Carnegie Mellon

imc financial markets

Spring Programming Competition – Practice Round

March 26, 2010

You can program in C, C++, or Java; note that the judges will re-compile your programs before testing.

Your programs should read the test data from the standard input and write results to the standard output; you should not use files for input or output.

All communications with the judges should be through the PC^2 environment.

Each problem's estimated level of difficulty is shown next to the title. This guidance is only an estimate, and is based on many, possibly incorrect, assumptions. Take it as only one, of many, factors contributing to your estimate of your own team's ability to solve the problem. Remember: Your mileage may vary!

(☺ ☺ ☺ ☺ ☺)	Very Challenging
(☺ ☺ ☺ ☺)	Challenging
(☺ ☺ ☺)	Nuances or subtlety, or upper level math/algo/problem solving
(☺ ☺)	Some nuances or library or algorithmic knowledge required
(☺)	Straight-forward, few if any nuances

The hypothetical examples below illustrate the notation for a straight-forward problem, Problem Y, and a very challenging problem, Problem Z:

Problem Y: Easy points! (☺)

Problem Z: Badge of Honor, e.g. Killer (☺ ☺ ☺ ☺ ☺)

Problem A: A Duty for the Bridges (☺ ½)

Medieval castles were fortresses made of heavy stone, shuttered with heavy wood, and protected by deep moats crossable only via certain narrow, heavily guarded bridges. Inside of their great walls, the royals enjoyed all of the luxuries of the Court, including the best of jesters, crochet, and scotch.

But, the King's Court also mediated disputes and managed the business of the kingdom. Those with business before the Court enter the castle via the designated bridge. And, much like many bridges of today, it was a toll bridge. Jesters, crochet, and scotch are not free, to be sure.

And, since there really is no fair way to tax working people to pay for the luxury of the rich, no such attempt was made. The taxation system was arbitrary. All those with business before the court were assessed a duty to cross the bridge. A duty of one *dinar* per bridge crossing was imposed for each letter of a person's name. Because of the Court's generosity to itself, it is important to note that the duty is assessed both entering and leaving the castle. In a small act of generosity, the court only charges for letters – not spaces, hyphens, or other punctuation or adornments.

Your job, given a list of names of petitioners who visited the Court on a particular day, is to determine the total *dinars* collected for their bridge crossings.

Input
The first line of the input will contain an integer n ($1 \leq n \leq 100$) which represents the number of persons visiting the Court on a particular day. Following that there will be *n* lines of input, each of which contains a non-empty string no longer than 80 characters, representing a person's name.

Output
For the given input, your program should output a single number, the total number of *dinars* collected from the provided list of people, each of who entered and exited the castle via the bridge.

Sample Input
```
2
Mike
Jane Smith
```

Sample Output
```
26
```

Problem B: Cold-Blooded Carry (☺ ☺ ½)

In magical times, dragons served as modern-day tanks. Because of their breadth of fire, they were a nearly invincible mode of transportation. Any enemy could simply be blasted away.

But, unfortunately, dragons are cold-blooded. Their body temperature adjusts to match that of their environment. Should they become either too hot or too cold, well, let’s just say that they aren’t such good transportation any more. In particular, the temperature of the dragon must remain between 89 F and 99 F, inclusive.

As a result, dragon-riding warriors and royalty need to be thoughtful of their otherwise invincible friends. The riders must ensure that they adjust their speed of travel across sunny and shady areas to maintain the proper temperature of the dragon throughout the journey.

A rider might need to speed up when riding through certain areas to minimize the impact they have on the dragon’s temperature, e.g. speed up in a shady area to minimize heat loss. Similarly, the rider might need to slow the dragon down while crossing an area in order to maximize its impact, e.g. slow down in a sunny area to gain more heat. A dragon’s body temperature increases one degree per hour when in a *SUNNY* area and loses one degree per hour when in a *SHADY* area. A dragon can slow down or rest, but it cannot travel faster than 100 miles per hour.

