Contest Problems
Philadelphia Classic, Spring 2015
Hosted by the Dining Philosophers
University of Pennsylvania

dining philosophers

UNIVERSITY OF PENNSYLVANIA | COMPUTER SCIENCE CLUB

Rules and Information

This document includes 13 problems. Novice teams do problems 1-9; standard teams do problems 5-13.

Any team which submits a correct solution for any of problems 1-4 will be assumed to be a novice team. **If you are not a novice team, please skip problems 1-4.**

Problems 1-4 are easier than problems 5-9, which are easier than problems 10-13. These problems are correspondingly labeled “Novice”, “Intermediate”, and “Advanced.” Order does not otherwise relate to difficulty, except that problem 13 is the hardest.

You may use the Internet only for submitting your solutions, reading Javadocs, and referring to any documents we link you to. You **may not** use the Internet for things like StackOverflow, Google, or outside communication.

As you may know, you can choose to solve any given problem in either **Java or Python**. If you would like to solve a problem in Java, we have provided a stub file that takes care of the parsing for you. If you would like to solve a problem in Python, you must create a file for the answer and parse the input file yourself. It may be useful to refer to the Java stubs for how this works. Python submissions should be named the same thing as the Java stub, but with a .py extension instead of .java (“ChooseYourStarter.py” instead of “ChooseYourStarter.java”, for example). If you use Python, you may refer to the Python standard library docs.

Do not modify any of the given methods or method headers in our stub files! They are all specified in the desired format. You may add class fields, helper methods, etc as you like, but modifying given parts will cause your code to fail in our testers.

There is no penalty for incorrect submissions. You will receive 1 point per problem solved. A team’s number of incorrect submissions will be used only as a tiebreaker.

Some problems use Java’s “long” type; if you are unfamiliar with them, they’re like an “int”, but with a (much) bigger upper bound, and you have to add “L” to the end of an explicit value assignment:

```
        long myLong = 1000000000000L;
```

Otherwise, the “long” type functions just like the “int” type.

1. Novice Problem - Choose Your Starter Pokemon

The first thing any new Pokemon trainer must do is choose their starting pokemon. This is a huge deal, and countless years have been spent debating the best starter, dating all the way back to Squirtle vs. Charmander vs. Bulbasaur.

Here, you are given information about the types of the three pokemon of your first opponent, and you must choose your starter Pokemon carefully based on the battles ahead of you. Each battle will include three foes, each with a particular type. You want to pick the Pokemon that will fare the best, as indicated by the table below.

Foe Type	Squirtle	Charmander	Bulbasaur
Water	0	-1	+1
Fire	+1	0	-1
Grass	-1	+1	0
Electric	-1	0	0
Rock	+1	-1	+1
Flying	0	0	-1

The score of each starter Pokemon is the sum of scores of all types it faces. For instance, a Charmander facing a water type Pokemon would reduce its score by 1. Thus, the Pokemon that would fare the best overall is the one with the best sum. If there is a tie in score, prefer to choose Squirtle, followed by Charmander, followed by Bulbasaur. Figure out which Pokemon will be the best!

Input Format

The input contains multiple test cases, each on one line. Each line of the input represents one battle, and it contains three strings, representing the types of the Pokemon faced in that battle. All these battles are passed as a String **battleTypes** in the Java stub, with each string containing 3 space-separated types.

Output Format

For each battle, output a string “squirtle”, “charmander”, or “bulbasaur” (in lower case), corresponding to the best starter pokemon for that battle.

Input Limits

There will be at most 216 test cases in the input.

Sample Input:	**Sample Output:**
fire fire fire	squirtle
water electric rock	bulbasaur
electric electric flying	charmander

2. Novice Problem - Gotta Rank ‘em All

For the uninitiated, every Pokemon has a distinct number associated with it. As a computer science student this mapping intrigues you in a number of ways. In this problem we are concerned with the numbers of the Pokemon in your deck. When your CS teacher told you about the rank of an item in an array, you immediately saw that there was a problem ripe for solving. The rank of an element of an array is the number of elements less than it in the array. So the rank of the least element of an array is zero and the rank of the greatest element is the length of the array minus one.

