**ACM High School Programming Contest**

**2020 Programming Problem Set**

**DO NOT OPEN THIS PACKET UNTIL INSTRUCTED TO BEGIN!**

1. **General Notes**
   1. Do the problems in any order you like. They do not have to be done in order from 1 to 12.
   2. All problems have a value of 60 points. Incorrect submissions receive a deduction of 5 points but may be reworked and resubmitted. Deductions are only included in the team score for problems that are ultimately solved correctly.
   3. There is no extraneous input. All input is exactly as specified in the problem. Unless specified by the problem, integer inputs will not have leading zeros. Unless otherwise specified, your program should read to the end of file.
   4. Your program should not print extraneous output. Follow the form exactly as given in the problem.
2. **Names of Problems**

|  |  |
| --- | --- |
| **Number** | **Name** |
| Problem 1 | Isabelle |
| Problem 2 | Frustrating Frustums |
| Problem 3 | Feeding Bevo |
| Problem 4 | Bone Apple Tea |
| Problem 5 | CS Cat of Good Grades |
| Problem 6 | The High-Lays Man |
| Problem 7 | Roadside Assistance |
| Problem 8 | Pyramint |
| Problem 9 | What’s for Dinner? |
| Problem 10 | Base Conversion |
| Problem 11 | Madison’s Wardrobe |
| Problem 12 | Passwords |

1. **Isabelle**

**Program Name: Isabelle.java**

here

**Constraints**

**Input**

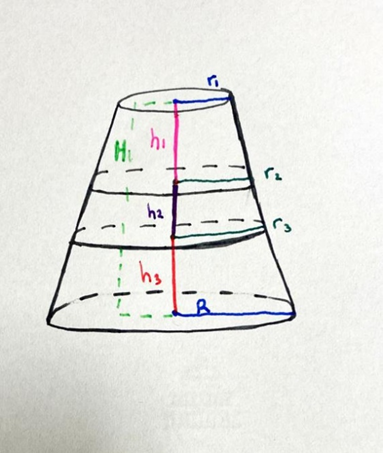
**Output**

**Example Input File**

**Example Output to Screen**

1. **Frustrating Frustums**

**Program Name: Frustums.java Input File: frustums.dat**

Do you know what a frustum is? It’s basically when you take a cone and cut the top off. Kind of looks like a circular plateau. The cool thing is you can find the volume of a frustum using the following formula:

***Volume (V) = 1/3 \* pi \* h(r2 + R2 + r\*R)***

where h is the height, R is the Radius of the bigger circle of the frustum and r is the radius of the smaller circle. Using this information, you can find the volume of any frustum. But can you find the frustum *within* the frustum?

If you cut a frustum into three pieces horizontally, you technically have 3 frustums. So, each frustum has an H, R, and r. We labeled the frustum in the picture. Given all the information except for r2, r3, and h1 can you find the volume of the middle frustum within the frustum?

**For clarification, here is what each variable stands for in the picture:**

H – Whole Height of the original frustum

R – Bottom Radius of whole frustum

r1 -top radius of the whole frustum, and top radius of the top frustum

r2 –  top radius of the middle frustum, and bottom radius of the top frustum

r3 - top radius of the bottom frustum, and bottom radius of the middle frustum

h1 -height of top frustum

h2 - height of middle frustum

h3 -height of bottom frustum

**Constraints**  
Each input variable will be a positive real number less than 100.

**Input**

The first and only line of input will contain five real numbers in the following order: r1, R, H, h2, h3.

**Output**

Output the volume of the middle frustum as described. Round your answer to EXACTLY three decimal places.

**Example Input File**

2 4 10 3 2

**Example Output to Screen**

64.424

1. **Feeding Bevo**

**Program Name: FeedingBevo.java Input File: feedingbevo.dat**

A dog looking at the camera

Description automatically generated

Rick and Lillian recently took their dog, Bevo, to the vet and were told that Bevo needs to be on a more regular feeding schedule to lose weight. However, Rick and Lillian are very forgetful and will sometimes feed Bevo more than they should, at the wrong time, or even forget to feed him at all. Rick has put together a button device that Bevo has been trained to press when he wants food. This button will send the date and time to a device that will determine if it is time to feed Bevo. Bevo is only allowed to eat once per day between 4pm and 7 pm.

**Input**

The first line of input will contain a single integer t that indicates the number of days of data. The first line of each day will contain a single integer n that denotes the number of times Bevo has pressed the button that day. Each of the following n lines will contain a date and time in the format example 09/27/1995 13:52.