Your task is to determine the minimum time, to the nearest whole hour (any fraction of an hour rounds up) necessary to complete the journey, without over heating or under heating the dragon. If it is not possible to complete the journey while protecting the health of the dragon, you should determine that it is impossible.

Description:	“SHADY 100”	“SUNNY 200”	“SUNNY 100”	“SHADY 100”	
	•	•	•	•	•
Mile Marker:	0	100	300	400	500

Input
The first line of the input contains a positive integer, *n*, that represents the number of test cases contained within the input. The *n* test cases immediately follow this line.

The first line of each test case contains two positive integers: *s* ($1 \leq s \leq 500{,}000$) which represents the number of sections of the journey and ($89 \leq t \leq 99$), which represents the starting temperature of the dragon. Each line of the test case describes a segment and contains either *SUNNY* or *SHADY*, followed by a positive integer representing the length of the segment in miles. The first segment begins at position *0*. Each subsequent segment begins where the prior segment ended.

Output
For each test case, your program should output the minimum number of hours, rounding up to the nearest hour), necessary to complete the journey without over heating or under heating the dragon. If it is not possible to complete the journey while protecting the health of the dragon, the program should output *IMPOSSIBLE* for the test case in place of a number.

Sample Input
```
2
4 98
SHADY 100
SUNNY 200
SUNNY 100
SHADY 100
2 98
SUNNY 100
SUNNY 200
```

Sample Output
```
6
IMPOSSIBLE
```

Problem C: Magical Quilt (☺ ☺ ☺ ☺ ½)

In modern times quilts are often sewn and given as gifts as a sign of love and thoughtfulness. They also provide soft, cuddly warmth on cool days and can be designed to compliment and enrich any décor. But, in times gone by, properly constructed quilts were thought to bring their holder magical powers.

In order to have magical powers, a quilt must be stuffed with down feathers and be designed in a grid pattern of properly arranged *GOLD*, *SILVER*, and *PLATINUM* squares. *PLATINUM* squares give the quilt magic, *SILVER* squares channel the power to the holder of the quilt, and *GOLD* squares look really pretty.

The quilt must be exactly 6x6. It must never have more than three *GOLD* squares touching each other vertically or horizontally, or they interfere with each other preventing the quilt from conveying any powers. The same is true of *SILVER* squares. No interference is created diagonally, and no matter how the quilt is folded, wrap-around is not a concern.

			P	P	
P			P		
P			P	P	
		P		P	
	P	P			P
P			P		

Example Input

G	G	G	P	P	S
P	S	G	P	S	S
P	G	S	P	P	S
G	S	S	S	P	G
G	P	P	S	G	P
P	S	S	P	G	S

Illegal Tiling

S	G	S	P	P	S
P	G	S	P	G	S
P	S	G	P	P	G
S	S	P	G	P	S
G	P	P	G	S	P
P	S	S	P	G	G

Legal Tiling

Input
The first line of the input contains a positive integer, *n*, that represents the number of test cases contained within the input. The *n* test cases immediately follow this line.

The first line of each test case contains one non-negative integer: *p* ($0 \le p \le 36$) which represents the total number of platinum squares in the quilt. The next *p* lines of each test case contain the row and column for each *p*, where rows and columns are number *0…5*.

Output
For each test case, you should print a single line containing the maximum magic, a.k.a. number of gold squares, obtainable with the described configuration of platinum squares.

