

Sample Output

```
9 5 4
8 1 8
7 7 2
```

In each case, this is the *only* way to achieve associativity with a *single change*.

PRIORITIZING REQUESTS

You are reviewing log data from a server application with the goal of optimizing the order in which the application processes incoming requests. You want to determine the order in which the requests could have been processed that would have made the average response time as small as possible.

The log data has been reduced down to just the essential information: when each request arrived and, once scheduled, how long it took to process it. Once scheduled, a request cannot be preempted. You need to determine the optimal order for processing the requests and output the minimum average response time.

The first line of the input file will be an integer $0 \le N \le 10^5$ giving the number of requests. Following will be N lines, each containing a request string having the following format:

`ARRIVAL_TIME SERVICE_TIME`

$0 \le$ `ARRIVAL TIME, SERVICE TIME` $\le 10^9$ are integers representing the arrival time and service time, respectively. The separator is a single space. The log might be a little bit jumbled, so do not rely on the input list being in order of arrival time.

Assume that the server clock starts at 0 and it cannot process a request having arrival time A until the server clock is at least A. That is, it cannot process requests that have not yet arrived. Furthermore, if at any moment there are requests waiting to be served, it must serve one of them. It cannot stall, hoping for better requests.

Output the minimum average response time for the given batch of requests. Truncate your answer to its integer part.

Sample Input

```
4
0 2
4 5
1 10
3 1
```

Sample Output

```
9
```

Explanation: At server time 0, a request is available to be processed with service time 2. The total response time for the first request is 2. When the server finishes with this request, the server time is 2 and there is one request available to be processed with service time 10. Since this second request had to wait a duration of 1 and the service time is 10, the total response time

is 11. When the server finishes processing the second request, the server time is 12 and there are two requests waiting.

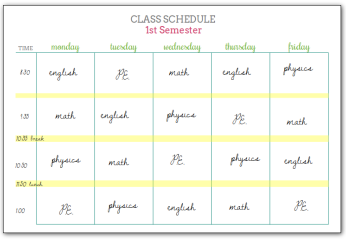
On one hand, suppose the server first processes the request that arrived at time 3 and has service time 1. With a wait time of 9 and a service time of 1, the response time for the third request is 10. Finally, the server clock is 13 and the server processes the final request with a response time of 14. For this service order the average response time is (2 + 11 + 10 + 14)/4 = 9.25.

On the other hand, suppose the server first processes the request that arrived at time 4 and has service time 5. With a wait time of 8 and a service time of 5, the response time for the third request is 13. Finally, the server clock is 17 and the server processes the final request with a response time of 15. For this alternate service order, the average response time is (2 + 11 + 13 + 15)/4 = 10.25.

Therefore, the minimum average response time is 9.25 and we output the integer part, 9.

E

SCHEDULING CLASSES



The CIS Department at the University of Oregon has many classes they would like to schedule in their new about-to-be constructed room 471 DES, but this cannot be done due to the demand for this special new space full of smart AI-enabled utilities. Instead it wants to place classes there in a way that maximizes its utilization, and you have been asked to write a program to help it do so. You will have a list of class schedules and estimated enrollments. Knowing that two classes cannot be scheduled in the room at the same time, your program should determine the maximum number of students who could be scheduled in the room by a valid schedule.

Suppose we indicate the class data as *(start,finish,size)*, where *start*, *finish,* and *size* are integers. We allow that one class may start immediately after one class ends (so the finish time of one class can be equal to the start time of the next). For example:

- If the class info is (1,2,10), (3,4,10), (1,3,20), (2,3,20), then the best answer is **40**. This is because we can validly schedule classes at times (1,2), (2,3), (3,4) whose sizes are 10+20+10=40.
- If the class info is (0,3,1), (8,11,1), (0,10,999), then pick only the long class, so the answer is **999**.

Note that here, for simplicity, you need only consider one day of classes, and the possible start/end times are 0, 1, 2, etc.