What you want to know is this: given a deck of distinct Pokemon cards, are there any cards where the number of the Pokemon is equal to the rank of that card in your deck? However you quickly realize that because Pokemon numbering starts at one this is actually impossible. Rather than accept this boring answer you act like a computer science teacher and make the problem harder. You accomplish this by saying that any card in your deck is greater than any card in the rest of your Pokemon card collection (effectively offsetting the rank by the size of the rest of your collection).

We will provide you with an array of integers representing the numbers of the Pokemon in your deck and an integer representing the number of cards in the rest of your collection. The cards in the rest of your collection (which you do not have access to) always **come before** the array of cards you are given. You can assume that the Pokemon are distinct. What you should return is a boolean that is true if there is a Pokemon whose number is its rank, and false if there is not.

Input Format

The input consists of multiple test cases. Each test takes up one line and consists of a series of numbers separated by spaces.

The first number is **numPokemon**, the number of Pokemon cards you currently have. The next **numPokemon** integers are the Pokemon cards in your deck (passed as an int[] **pokemon** in the Java stub file) and the last integer is **remainderSize,** the number of cards in the rest of your collection.

Output Format

The output consists of a single line with either true or false depending on the answer. The format of true and false is determined by how Java translates booleans into strings.

Sample Input:	**Sample Output:**	**Explanation**
5 134 5 1 12 2 3	true	**numPokemon** = 5 **pokemon** = [134, 5, 1, 12, 2] **remainderSize** = 3 There are 5 cards listed here: [134, 5, 1, 12, 2], and 3 more in the collection.

		The cards in the collection are ranked 0 to 2 (since there are 3 of them), card 1 is ranked 3, card 2 is ranked 4, card 5 is ranked 5, card 12 is ranked 6, and card 134 is ranked 7. Thus, card 5 fulfills the desired condition.
5 134 5 1 12 2 10	false	There are 5 cards listed here: [134, 5, 1, 12, 2], and 10 more in the collection. The cards in the collection are ranked 0 to 9, card 1 is ranked 10, card 2 is ranked 11, card 5 is ranked 12, card 12 is ranked 13, and card 134 is ranked 14, so no card has number same as its rank.
7 3 7 1 100 24 5 10 0	false	There are 7 cards listed here: [3, 7, 1, 100, 24, 5, 10], and no other cards in the collection. No card has number same as its rank.

3. Novice Problem - Pokeblocks

Pokeblocks are candy given to Pokemon to increase their competitive conditions for beauty pageants. They are made from berries, and the specific combination of berries make them different colors. Given the recipes for different colored blocks and an array of different types of berries and their amounts, you need to find the most Pokeblocks that those berries can produce.

We will provide an array that represents the berries. The specific berry for each array index, as well as the recipes for each Pokeblock, is listed below. Not every berry may be used. From this data, output the amount of each color of Pokeblock that could be produced. (You should consider each color block independently: that is, you would return the number of red blocks you could make if you didn't make any other kind of block, the number of green blocks you could make if you didn't make any other kind of block, etc.)

Berries in order:
[Cheri,Chesto,Sitrus,Rawst,Figy,Wiki,Leppa]
Recipes:
Cheri+Figy+Leppa=Red Block
Sitrus+Rawst=Green Block
Wiki+Chesto=Blue Block

Input Format

The input consists of multiple test cases.

Each test case consists of a line of 7 integers, representing the array of berry amounts with the specific order listed below. This is passed as an array int[] **a** in the Java stub file.

Output Format

For each test case, return a String of space-separated numbers that represent the number of red, green, and blue blocks that can be made.