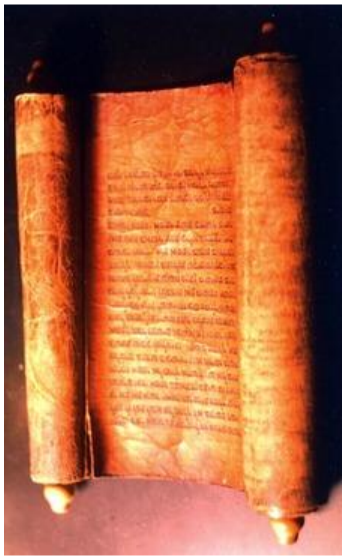
Sample input

```
1
34
0  0
0  1
0  2
0  3
0  4
0  5
1  0
1  1
1  2
1  3
1  4
1  5
2  0
2  1
2  2
2  3
2  4
2  5
3  0
3  1
3  2
3  3
3  4
3  5
4  0
4  1
4  2
4  3
4  4
4  5
5  0
5  1
5  2
5  3
```

Sample output

```
2
```

Problem D: Sacred Scrolls (☺☺)



A secret room was recently discovered in the ruins of an ancient monastery. It contains a large quantity of scrolls held as sacred by the monks who once formed the community. Because there are so many scrolls, historians will need to prioritize their study of the documents. In order to do this, they decide to study the most sacred scrolls before the least sacred scrolls.

The sacredness of a scroll is the maximum sacredness of any line of text within the scroll. A line of text is not sacred (has a sacredness of 0), unless a single character composes more than half of the line, e.g. “aba” is sacred, but “cdb” is not. If a line is sacred, its sacredness is equivalent to the number of instances of the dominant characters in the string. The dominant is exactly the single character in the majority (not plurality), within a string. For example, *aadabaabb* has a sacredness of *5*. The ancient language uses only the characters *a*-*z*.

Your task, is to compute the sacredness of a text given its lines.

Input
The first line of the input will contain an integer n ($1 \leq n \leq 100$) which represents the number of test cases, a.k.a. scrolls. The first line of each test case contains l lines, where ($l > 0$). The subsequent *l* lines within each test case contain the text of each line. Each line will have at most 100 characters.

Output
For each test case, you should print out a single number representing the sacredness of the scroll.

Sample Input
```
2
4
abbadabba
babaabbab
bbbababab
abca
1
abcdefggh
```

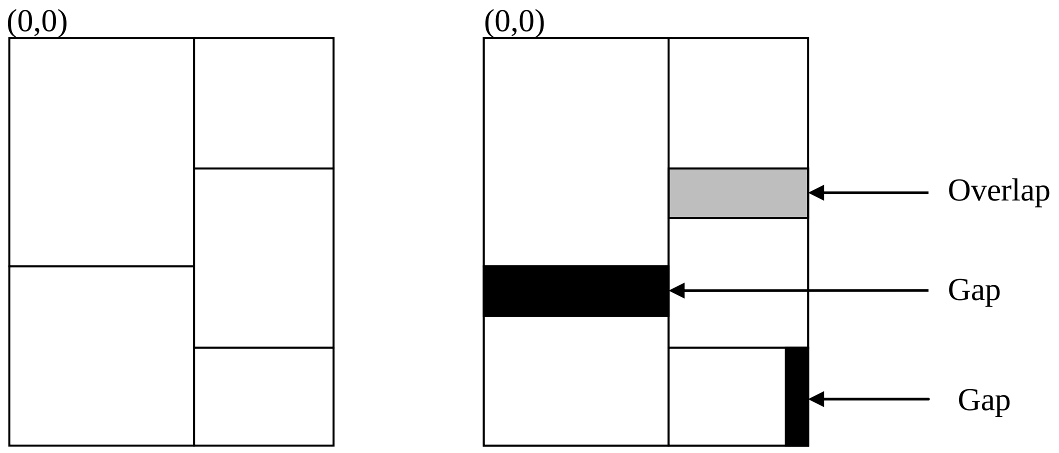
Sample Output
```
6
0
```

Problem E: Transparent Wood (☺ ☺ ☺ ☺)

A powerful storm has passed through your village, destroying all of the windows. Unfortunately, it will take the glassmakers months or years to make enough glass for everyone to repair their windows. Rather than wait for glass, which proved it's self to be too fragile, anyway, you visit the village magician.

