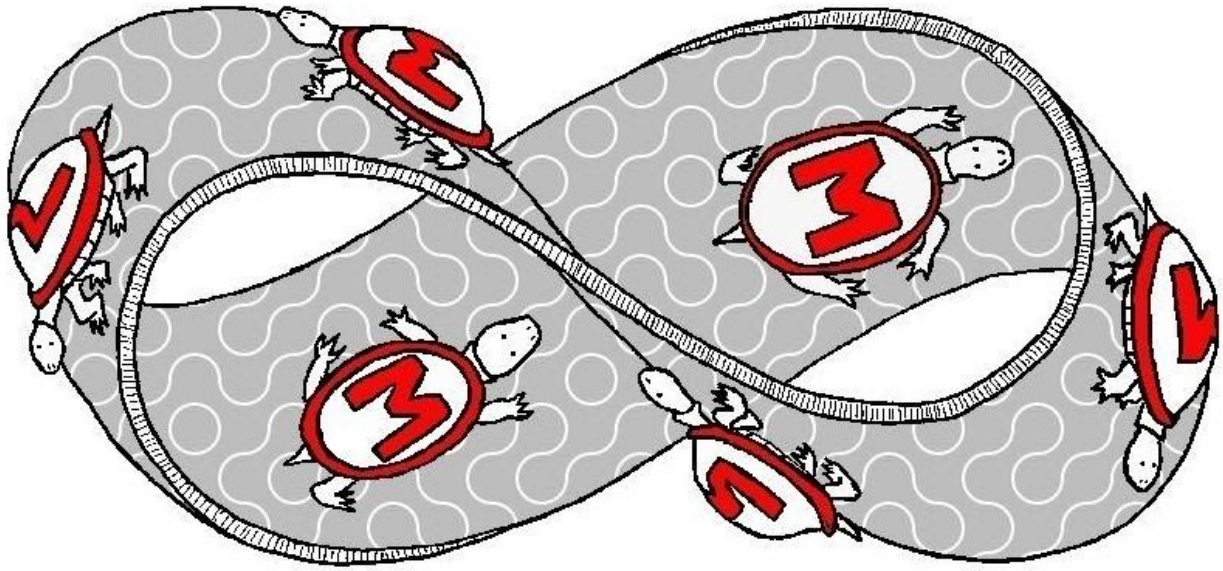
2020 University of Maryland High School Programming Contest



1. Fundraising with Angela (Part I)

Angela has decided to organize an auction to help needy cats. She has many items to auction off. She decides to run a *sealed bid* auction, where participants independently submit their bid for each item. She is the only one who sees each bid -- that is, she is the one who opens the envelope, and bidders do not get to see each others' bids until after Angela has decided who wins which item. For each item, Angela decides to give the item to the person who bids the most, and will also charge that person exactly what they bid.

Angela, helping the needy cats

You need to implement a method that will take as input the bids of each agent for each item, and return the total revenue that Angela will collect after running her simple auction.

Input/Output Format

Input:

The first line in the test data file contains the number **x** of test cases. After this, the **x** test cases are given one by one. Each test case begins with two integers **n** and **m**: the number of participating agents in the auction (guaranteed to be an integer greater than 0), and the number of items Angela is auctioning off (guaranteed to be an integer greater than 0). After this are **n** lines, each with **m** values; these represent the bids for each agent and each item (all guaranteed to be nonnegative integers), in order. For example, if the j^{th} entry in the i^{th} row is 12, then agent *i* bid $12 for item *j*.

Output:

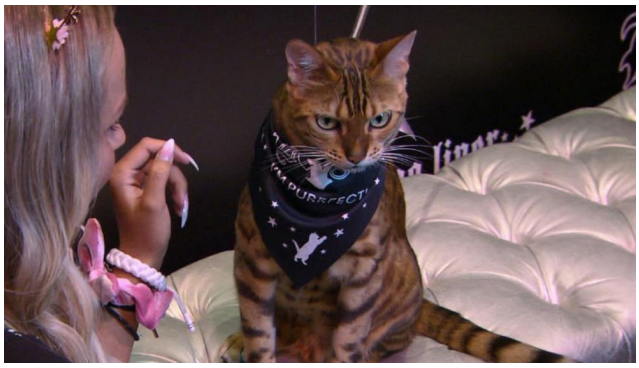
The output is a single integer representing the total revenue collected by Angela during the auction.