Sample Input:	**Sample Output:**	**Explanation:**
3 5 3 6 4 7 2	2 3 5	For example, there are 3 Cheri berries, 4 Figy berries, and 2 Leppa berries, so 2 red blocks can be created.
6 8 3 4 7 4 4	4 3 4	For example, there are 4 Wiki berries and 4 Chesto berries, so 4 blue blocks can be created.
4 8 4 85 4 5 3	3 4 5	For example, there are 4 Sitrus berries and 85 Rawst berries, so 4 green blocks can be created.

4. Novice Problem - Pokemon T-Shirts

Pokemon want to dress stylishly. However, since no Pokemon has worn clothing before, they need help picking the right T-Shirt. Luckily, we have come up with a clear formula for calculating the size for a Pokemon. We need your help implementing a program which tells a Pokemon which T-Shirt size will fit it best.

T-Shirt size is a function of height as follows:

- 0 ≤ h ≤ 5ft: Small
- 5 < h ≤ 10ft: Medium
- 10 < h ≤ 15ft: Large
- 15 < h ≤ 20ft: XL
- 20 < h ≤ 30ft: XXL
- >30 ft: No shirt fits

T-Shirt size is a function of weight as follows:

- 0 ≤ w ≤ 20 lbs: Small
- 20 < w ≤ 100 lbs: Medium
- 100 < w ≤ 200 lbs: Large
- 200 < w ≤ 400 lbs: XL
- 400 < w ≤ 1000 lbs: XXL
- >1000 lbs: No shirt fits

For a given Pokemon, if the two functions give two different sizes, **return the larger size**. For example, if a Pokemon weighs 10 pounds but is 7 feet tall, it gets a Medium shirt.

We will provide you with two doubles **height** and **weight**. You should return one of the following Strings.

- **S** for Small
- **M** for Medium
- **L** for Large
- **XL** for Extra Large
- **XXL** for Extra Extra Large
- **NONE** for No shirt fits

Input Format

The input file consists of multiple test cases.

Each test case consists of one line of two doubles, **height** and **weight**, representing the height in feet and weight in pounds of each Pokemon.

Output Format

For each test case (each line in the input file) print one of the appropriate size Strings above on its **own line**.

Sample Input	**Sample Output**	**Explanation**
7.0 10.0	M	The Pokemon is 7.0 feet tall and weighs 10.0 pounds. The weight function says the Pokemon needs a small, but the height function says that the Pokemon needs a medium. Thus, we go with the larger size: **medium**.
35.0 700.0	NONE	The Pokemon is 35.0 feet tall and weighs 700.0 pounds. The weight function says that the Pokemon needs a XXL, but the height function says that the Pokemon will not fit in a shirt. Thus, we go with the larger size, so the Pokemon **doesn’t get a shirt**.

5. Intermediate Problem - Catching Pokemon

You are a Pokemon trainer on a quest to catch ‘em all! There are **numPokemon** Pokemon which you want to catch, but due to budget constraints, you only have **numPokemon** Pokeballs, so you will have to use one Pokeball on each Pokemon. You also have a table of probabilities **catchProbability**, where **catchProbability**[i][j] is the chance of Pokeball i catching Pokemon j. What is the maximum probability that you manage to catch all the Pokemon successfully?

Input format

The input file will consist of multiple test cases in order.

On the first line of each test case, there is one int **numPokemon**, representing the number of Pokemon to catch (as well as the number of Pokeballs available). This is passed as an int **numPokemon** in the Java stub.

The remainder of the input represents a 2D array **catchProbability**. On each of the subsequent **numPokemon** lines of input, there are **numPokemon** doubles describing a Pokeball, with the j-th value on the i-th line representing **catchProbability**[i][j], the chance to catch the j-th Pokemon with the i-th Pokeball. This is passed as a 2D array double[][] **catchProbability** in the Java stub.