The input will start with a number C, $C \leq 20$, of test cases. Each test case will start with an integer N, $1 \leq N \leq 5000$, the number of classes for this case. This is followed by N lines of the form S F Z, where S and F are integers, $0 \leq S < F \leq 5000$, with S indicating the start time of a class and F the finish time. You may **not** assume that the list of classes is sorted by either the start time S or the finish time F, but you can assume that S is strictly earlier than F (that is, $S<F$). The estimated class size $Z>0$ is also an integer and you may assume that Z students can fit in this room.

The output needs to be a single integer M for each test case on its own line. This number M is the largest possible total number of students attending classes that could be scheduled in this room without overlap.

See reverse for sample I/O.

Sample Input

```
4
4
1 2 10
2 3 20
3 4 10
1 3 20
6
1 4 5
3 5 5
2 6 5
2 4 5
3 7 5
4 5 5
3
0 3 1
8 11 1
0 10 999
3
1 3 6
3 5 6
2 4 10
```

Sample Output

```
40
10
999
12
```

F

Two-Time Pad

In cryptography, a *one-time pad* (OTP) is a long stream of random letters. Guided by the OTP, a *plaintext* message is transformed letter-by-letter to produce the *ciphertext*. An authorized receiver in possession of a duplicate OTP will be able to reverse the process. On the other hand, the ciphertext will appear random for eavesdroppers, so that, when used properly, OTP encryption is *unbreakable*.

This programming exercise was inspired by an infamous misuse of a one-time pad

Encryption. We first assign the first 26 natural numbers to the letters of the alphabet (e.g., subtract 65 from the ASCII code of the letter):

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25

Then, to encrypt a plaintext letter by the corresponding one-time pad letter, add their numbers mod 26. For example, an OTP starting `HCYS` encrypts `DUCK` to `KWAC`, since

H		D		K		C		U		W		Y		C		A		S		K		C
7	+	3	≡	10 ,		2	+	20	≡	22 ,		24	+	2	≡	0 ,		18	+	10	≡	2 .

Our scenario. You are eavesdropping on the secret correspondence of your adversaries. You know that they use a one-time pad and also that, before encryption, plaintext messages are typically prefixed by a single letter that identifies the sender.

On this day, concerned about static on her line, the intended receiver asked the sender to repeat his message. Since his identity was already acknowledged, the sender figured he could drop the identity-letter prefix in that second transmission. Furthermore, he thought he could avoid a waste of random letters and restart his enciphering at the original point in the OTP.

You have intercepted both the original and the repeated ciphertext. You also have independently established the identity of the sender through direction-finding. Armed with that information, you must decipher the message.

The problem. You are given the initial ciphertext and the resent ciphertext. These will be contiguous strings of length ≤ 50 and comprised of upper-case letters (no spaces). Both encryptions would have started at the same place in the one-time pad, but the second dropped the initial letter of plaintext before encrypting. You also know that initial letter (the sender's identifier).

The input is given over three lines, the first cipher text followed by the second ciphertext, then followed by the sender's ID.

You should output the plaintext for the second transmission.

See reverse for sample I/O.

Sample Input

```
YZDGYBKTLYRFQ
BSDNAJAMVVNW
X
```

Sample Output

```
ATTACKATDAWN
```

Comments.

The sample input was created with a one-time pad that began

BZKNYZATSVRJD $\cdots$.

The encryptions were

BZKNYZATSVRJD	1	25	10	13	24	25	0	19	18	21	17	9	3
XATTACKATDAWN	23	0	19	19	0	2	10	0	19	3	0	22	13
YZDGYBKTLYRFQ	24	25	3	6	24	1	10	19	11	24	17	5	16

BZKNYZATSVRJ	1	25	10	13	24	25	0	19	18	21	17	9
ATTACKATDAWN	0	19	19	0	2	10	0	19	3	0	22	13
BSDNAJAMVVNW	1	18	3	13	0	9	0	12	21	21	13	22

The sample input is shown in red.