Examples:

Input:	**Output:**
3 3 1 1 4 2 3 3 1 2 3 2 3 1 3 1 2 2 2 10 0 10 0	4 9 10

2. Fundraising with Angela (Part II): The Purrfect Auction

Angela's initial auction (Problem 1) was so successful that she has decided to run a second auction. She has, once again, collected many items that she would like to auction off, with proceeds to benefit needy cats. As in the first auction, she will collect a sealed envelope from each participant, and that envelope will contain a bid for each item from that participant. However, Angela has decided to change the pricing and allocation rules for this new auction.



A very, very needy cat

Angela has a soft spot for participants who bid perfectly ("purrfectly"). She defines a *purrfect number* to be a positive integer that is equal to two plus the sum of its positive divisors, excluding the number itself. For example, 3 is a purrfect number: it is 2 more than the sum of its divisors excluding itself, 1. If one or more participants bids a purrfect number for an item, then she will allocate that item randomly to one of the purrfect bidders, and she will charge them $0.

Angela really dislikes participants who bid imperfectly ("impurrfectly"). She defines an *impurrfect number* to be an integer that is the product of some integer and itself. For example, 25 is an impurrfect number: it is the product of 5 and itself.

If no participants bid purrfectly, then she will allocate the item to the highest bidder. If that winner bid an impurrfect number, then she will charge that winner twice their bid. Otherwise, she will charge the winner their true bid.

You need to implement a method that will take as input the bids of each agent for each item, and return the total revenue that Angela will collect after running her purrfect auction.

Input/Output Format

Input:

The first line in the test data file contains the number **x** of test cases. After this, the **x** test cases are given one by one. Each test case begins with two integers **n** and **m**: the number of participating

agents in the auction (guaranteed to be an integer greater than 0), and the number of items Angela is auctioning off (guaranteed to be an integer greater than 0). After this are **n** lines, each with **m** values; these represent the bids for each agent and each item (all guaranteed to be nonnegative integers), in order. For example, if the j^{th} entry in the i^{th} row is 12, then agent *i* bid $12 for item *j*.

Output:
The output is a single integer representing the total revenue collected by Angela during the purrfect auction.

Examples:

Input:	**Output:**
3 3 1 1 4 2 3 3 1 2 3 2 3 1 3 1 2 2 2 10 0 10 0	8 0 0

3. Conga Line at the Cafe Disco

The Michael Scott Paper Company was so successful that it was quickly acquired by Dunder Mifflin. Its founder, owner, and president Michael Scott converted the company's old office space into an espresso-laden discotheque, crudely titled Cafe Disco. To everyone's surprise, this cafe was a huge success, and is currently renowned in the greater Scranton area for its dance parties -- specifically, its conga lines, where every participant picks their own integer and shouts it repeatedly while dancing around. In this way, we can view a conga line as an ordered list of integers.

Cafe Disco: not a great cafe, not a great disco.

Conga lines are fun to rearrange. At Cafe Disco, conga lines are especially fun if they are *redoublable*. We call a list of integers A with even length *redoublable* if and only if it is possible to reorder it such that $A[2 * i + 1] = 2 * A[2 * i]$ for every $0 \leq i < \text{len}(A) / 2$. (For our purposes, every conga line is of even length.)

You need to implement a method that will take as input the current ordered conga line of integers and return `true` if and only if it is redoublable; otherwise, your method should return `false`.

Input/Output Format

Input:

The first line in the test data file contains the number **x** of test cases. After this, the **x** test cases are given one by one. Each test case consists of a single line beginning with a nonnegative number **n**, representing the size of the **n**-length array of integers in the array that may (or may not) be redoublable. Following this are **n** integers representing the initial state of the conga line.

You may assume that **n** is even and is less than 100,000. The absolute value of each element in the **n**-length array is not larger than 500,000.

Output:

The output of your method is the boolean `true` if the array is redoublable, and `false` otherwise. (Our shell code will translate this to a string, either “This conga line can be reversed.” or “This conga line cannot be reversed.”)