She provides you with rectangular sheets of transparent wood, but cautioned that if they are cut or overlapped, they will be weakened and fail within the month. You hire a handyman to install your newly purchased transparent wood windows.

But, before you pay, you want to ensure that the installation was proper and did not overlap any sheets or leave any gaps between or around them. So, you write a program that, given the dimensions of the window opening, the dimensions of each sheet, and the coordinates of each sheet's upper left corner, verifies that the installation meets the requirements.



Input
The input includes multiple test cases. The first line of the input contains a single non-negative integer indicating the number of test cases to follow. Each test case describes the opening and the sheets of wood used. The first line of each test case contains the three non-negative integers, *o, h, and s*, where the first number, *o,* indicated the width of the opening. The second number, *h,* indicates the height of the opening. And, the third number, *s*, indicates the total number of sheets of transparent wood used to over the opening. The test case is further described by *s* lines, each of which contains four non-negative integers describing a sheet of wood. The first two, *x*, and *y*, are its coordinates, where the origin is the upper-left corner of the opening. The next two numbers on the line are *sw* and *sh*, the width of the sheet and the height of the sheet, respectively. There will be fewer than 100,000 sheets of transparent wood. The *o,h,sw,* and *sh* will be between 1 and 10,000,000. *x* and *y* will be between 0 and 10,000,000.

Output
The output consists of one line for each test case *PAY* or *WITHHOLD.* The output should be *PAY*, if the window is installed correctly and the installer should be paid, and *WITHHOLD*, otherwise.

Sample input
```
2
8 10 3
0 0 8 5
0 5 4 5
4 5 4 5
8 10 3
0 0 8 5
0 5 3 5
4 5 4 5
```

Sample output
```
PAY
WITHHOLD
```

Problem F: Boxed! (☺ ☺ ☺ ½)

Okay. Admit it. You've got an *eBay* habit. And, it has left you with a stack of empty shipping boxes in your closet. You'd like to pack the boxes within each other to get them out of the way, leaving you with fewer visible boxes.

This could be a really hard problem, but all of the boxes have square bottoms, e.g. the width and length of the bottom are the same. And, you are only interested in stacking one box into another – you do not want to place two or more boxes side by side or on top of each other, etc, within the same box. The goal is simple box-into-box packing.

Given a list of boxes, described by the widths and lengths of the base and the height of the box, please determine how many boxes will be left over after nesting as many boxes as possible into a single box. A box can fit into another box if it is strictly smaller in all dimensions. Boxes cannot be rotated.

Input
The input will consist of multiple test cases. The first line of the input file contains a single non-negative integer, *n*, that indicates the total number of test cases present within the input. The test cases follow on the subsequent lines. Each test cases begins with a line containing a single non-negative number, *b*, indicating the number of boxes to be described within the test case. The subsequent *b* lines contain a description of a box. Each line consists of three integers, *w* ($0 < w \leq 100$), *l* ($0 < l \leq 100$), and *h* ($0 < h \leq 100$), where *w* and *l* are guaranteed to be the equal and represent the dimensions of the bottom of the box and *h* represents its height. There will be fewer than 10,000 boxes.

Output
For each test case, you should print a single line. It should be a single integer indicating the total number of left over boxes after filling one box with as many other boxes as is possible.

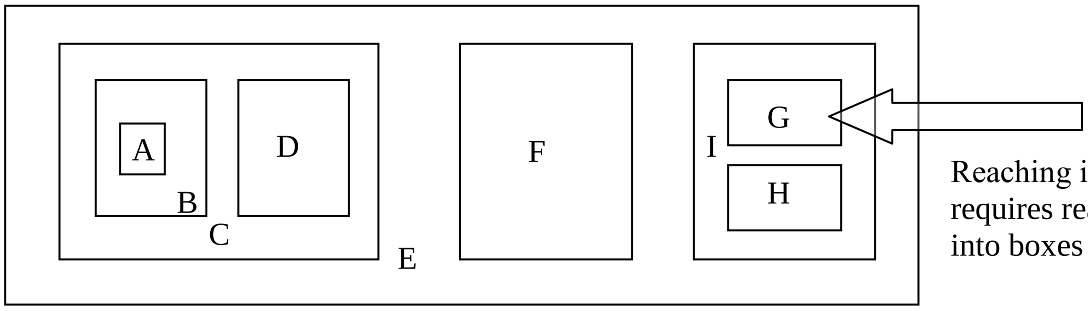
Sample input
```
2
3
4  4  3
3  3  2
2  2  1
5
4 4 4
4 4 3
4 4 2
3 3 2
2 2 1
```