Output format

For each test case, output one single double value, to 3 decimal points, representing the maximum probability to successfully catch all the Pokemon.

Input Limits

There will be at most 10 test cases.

1 <= **numPokemon** <= 10

Sample Input	**Sample Output**	**Explanation**

2
0.7 0.5
0.5 0.7
	0.490	Using the first Pokeball on the first Pokemon, and the second Pokeball on the second Pokemon, the chance to catch each Pokemon is 0.7, so the overall chance is 0.49

3
0.8 0.9 0.1
0.3 0.3 0.3
0.5 0.7 0.5
	0.168	Use the first Pokeball on the first Pokemon, the second on the third, and the third on the second

6. Intermediate Problem - Pokemon Evolution

Pokemon have a rich and fascinating evolutionary history.[1] Several professors at Penn are conducting research on the evolution of Pokemon, but they need to know how one given Pokemon is related to another. More precisely, they want to know the lowest common ancestor by which they are related. For example, are they both water pokemon? Are they both origin Pokemon? Are they only related via the ancient soup (the organic mix which predates all Pokemon)? This information is critical to learning about and understanding pokemon.

We define the BinaryTreeNode class at the bottom of Evolution.java. Note that a BinaryTreeNode is a binary tree in which each Node has exactly 0, 1, or 2 children. Each BinaryTreeNode also has a String field 'name' which can either represent a Pokemon itself or some other information about Pokemon. For example, a BinaryTreeNode object's 'name' field can be "Pikachu" or "WaterType."

We will provide you with a BinaryTreeNode named **root** which is the root of the evolutionary tree. We will also provide you with two Strings **a** and **b**. For each String, there is exactly one BinaryTreeNode in the tree with a name field corresponding to that string. You should return the lowest common ancestor[2] of the BinaryTreeNodes corresponding to the given strings.

Input Format

Read the input file until it ends. The file begins with a test case. Every time you encounter a line with "*", the previous test case has ended and the next test case will begin on the next line. Each test case begins with two strings **a** and **b**. For each String, there is exactly one BinaryTreeNode in the tree with a name field corresponding to that string. The lines following this firt line until the "*" line give you information on how to construct the BinaryTree for this test case. Each line represents a single BinaryTreeNode. On each line, we give you four variables separated by spaces.

- **id** is an integer and it represents the ID of that line's BinaryTreeNode. **id** is always between 0 and 100 (including 100).
- **name** is a string and it is the name field of that line's BinaryTreeNode.
- **left** is an integer and it represents the ID of the BinaryTreeNode's left child. If left is negative, the BinaryTreeNode does not have a left child.
- **right** is an integer and it represents the ID of the BinaryTreeNode's right child. If right is negative, the BinaryTreeNode does not have a right child.

For example, one of these lines might look like:

[1] http://i.imgur.com/ZfT4UWw.jpg

[2] http://en.wikipedia.org/wiki/Lowest_common_ancestor
"In graph theory and computer science, the lowest common ancestor (LCA) of two nodes v and w in a tree or directed acyclic graph (DAG) is the lowest (i.e. deepest) node that has both v and w as descendants, where we define each node to be a descendant of itself (so if v has a direct connection from w, w is the lowest common ancestor)."

0 Pikachu 1 5

Output Format

For each test case print the name of the lowest common answer BinaryTreeNode on its own line.

Sample Input	**Sample Output**	**Explanation**
Raichu Blastoise 0 TheAncientSoup 1 4 1 Pichu 2 -1 2 Pikachu 3 -1 3 Raichu -1 -1 4 Squirtle -1 5 5 Wartortle -1 6 6 Blastoise -1 -1	TheAncientSoup	**Root:** TheAncientSoup | / \\ / \\ Pichu Squirtle / \\ | | | | Pikachu Wartortle / \\ | | | | Raichu Blastoise **a:** Raichu **b:** Blastoise The first node that both Raichu and Blastoise are descendents of is TheAncientSoup.