Examples:

Input:	**Output:**
3 6 1 2 4 4 8 16 4 4 -2 2 -4	This conga line cannot be redoubled. This conga line can be redoubled.

4. Mindful Drienf

Given the name of a character, your method needs to determine how many unique ways can you rearrange the letters in that name given the rules that you need to take the spaces and capitalization into account.

Note that since there are repeated letters in "David Wallace" but there are also capitalized letters, things can get interesting. The two 'l' characters are interchangeable, as are the three 'a' characters, but the 'D' and 'd' are not. Because there are two "words" in the original name, the re-arrangement must also have two "non-empty words" in it, each starting with a capitalized letter. Valid examples include “Daivdal Wecla" and “Waivdal Decla" and “D Weclaaivdal" (just as a few of the many).

Dunder Mifflin can be shuffled to become Mindful Drienf

Input/Output Format

Input:

The first line in the test data file contains the number **x** of test cases. After this, the **x** test cases are given one by one. Each test case consists of a single line with a name on it. The name might be one “word” or two “words” long.

Output:

The output of your method is the number of unique ways to rearrange the input name by the given rules.

Examples:

Input:	**Output:**
2 Stephanie David Wallace	Name Stephanie can generate 20160 unique possibilities. Name David Wallace can generate 6652800 unique possibilities.

5. Accounting with Oscar

Oscar loves to catch miscreants. It saves the company money, and it gets him closer to retirement. He spends a huge portion of his time validating expense reports from the salespeople at Dunder Mifflin. He wishes this validation could be faster; that's where you come in.

Oscar, a man of integrity, a man of justice

Oscar is validating receipts from salespeople who go to predictable restaurants. He wants to pick out anomalies, and quickly. Toward that end, he has accessed credit card info for many of his fellow employees. Your task is to help flag odd behavior. Specifically, given a master string M and a search string S -- representing transactions at these restaurants -- Oscar needs to find the minimum window in M which will contain all the characters in S.

Input/Output Format

Input:

The first line in the test data file contains the number **x** of test cases. After this, the **x** test cases are given one by one. Each test case consists of two strings. First, a master string is given. Next, on a new line, a search string is given. Both strings can be assumed to be nonempty.

Output:

Your code should output a string representing the minimum window cover for the search string. If no window is found, your code should output "".

Example:

Input:	**Output:**
4 STANLEY	NLEY BANC YKAP

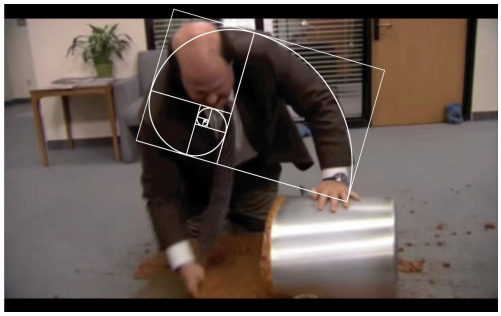
NLY ADOBECODEBANC BANC KELLYKAPOOR YP KELLYKAPOOR KR	KAPOOR

6. Kevin’s Famous Chili

To quote Kevin: “At least once a year, I like to bring in some of my Kevin's Famous Chili. The trick is to undercook the onions. Everybody is going to get to know each other in the pot. I'm serious about this stuff. I'm up the night before pressing garlic and dicing whole tomatoes. I toast my own Ancho chilies. It's a recipe passed down from Malones for generations. It's probably the thing I do best.” In short: Kevin is serious about chili!

A pot full of Kevin’s famous chili …

This year, Kevin is concerned that he won’t be able to slice, dice, add, and expand his ingredients into the perfect chili his fans in the office have come to expect. He is only worried about four ingredients: onions, tomatoes, Ancho chilies, and garlic. He has between 1 and 9 of each of these ingredients (integers). He can use the operators *, /, +, -, (,) to combine all of these ingredients into the number 24, the perfect combination of integer inputs into his famous chili.



… Chili perfection

You should help Kevin! To do so, write a method that takes as input 4 integers between 1 and 9, representing his stock. Your method should tell Kevin if he can use the operators *, /, +, -, (,) to get his stock to the (integer) number 24.