**Output**

For each time Bevo presses the button:

* Print “It is not time to feed Bevo!” if it outside of Bevo’s feeding time.
* Print “Feeding Bevo at [time].” if it is within Bevo’s feeding time, where [time] denotes when Bevo pressed the button.
* Print “Bevo has already eaten!” if Bevo presses the button more than once during his feeding time.

At the end of each day, print the date followed by either “Bevo ate today.” or “Bevo did not eat today.” Print “=====” after each day.

**Example Input File**

2

1

02/26/2020 19:00

3

03/18/2019 01:14

03/18/2019 16:00

03/18/2019 16:12

**Example Output to Screen**

It is not time to feed Bevo!

02/26/2020 Bevo did not eat today.

=====

It is not time to feed Bevo!

Feeding Bevo at 16:00.

Bevo has already eaten!

03/18/2019 Bevo ate today.

=====

1. **Bone Apple Tea**

**Program Name: BoneAppleTea.java Input File: boneappletea.dat**

Your English teacher has just assigned a project to the class. It seems easy enough, however your teacher decrees it a partner project, and as luck would have it, the partner you were assigned is not the sharpest tool in the shed. There are nine phrases in particular that your partner cannot spell: angel, bon appetit, bonjour, close, disgust, fool proof, human being, kindergarten, and synonym. These nine phrases are vital to the project, and no matter how much you try to explain, your partner cannot understand. So you decide to make a program that will translate these phrases into a language they can understand.

You will make a library consisting of the nine strings: “angle”, “bone apple tea”, “bone jaw”, “clothes”, “discuss”, “full proof”, “human bean”, “kidney garden”, and “cinnamon”. You will make a program that reads the phrases your partner cannot understand and prints the best match for each phrase from the library above.

**Input**

The first line of input will consist of a single integer t, denoting the amount of phrases to be translated. Each subsequent line contains one string s, which is the phrase to be translated.

**Output**

For each phrase, output the correct translation.

**Example Input File**

9

bonjour

synonym

bon appetit

fool proof

disgust

angel

close

kindergarten

human being

**Example Output to Screen**

bone jaw

cinnamon

bone apple tea

full proof

discuss

angle

clothes

kidney garden

human bean

1. **CS Cat of Good Grades**

**Program Name: .java Input File: Name.dat**

here

**Constraints**

**Input**

**Output**

**Example Input File**

**Example Output to Screen**

1. **The High-Lays Man**

**Program Name: HighLaysMan.java Input File: highlaysman.dat**

You and your buddy are on your way back from a road trip at night. Your buddy is driving and you are in the passenger seat. You are going down a highway when you suddenly see a small object in your headlights - it appears to be a bag of chips. Not far behind the chip-bag, you see some kids and a car sitting still in your lane. Your friend quickly switches lanes, but then you see another in that lane a little further up the road. Then another, then another. He starts having trouble navigating. You can see where all of the obstacles are from much further away than your friend so he asks you to guide him through this trial.

You know that your favorite mapping app doesn't know where the chip bags are, so it is up to you. You figure that the most efficient way to do this is to impose where you think the chip bags - and consequently, the people - are on the road, and write a custom program to find out if you can even make it through safely before you even worry about what path to take.

Because you don't know exactly where the chip bags are and your friend doesn't want to get too close to them anyway, you don't want to get within two grid-cells of where you believe the bags end up on the map, unless there is a barrier in the way of where you would pass it.

**Constraints**

1 ≤ q ≤ 10

3 ≤ n, m ≤ 25

**Input**

The first line of data will contain an integer q, which indicates the number of datasets to follow. For each dataset, the first line will contain two integers n and m, which are the dimensions of the map you are using. The next n lines will contain m characters each, representing your map. Each map will have exactly one ‘S’ and one ‘E’ character, which indicate your starting and ending position, respectively. Additionally, the map will have ‘#’ characters, which are barriers you can’t cross, ‘.’ characters, indicating safe travel, and ‘L’ characters, which symbolize the bags of chips you cannot travel within two grid-cells of.

**Output**

For each dataset, if you can travel from the given starting position to the given ending position, output “YES”. Otherwise, output “NO”.

**6. The High-Lays Man (cont.)**

**Example Input File**

5

6 6

#..E.#

#....#

#...L#

#....#

#....#

#..S.#

6 5

#.E.#

#...#

#..L#

#...#

#...#

#.S.#

6 6

....E.

......

L.....

......

.....L

S.....

6 6

.....E

......

L.....

....L.

......

S.....

3 7

...L...

...#...