Raichu Pichu
0 TheAncientSoup 1 4
1 Pichu 2 -1
2 Pikachu 3 -1
3 Raichu -1 -1
4 Squirtle -1 5
5 Wartortle -1 6
6 Blastoise -1 -1
	Pichu	**Root:**
		TheAncientSoup
		          |
		        / \\
		      / \\
		 Pichu Squirtle
		   / \\
		   | |
		   | |
		Pikachu Wartortle
		  / \\
		 | |
		 | |
		Raichu Blastoise

		a: Raichu **b:** Pichu The first node that both Raichu and Pichu are descendents of is Pichu. While they are also both descendants of TheAncientSoup, Pichu is the “lower” ancestor.

7. Intermediate Problem - Rocket Boxes

Team Rocket is trying to arrange all the boxes in their hideout. They are trying stack the boxes in triangle formation. The top row of boxes, defined as row 1, has 1 box; the row directly beneath it, row 2, has 2 boxes, etc. such that the nth row has n boxes. Each box sits upon exactly two other boxes. Under orders from their leader, they want to stash items inside the boxes in a specific fashion: The bottom row (also the kth row), has some number k boxes. Some of these boxes contain one item, some contain two, and some are empty. They continue storing items in the boxes in the rows above by making sure that each box contains exactly as many items as the sum of the number of items in the two boxes it sits upon. Additionally, the single box at the very top must contain an even number of items. How many ways can Team Rocket stash items inside the bottom row of boxes to ensure that all of their leader's requirements are fulfilled?

Input format:

The input consists of multiple test cases.

Each test cases consists of a single line. This line contains a single integer **size**, the number of boxes on the bottom row.

Output format:

For each test case, output a single integer, equal to the number of ways to fill in the bottom row's boxes with 0, 1 or 2 items each such that the top box has an even number of items.

Input limit:

1 <= **size** <= 20

Sample Input	**Sample Output**	**Explanation**
2	5	There are 5 possible ways to fill out the bottom row: 00, 02, 20, 22, 11
3	15	There are 14 possible ways to fill out the bottom row: 000, 010, 101, 111, 002, 020, 200, 022, 202, 220, 222, 012, 210, 212, 121
4	41	0000, 0002, 0020, 0200, 2000, 0022, 0202, 0220, 2002, 2020, 2200, 0222,

		2022, 2202, 2220, 2222, 0011, 0211, 2011, 2211, 0101, 0121, 2101, 2121, 0110, 0112, 2110, 2112, 1001, 1021, 1201, 1221, 1010, 1012, 1210, 1212, 1100, 1102, 1120, 1122, 1111

8. Intermediate Problem - Pikachu goes to Pewter City

Your beloved Pikachu is setting out on a journey from Pallet Town, which is in the extreme Southwest of a section of the world map, to Pewter City, which is in the extreme Northeast of the same section of the world map. We consider this section of the world map to be a grid of squares containing either a town or grass. Pikachu is in a hurry, so he will walk one square North or one square East each day, continuing on until he reaches Pewter City. An example of this setup, and one possible path (highlighted), is below.

Grass	Grass	Grass	Pewter City
Grass	Grass	Grass	Grass
Pallet Town	Grass	Grass	Grass

Unfortunately Pikachu has bad eyesight, so he can't tell exactly where Pewter City is, and he needs your help. You have an old map that says it will take **i** days for Pikachu to reach Pewter City, not including the day he arrives. Thus, for the above example, **i** is five. In addition, the message tells you that Pewter City is **u** units North of Pallet Town, so in the above example, **u** is two.

As you send Pikachu off on his journey, he knows he must go **i** squares total and **u** squares up, but he hasn't decided on his exact route. This makes you wonder how many different routes he can take (i.e. how many distinct paths there are between Pallet Town and Pewter City in which Pikachu only moves North and East)? Unfortunately the old map only has the values of **u** and **i** in Roman numerals!