Sample output
```
0
2
```

Problem G: Revenge of the Boxes (☺ ☺ ☺)

In your move from one building to another, you boxed up your office. The process was a bit haphazard. In general, you put small things, like paperclips into small boxes, then larger things like staplers and tape dispensers into larger boxes. And, yet larger things, like the telephone and stereo into yet larger boxes. Then, you took all of the boxes and shoved them into each other, as space allowed, sealed them all up, and called it a mission accomplished.

Unpacking is a lot of work. And, well, time is always short. So, you've never quite gotten around to it. Instead you just sift through the boxes to get things when you need them. But, to save space, you never take one box out of another. Instead, you just open one box, reach into it to get to another, open that, to get to another, and repeat until you get into the intended box.



Reaching into "Box G" requires reaching through or into boxes E, I, and G.

Given information about how the boxes are packaged, this question asks you how many boxes you'll need to reach through or into to get into a specific one of them.

Input
The first line of the input will contain a non-negative integer *n* which represents the number of test cases present in the input. The input cases follow. Each test case begins with a line containing a non-negative integer *b* (1 <= *b* <= 10,000), which represents the number of boxes in the test case, and a string *t*, which identifies the target box using a label as described below. Following this line, each test case contains one line per box. A line contains two box labels such that a line reading "BoxA BoxB" is interpreted as "BoxA is directly inside of BoxB." The outer-most box is described as being within itself, e.g., "BoxA BoxA". Box labels are composed only of alphabetical characters and cannot exceed 35 characters in length. The input given will always represent a valid nesting of boxes.

Output
For each test case, you should print out a single line containing a single number: The number of boxes through or into which you need to reach to get into the target box.

Sample Input
```
2
9 G
A B
B C
C E
D C
E E
F E
G I
H I
I E
5 E
A B
B C
C E
D C
E E
```

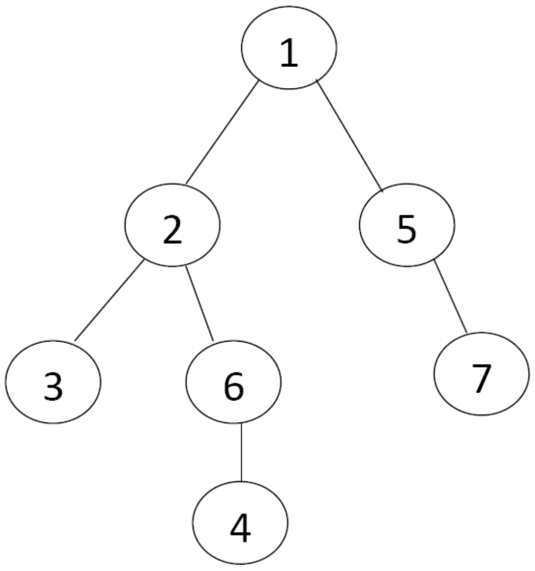
Sample Output
```
3
1
```

Problem H: Trees (☺ ☺ ☺ ☺ ☺)

Given a rooted binary tree where each vertex is numbered we can construct a sequence of numbers as follows:

```
Construct Sequence(v):
    Add v to the sequence
    if(v has one child)
        ConstructSequence(v's child)
    if(v has two children)
        ConstructSequence(v's child with smallest label)
        ConstructSequence(v's child with largest label)
```

Calling ConstructSequence(root) will generate the sequence of numbers for the tree with the given vertex as root. For example on the following tree the sequence would be: 1,2,3,6,4,5,7



You will be given a sequence of numbers, out of all trees such that ConstructSequence generates the given sequence, and returns the height of the tree with minimum height. The height of a tree is the longest distance from a vertex to the root. For example of the height of the example tree is 3 (the path from the root to vertex 3 is the distance 3).