S.....E

**Example Output to Screen**

YES  
NO  
YES  
NO  
YES

1. **Roadside Assistance**

**Program Name: Roadside.java Input File: roadside.dat**

Your business model has been a smashing success. Your company offers emergency assistance to drivers who have car troubles during their travels, such as the random flat tire one might encounter whilst trapped on a bridge. Unfortunately, you didn’t anticipate this much…*traffic*. More and more customers are calling in every day, requesting help from your drivers. Some assistance drivers have to travel directly from one accident to another. The congestion is putting a strain on your business, and you need to optimize the routes you take to each accident. This is not an easy task. There are many possible accident locations, and each possible pair of locations could potentially have hundreds of different paths linking them. For instance, in the figure to the right, there is a road linking Location #0 with Location #1. It is of length 7. However, you could also take a different path between those two locations (0 -> 2 -> 3 -> 1), which is 6 units long. The actual city (in the judges’ data) has many more than just 5 locations, which further complicates the task of optimizing routes.

**0**

**1**

**2**

**3**

**4**

3

7

2

1

4

Your job, given a map of the city, is to determine the shortest distance between two given accident locations in each query.

*(The figure shown to the right is a visual example of the sample input.)*

**Constraints**

1 ≤ n ≤ 100

1 ≤ m ≤ n\*(n-1)/2

1 ≤ q ≤ 100

0 ≤ a,b,u,v *<* n

1 ≤ w ≤ 1000

**Input**

The first line of data will contain three integers n, m and q. The first integer n indicates the number of possible accident locations. The next integer m denotes the number of roads in the map. Each road is a two-way street that connects two distinct accident locations. The third integer q stands for the number of queries you will be given. Each query is a request for the minimum distance between two locations.

The next m lines each contain three integers a, b and w. These lines indicate that the two locations a and b are linked via a two-way road of length w.

The next q lines each contains two integers u and v. These two integers serve as a query for your program to solve.

**Output**For each query, display the minimum possible distance between u and v. It is guaranteed a driver can reach location v from location u, and that a ≠ b and u ≠ v for all input. In other words, no road will link an accident location to itself, and no query will request the distance from a node to itself.

**7. Roadside Assistance (cont.)**

**Example Input File**

5 5 2

0 1 7

0 2 3

1 3 2

1 4 4

2 3 1

0 1

2 4

**Example Output to Screen**

6  
7

1. **Pyramint**

**Program Name: Pyramint.java Input File: pyramint.dat**

You’ve obtained a very large supply of mints from your…well, it doesn’t really matter where they came from. Don’t worry about it. They’re here. Your task? Assemble them in the perfect shape: a pyramid of mints. A pyramint! You’ll need to think through just how many mints will be required to complete this engineering feat. To make your calculations easier, we’ll say a pyramint has a four-sided base, and a height of n. More formally, a pyramint is defined as a three-dimensional structure with the following recursive definition:

A pyramint of height 1 (a 1-pyramint) is just a single mint.

For n > 1, an n-pyramint has a base of an nxn square of mints, directly above which is an (n-1)-pyramint.

**Constraints**1 ≤ t ≤ 10

1 ≤ n ≤ 3,000,000

**Input**

The first line of input will contain a single integer t that indicates the number of test cases to follow. The first and only line of each test case will contain a single integer n that denotes the height of the requested n-pyramint.

**Output**

For each test case, output the number of mints needed for an n-pyramint.

**Example Input File**

5

1

2

4

13

1860

**Example Output to Screen**

1

5

30

819

2146682110

1. **What’s for Dinner?**

**Program Name: Dinner.java Input File: dinner.dat**

Flavius and his friends are hungry. However, no one ever seems to agree on where to eat dinner together. His friend Josephus creates the following process to pick who decides what’s for dinner.

Start with n people standing in a circle. One person is designated Person 1 and then they are numbered (clockwise) through to Person n. Person 1 then removes Person 2 from the circle. Continuing around the circle, Person 3 removes Person 4 from the circle, Person 5 removes Person 6 from the circle, and so on. This procedure continues with the next person removing the closest person next to them clockwise. They go on until only one person remains in the circle, and this person must choose what’s for dinner.

Flavius, however, only wants to eat at one restaurant… Olive Garden. Instead of standing at a random position in Josephus’s circle and hoping for the best, Flavius wants you to write a program to determine where he should stand in the circle so that he will always be able to pick Olive Garden for dinner.

**Constraints**1 ≤ t ≤ 12

2 ≤ n ≤ 10000

**Input**

The first line of input will contain a single integer t that indicates the number of test cases to follow. The first and only line of each test case will contain a single integer n that denotes the number of people standing in the circle.

**Output**

For each test case, output “Person” followed by the position number Flavius should stand at in the circle.

**Example Input File**

3

2

8

12

**Example Output to Screen**

Person 1

Person 1

Person 9

1. **Base Conversion**

**Program Name: Base.java Input File: base.dat**

Two plus two equals eleven. Sound wrong? Couldn’t possibly be correct? Well, what if I wrote it like this?