Input/Output Format

Input:

The first line in the test data file contains the number **x** of test cases. After this, the **x** test cases are given one by one. Each test case consists of exactly 4 integers representing the amount of each ingredient Kevin has.

Note:

The division operator " / " represents *real* division, not integer division. For example, 6 / (1 - 2/3) = 18. Every operation is executed between exactly two numbers. In particular, we cannot use " - " as a unary operator. For example, with [1, 1, 1, 1] as input, the expression -1 - 1 - 1 - 1 is not allowed.

Output:

For each test case, your program should output the boolean `true` if the inputs can be operated on in the appropriate fashion to reach 24, and `false` otherwise.

Note:

We have provided a skeleton program that reads the input and prints the output. All you need to do is implement the body of `makeChili(int[] stock)` to return the correct value.

Example:

Input:	**Output:**
3 8 3 8 3 1 1 9 5 3 8 3 8	Kevin's chili is going to be great this year. Kevin's chili is not going to be great this year. Kevin's chili is going to be great this year.

7. Kevin's Big Spill

Kevin took his big pot of chili to the Dunder Mifflin offices, but unfortunately dropped the pot before anyone could sample it. Fortunately, nobody else was in the office when he arrived, so he should be able to scoop some of that chili back into the pot. However, because the Dunder Mifflin offices are very old and very dirty, some amount of chili will be trapped in the carpet. Kevin wants to calculate how much chili he will have to leave behind.

Poor Kevin

You can assume that the floor is a one dimensional line, covered in variable-height, one-unit-sized (integer) blocks of carpet. The elevation of the carpet is also given in increments of one unit. Kevin will not be able to recover any chili that is trapped in "bucket", that is, a portion of the floor surrounded by walls of carpet on either side. For example, in the figure below, six units of chili are trapped: one unit is trapped at x=2 by walls of height 2 on one side and height 3 on the other; two other units of chili are trapped at x=5 and x=7 by walls of height 2 on both sides, and a final three units of chili are trapped in a bucket between x=9 and x=11. The figure below traps a total of six units of chili.

Six units of chili will not be recovered by Kevin

Assume you know the elevation map (a list of nonnegative integers) of the office floor. Your task is to write a method to tell Kevin how much chili will be left behind no matter how hard he tries to scoop it back into his pot. Note that Kevin can scoop up any chili that flows off the sides of the floor.

Input/Output Format

Input:

The first line in the test data file contains the number **x** of test cases. After this, the **x** test cases are given one by one. Each test case consists of a single line beginning with a positive integer **n**, representing the size of the **n**-length floor. Following this are **n** nonnegative integers representing the height map of the office floor.

Output:

For each test case, your program should output a nonnegative integer representing how much chili was trapped in the floor (e.g., for the example above, your program would output the integer 6).

Example:

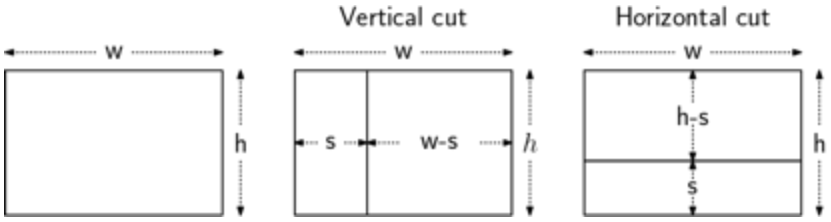
Input:	**Output:**
3 3 2 0 2 7 0 2 0 2 0 1 0 13 0 2 1 3 2 1 2 1 2 1 0 2 0	2 3 6

8. Cubic(ubic(ubicles)les)les

Everyone knows there are two common traits of every office---lots of cubicles, and lots of hierarchies. Kelly Kapoor is head of the planning committee to reallocate cubicle space for the year 2020 in the Scranton offices of Dunder Mifflin.