Input
The first line of the input will contain T the number of test cases. Each test case starts with N (between 1 and 250), the length of the sequence, on a line by itself. The next line will contain N space separated integers that represent the sequence. No number will occur more than once in the sequence. Each number in the sequence will be between 1 and 1000.

Output
For each test case output the minimum height on a line by itself.

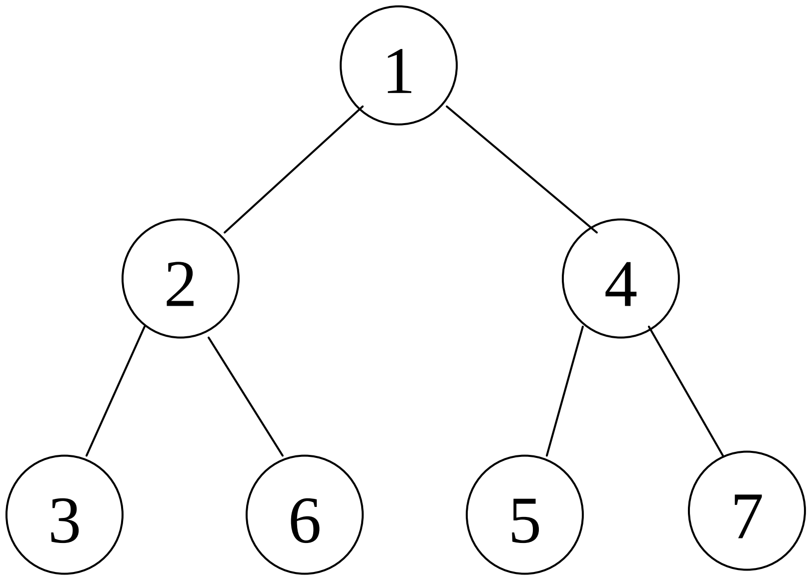
Sample Input

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

Sample Output

```
2
1
2
```

Below is a picture of a tree of height 2 which generates the sequence given in the first test case



Problem I: Blackboards (☺☺)

Andrew Carnegie has come back from the dead to give a lecture at CMU. The lecture is very important so he wants everything to go off perfectly. The room he is in has N sliding blackboards numbered 1 through N. Each blackboard is on a separate track. Each blackboard can slide to an integer height between 1 and H inclusive. If two blackboards are at the same height the lower numbered one obscures the higher numbered one.

Andrew Carnegie has carefully written his notes over all N blackboards. At some point in time during the lecture Mr. Carnegie wants some subset of the boards viewable. A board is viewable if it is the lowest numbered board at its height. Mr. Carnegie wants to know whether the boards can be slid such that all of the boards in a specific subset are viewable.

Input
The first line of the input contains an integer *T* that represents the number of test cases. The first line of each test case contains three non-negative integers: *N, H, and S*. *N* is the number of sliding blackboards, it will be between 1 and 1,000,000,000. *H* is the maximum height a blackboard can be placed at it will be an integer between 1 and 1000. *S* is the size of the subset that Mr. Carnegie wants to show. S will be between 1 and 1000. The next line will contain *S* integers representing the subset of the boards that Andrew Carnegie wants to show. These numbers will be between 1 and N inclusive, and all of the numbers will be distinct.

Output
For each test case, you should print out a single line. If it is possible to make all of the boards in the subset visible output “YES”, otherwise output “NO”.

Sample Input
```
2
3 2 2
1 2
3 2 2
2 3
```

Sample Output
```
YES
NO
```