23 + 23 = 113

Ah, yes. Numbers in different bases. The question is, for a given number x, which base is the *preferred* representation of that number? For this problem, a *preferred* base for a number is whichever base contains the most appearances of the digit six. So, for the number 144 in base 10, the base that contains the most sixes is 23. (14410 -> 6623). No other base of 144 has more than one six in its representation.

Keep in mind, bases above base 36 are not properly defined, so this problem will not include any base above base 36. If two or more bases both contain the same largest number of sixes, choose the larger base. It is guaranteed there is at least one base less than or equal to base 36 where the representation of x contains at least one six.

**Constraints**

1 ≤ t ≤ 10

6 ≤ n ≤ 231-1

**Input**

The first line of input will contain a single integer t, which will indicate the number of test cases to follow. The following t lines will each contain a single integer x, which is assumed to be given in base 10.

**Output**

For each test case, output the *preferred* base for the given number x.

**Example Input File**

3

144

89

13

**Example Output to Screen**

23  
14  
7

1. **Madison’s Wardrobe**

**Program Name: Wardrobe.java Input File: wardrobe.dat**

Madison recently was hired and is trying to prepare herself to enter the job force. She knows she has to buy new work shirts, but wants to make it seem like she has a bigger wardrobe than she really does. Her work week is only four days long (Monday, Tuesday, Wednesday, Thursday), so she could buy four shirts (red, black, grey, white) and wear those in that order every week, but then her coworkers would judge her. She wants to know, based on the number of shirts she buys, how many ways she can arrange N shirts before she has to repeat the same sequence of N shirts.

**Input**

An unknown number of test cases are given. Each test case contains a single integer N, the number of shirts Madison is considering buying. This integer can be as little as 0 and as high as 1000. Madison has a large closet, so it’s okay.

**Output**

For each integer given, print the number of ways Madison could arrange N shirts for N days.

**Example Input File**

0

1

2

3

4

5

6

**Example Output to Screen**

1

1

2

6

24

120

720

1. **Passwords**

**Program Name: Passwords.java Input File: passwords.dat**

At your most recent programming competition, you had to sign in to an account that they provided to help the judging process go smoothly. They just happened to have problems with the passwords... again. The proctors couldn't get the given passwords and usernames to work correctly and tried to update their software and passwords on the spot then they needed a different router- it was a big mess. Since then, they have tried to be a little more organized; and even though keeping up with passwords is something else entirely, they want to make sure that any password they settle on is good. That's where you come in. Any given person can have a different idea of what makes a good password for any given contest, so each set of passwords will be judged on a given set of conditions. A password's strength is determined by how many of the conditions they meet.

100% = Excellent,

75-99.99% = Strong,

50-74.99% = Moderate,

25-49.99% = Weak,

0.01-24.99 = Terrible,

0% = Abysmal.

Each command is as follows:

Key: [] - literal characters, b - a boolean (true/false) (must have/must not have) (directly/subsequently), C - comparison operator (<,=,>), N - a given integer value

Capital b | can or can't have a capital letter [A-Z]

Lowercase b | can or can't have a lowercase letter [a-z]

Number b | can or can't have a number [0-9]

Special [] b | can or can't have the specific character

Length N C | length must be <=, ==, or >= (C) the given number, N

Follow []1 []2 b1 b2 | []1 must or can't (b2) follow []2 directly or subsequently (b1)

**Input**

The first of input will contain a single integer n, describing the number of test cases to follow. The first line of each test case contains a single integer k, indicating how many filters for the passwords will exist. The next k lines will describe the filters. The next line in the data set will have a single integer m, the number of passwords to test. The next m lines will each contain a single line of characters (no whitespace) to run through the filters.

**Output**

Output each password, followed by a colon (:), and its strength. Print “=====” after each dataset.

**12. Passwords (cont.)**

**Example Input File**

2

4

Capital true

Lowercase true

Length 9 >

Special ! false

5

RalphIsAGirl

RALPHISAGIRL

ralphisagirl!

rlphgrl!

234!

6

Special \* true

Length 5 =

Follow A B true true

Follow \* C true false

Follow \* C false false

Follow \* B false true

6

BA\*CD

BAC\*D

B\*ACD

BACD\*

BA\*C\*

AC\*B

**Example Output to Screen**

RalphIsAGirl: Excellent  
RALPHISAGIRL: Strong  
ralphisagirl!: Moderate  
rlphgrl!: Weak  
234!: Abysmal  
=====  
BA\*CD: Excellent  
BAC\*D: Moderate  
B\*ACD: Strong  
BACD\*: Strong  
BA\*C\*: Moderate  
AC\*B: Terrible  
=====