Recall that Roman numerals are computed as follows:
I = 1
V = 5
X = 10
L = 50
C = 100

Roman numerals are normally arranged greatest to least from left to right, and the numerical value is the sum of the numerical values of the digits. However, a single digit cannot be repeated more than 3 times, so we place a smaller digit in front of a larger one to denote deducting the smaller digit from the larger. Here are some examples:

III = 3
IV = 4
XIV = 14
XIX = 19

Should you need more explanation or examples, you may visit this chart.

Input Format

The input consists of multiple test cases, each consisting of two lines.

The first line of each test case contains a single string, representing the Roman numeral for **i**.

The second line of each test case also contains a single string, representing the Roman numeral for **u**.

Output Format

For each test case, output an integer representing the number of paths from Pallet Town to Pewter City. You can be guaranteed that the solution will be less than Integer.MAX_VALUE. *However, it may be helpful to use type 'long' for intermediary computation.*

Input Limits

- You can be guaranteed that the inputs will be valid Roman numerals expressed as capital letters
- You can further be guaranteed that the numeric value of **i** will be less than or equal to that of **u** (i.e. the inputs represent a valid grid)
- No Roman numerals will be used beyond I, V, X, L, and C

Sample Input	**Sample Output**	**Explanation**
IV IV	1	i = 4, u = 4 Represents a 4x1 grid One path: 4N
III I	3	i = 3, u = 1 Represents a 2x3 grid Three paths: 1N, 2E 1E, 1N, 1E 2E, 1N

9. Intermediate Problem - Avoiding Trainer Battles

You've just emerged (victoriously) from a series of battles. Unfortunately, you won only by the skin of your teeth, and now all of your Pokemon are incapacitated. At this moment, you stumble upon a veritable minefield of trainers itching for a battle.

You do have a 2-D array **map**[][] of size **m** x **n** representing a map. You wish to go from **map**[0][0] to **map**[**m**-1][**n**-1] while avoiding all of the trainers. Thankfully, you won't be completely blindsided, as we've compiled for you a list of all trainers, as well as the direction they're facing. Each trainer can see everything in a straight line in the direction he or she is facing, until his or her line of sight meets a wall or another trainer. The moment you cross a trainer's line of sight, you will have no choice but to do battle with that trainer.

You can walk on empty space, but you cannot walk through walls or trainers.Given this information, you want to find the shortest path through the map involving no battles.

Input Format

The input file consists of multiple test cases.

The first line of each test case contains two integers, **m** and **n**, representing the height and width of the grid respectively. These are passed as ints **m** and **n** in the Java stub.

The next **m** lines of the input represent the 2D array **map**. Each line contains **n** characters. Each character is either a "#" indicating a wall, a "." indicating an empty space, or one of the following characters "v", ">", "<" or "^", indicating a trainer facing a particular direction (for instance, "v" is a trainer facing downwards). This is passed as a 2D array char[][] **map** in the Java stub.

Output Format

For each test case, output a single boolean, true if it is possible to reach your destination without encountering a single trainer, false otherwise. The format of true and false is determined by how Java converts booleans to strings

Input Limits

There will be at most 10 test cases.

1 <= **m,n** <= 100

Sample Input	Sample Output	Explanation

`5 5`
`.....`
`.....`
`>....`
`.....`
`.....`
	false	There is no way to get around the trainer without crossing his or her line of sight

`5 5`
`.....`
`>..#.`
`.....`
`.#..<`
`.....`
	true	Traveling all the way to the right, then 2 steps down, then all the way to the left, then 2 steps down, then all the way to the right will help you get to your destination

10. Advanced Problem - Between n Rocks and a Diglett's Cave

On your journey as a Pokemon trainer you find yourself in one of the many caves in the land of Hoenn. You spend many hours trying to find your way out of the cave but you get tired of having to continually use Rock Smash. Rather than waste time smashing rocks, you decide to sit down with a map of the cave and devise a program to determine the least number of rocks you need to break to get out of the cave

We will give you a 2D array of integers representing the map and your starting and ending locations. If you can stand on space the corresponding element of the array will contain 0, 1 if it contains a rock, and 2 if you can't stand on it. You can move from a space up, down, left, or right, *not diagonally*. You should output the minimum number of rocks needed to get through and -1 if there is no path.