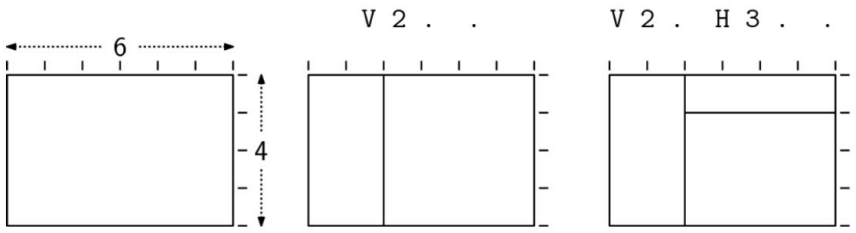
Kelly has very strong opinions about cubicle placement

Kelly starts with a rectangular office space that is w feet wide (along the x-axis) and h feet high (along the y-axis). A **vertical cut** of size s, where 1 ≤ s ≤ w-1, splits the rectangle into a left rectangle of width s and a right rectangle of width w-s. A **horizontal cut** of size s, where 1 ≤ s ≤ h-1, splits the rectangle into a lower rectangle of height s and an upper rectangle of height h-s.



A **hierarchical cubicle partition** (**HPC**) is defined as follows. First, we start with an initial rectangle of some given width w and height h. The directive "`V s`" performs a vertical cut of size s, and the directive "`H s`" performs a horizontal cut of size s. In either case, this then followed by a sequence of directives for partitioning the left (or lower) subrectangle and a sequence of directives for partitioning the right (or upper) subrectangle. The directive "`.`" (a period) stops the recursive cutting process and returns to the previous level.

For example, starting with a rectangle where w = 6 and h = 4, the directive sequence "`V 2 . .`" means perform a vertical cut of size 2 ("`V 2`"), then do no further cutting to the left side ("."), and do no further cutting to the right side ("."). The directive sequence "`V 2 . H 3 . .`" starts with the same vertical cut of size 2 ("`V 2`"), does no further cutting to the left side ("`.`"), the right side is then split by a horizontal cut of size 3 ("`H 3`"), no further cutting is done to the lower side ("`.`"), and no further cutting is done to the upper side ("`.`"). (See the figure below.)



Write a program that is given the width and height of the entire office and then a sequence of cutting directives. The program outputs an “ASCII” drawing of the final HPC. For example, the figure below shows the drawing for the three cubicle layouts of the previous figure.

```
+--+--+--+--+--+--+    +--+--+--+--+--+--+    +--+--+--+--+--+--+
|                 |    |     |           |    |     |           |
+  +  +  +  +  +  +    +  +  +  +  +  +  +    +  +  +--+--+--+--+
|                 |    |     |           |    |     |           |
+  +  +  +  +  +  +    +  +  +  +  +  +  +    +  +  +  +  +  +  +
|                 |    |     |           |    |     |           |
+  +  +  +  +  +  +    +  +  +  +  +  +  +    +  +  +  +  +  +  +
|                 |    |     |           |    |     |           |
+--+--+--+--+--+--+    +--+--+--+--+--+--+    +--+--+--+--+--+--+
```

Input/Output Format

Input:

The first line contains the number of trials to run. After this, each line consists of two positive integers, indicating the initial width and initial height. The remainder of the line is a sequence of cutting directives, separated by spaces. No error checking is needed. You may assume that the input is syntactically well formed, and the cutting sizes are all legal.

Output:

Each drawing starts with a single line “`Drawing t`” where t is the index of the current trial. This is followed by the ASCII drawing of the final cubicle partition.

We have provided a skeleton program that reads the input. All you need to do is implement the body of `drawCubicles(int w, int h, ArrayList<String> hcpList)`, which prints the drawing given the initial width w and height h and an array-list of input tokens. For example, for the input “`V 2 . .`”, `hcpList` would contain the four elements “`V`”, “`2`”, “`.`”, and “`.`”. (Some examples are given next.)

Example:

Input:	**Output:**
3 6 4 V 2 . . 6 4 V 2 . H 3 . . 6 4 V 2 H 1 . . H 3 V 2 . . .	Drawing 1 +--+--+--+--+--+--+ | | | + + + + + + + | | | + + + + + + + | | | + + + + + + + | | | +--+--+--+--+--+--+ Drawing 2 +--+--+--+--+--+--+ | | | + + +--+--+--+--+ | | | + + + + + + + | | | + + + + + + + | | | +--+--+--+--+--+--+ Drawing 3 +--+--+--+--+--+--+ | | | + + +--+--+--+--+ | | | | + + + + + + + | | | | +--+--+ + + + + | | | | +--+--+--+--+--+--+