In the sample input we visualize the array with X for false, 0 for true. Indexing starts with (0,0) in the top left, (row, column).

Input Format

The input file consists of multiple test cases in order.

The first line of each test case consists of 6 integers separated by spaces. The first two integers are **rows** and **cols**. The second two are **startr** and **startc**, and the final two are **endr** and **endc**. The next **rows** lines consists of **cols** characters each a 0, 1 or 2 meaning the same thing as the diagram. The goal is to move from (**start**, **startc**) to (**endr**, **endc**).

Output Format

For each test case, output a single integer, representing the number of rocks we need to break to get to our destination. If it is not possible to reach our destination, output -1 instead.

Input limits

0 < rows*cols <= 10000

Sample Input	Sample Output	Explanation

`4 6 1 0 0 5`
`222000`
`001000`
`001000`
`002000`
	1	The grid has 4 rows and 6 columns, with starting point (1,0) and ending point (0,5). We need to break one rock to reach our destination.

`5 7 1 0 0 3`
`2210022`
`0211200`
`0022220`
`0000000`
`2222222`
	-1	The grid has 5 rows and 7 columns, with starting point (1,0) and ending point (0,3). It is not possible to reach our destination.

`5 7 1 0 0 3`
`2210022`
`0211200`
`0012220`
`0000000`
`2222222`
	3	The grid has 5 rows and 7 columns, with starting point (1,0) and ending point (0,3). We need to break 3 rocks to reach our destination.

11. Advanced Problem - Pokemon Training

There are some people who think that Pokemon training is merely walking around randomly in tall grass looking for wild Pokemon to battle. These people are wrong; you have managed to get Pokemon training down to a science - you can even influence what wild Pokemon you meet in wild grass! You are attempting to train one of your Pokemon to be able to defeat the Elite Four in a particular patch of wild grass which contains **numPokemon** different types of Pokemon, and thus, you wish to obtain as many experience points as possible in one round of training.

Your Pokemon begins at **startHP** hit points, and will lose a fixed amount of hit points with every battle, depending on what wild Pokemon he or she battles. Also, for each wild Pokemon you successfully defeat (without reaching 0 hit points), you will gain a certain number of experience points, also depending on the wild Pokemon you fought. Your Pokemon will not recover hit points in between battles, and you can stop your training any time you wish.

Thus, given a list of wild Pokemon you can encounter (you can encounter the same Pokemon more than once if you wish), what will be the maximum number of experience points you can gain without needing to go back to the Pokemon center to heal?

Input Format

The input file consists of multiple test cases.

The first line of each test case contains two ints **numPokemon** and **startHP**, the number of different types of Pokemon, as well as the starting hit points of your own Pokemon.

The next **numPokemon** lines of input each represent one type of Pokemon you can encounter in the wild grass. Each line contains two ints, **damageDealt** and **xpGained**, the exact damage dealt to your Pokemon in defeating this wild Pokemon, and the resultant number of experience points gained after defeating it.

Output Format

For each test case, output a single int, representing the maximum number of experience points you can gain without reaching 0 hit points.

Input Limits

There will be at most 10 test cases.
1 <= **numPokemon** <= 400
1 <= **startHP** <= 400
1 <= **xpGained** <= 1000

Sample Input	**Sample Output**	**Explanation**
1 5 1 10	40	You can defeat 4 Pokemon, each time taking 1 point of damage and receiving 10

		experience points.

3 11
7 20
4 11
2 3
	25	You can defeat two of the second Pokemon and one of the third Pokemon, receiving 4 + 4 + 2 = 10 points of damage, and gaining 11 + 11 + 3 = 25 experience points.

12. Advanced Problem - Island Ferries

Piplup lives on a group of icy cold islands, numbered from 1 to **numIslands**. Today, he needs to get from island 1, which he is currently on, to some island numbered **destIsland**. However, it is particularly cold today, so he does not feel like swimming. Instead, he will take special ferries from island to island to reach his destination.

There are **numFerries** ferries in total, each which has a route from one island to another island, with a fixed cost. No route between two islands has more than one ferry servicing it, and each ferry can only be used once. However, the ferry captains have decided to be particularly scheming today! They are trying to fleece our poor Piplup for as much money as possible. However, due to special regulations, they are not able to just outright demand more money. Instead, what they have decided to do is to secretly shuffle their routes such that Piplup will have to pay the most money to get to his destination.

Thus, before Piplup begins his journey, the captains of ferries which begin at the same island will collude with each other to swap destinations amongst themselves (while keeping each ferry at the same cost). For example, if there are two ferries leaving from island 2, which lead to islands 3 and 4, for cost 20 and 30 respectively, they could choose to swap destinations amongst themselves, so that the cost to reach island 4 is 20, and the cost to reach island 3 is 30 instead.

Help Piplup to determine how much he will have to spend to reach his destination, assuming the ferries shuffle themselves into the worst configuration possible!

Input Format

The input file consists of multiple test cases.

The first line of each test case contains two ints, **numIslands** and **numFerries**, the number of islands and ferries available.

The second line of each test case contains one int, **destIsland**, the number of the island which Piplup wishes to reach

The next **numFerries** lines of input represent one ferry each. Each line contains three ints, **startIsland**, **endIsland**, and **ferryCost**.

Output Format

For each test case, output a single int, representing the minimum amount Piplup will have to spend to reach his destination, assuming the ferries shuffle themselves into the worst configuration possible for him.

Input Limits

There will be at most 10 test cases.

1 <= **numIslands** <= 10,000

1 <= **numFerries** <= 40,000

1 <= **ferryCost** <= 1,000,000

Sample Input	**Sample Output**	**Explanation**

3 3
3
1 2 10
1 3 5
2 3 4
	9	The captains of the first and second ferry will switch destinations, so that it will cost 10 dollars to reach island 3 from island 1, but only 5 dollars to reach island 2 from island 1. Then, the best route will be to travel to island 2 for cost 5, then to island 3 for cost 4.

5 6
5
1 2 10
1 3 8
1 4 9
2 5 10
3 5 9
4 5 8
	18	The captains on the first island will shuffle themselves such that it will cost 8 dollars to reach island 2, 9 to reach island 3, and 10 to reach island 4. Then, any route to island 5 will cost a total of 18 dollars.

13. Advanced Problem - Professor Oak's Challenge

After completing many challenges, Professor Oak presents you with a final challenge. He reveals this mysterious formula:

$$\frac{1}{x} + \frac{1}{y} = \frac{1}{n}$$

where x, y, and z are all positive integers. For a given limit **lim**, he defines N(**lim**) to be the number of integer solutions which satisfy x < y <= **lim**.

This formula is said to reveal the ancient secrets to how Pokemon were formed. Unfortunately, even Professor Oak cannot figure out how to calculate N(**lim**) quickly. He asks your team to figure it out for him.

Input Format

The input consists of multiple test cases, each which consists of a single integer **lim**.

Output Format

For each test case, output a single integer, representing the value N(**lim**).

Input Limits

1 <= **lim** <= 50,000,000

Sample Input	**Sample Output**	**Explanation**

15	4	These are the only solutions:
		x=3, y=6, n=2
		x=4, y=12, n=3
		x=6, y=12, n=4
		x=10, y=15, n=6

1000	1069	Trust us.