The 3rd Annual University of Akron Programming Competition

Presented by
The University of Akron Computer Science Department
Association for Computing Machinery Student Chapter

April 21, 2012

Rules:

1. There are **eight** questions to be completed in **four hours**.

2. C, C++ and Java are the only languages available.

3. Data is read from Standard Input and output is sent to Standard Output. Do not prompt for input values in the code you submit to be judged. Do not attempt to read from or write to any files.

4. All programs are submitted as source code only. Submitting compiled information (binary, .class) will result in a Compilation Error penalty.

5. Non-standard libraries cannot be used in your solutions. The Standard Template Library (STL) and C++ string libraries are allowed. The standard Java API is available, except for those packages that are deemed dangerous by contest officials (e.g., that might generate a security violation).

6. The input to all problems will consist of multiple test cases unless otherwise noted. The input listed on the problem description is not intended to be a comprehensive input set. The input is guaranteed to adhere to the descriptions in each problem; you do not need to check for invalid input.

7. Use of personal electronics during the competition is cause for disqualification. Cell phones must be turned all the way off (not silent mode, not airplane mode, etc.). You may use any written notes, books, or reference materials you bring with you.

8. Programming style is not considered in this contest. You can code in any style or with any level of documentation your team prefers.

9. All communication with the judges will be handled through PC^2. All judges' decisions are final.

Problem A *Chocolate Chip Pancakes*

You *really* love chocolate chip pancakes. It's nearly impossible to describe in a programming contest problem description how much you love chocolate chip pancakes. Pancakes are pretty good, and chocolate chips are pretty good, but their combined power is far greater than either the sum or product of their individual powers. That's pretty powerful!

While making pancakes for you and your friends to share, you realize that you only have finitely many chocolate chips and you may or may not be able to put chocolate chips in *all* of the pancakes. You quickly calculate how many of the pancakes can have chocolate chips and you make up the batch of pancakes. Some may have chocolate chips, some may not, but you do the best you can.

You need to split the pancakes evenly amongst your friend group (the number of pancakes is such that they can be evenly distributed) and the stack of pancakes starts off with all of the chocolate chips ones at the top. You agree to make a random number of partial stack flips (by inserting a spatula under one of the pancakes and flipping the stack above that point) before distributing the pancakes around the table. For example, if you had 8 pancakes and were eating with your friends Tim, Mike, and Jarod, then you'll each end up with 2 pancakes. Let's say there are 4 pancakes with chocolate chips, and you perform one stack flip on the top 6 pancakes. After that flip, pancakes 3, 4, 5, and 6 (numbering from the top down) would have chocolate chips. Tim would get the top pancake, Mike would get pancake 2, Jarod would get pancake 3 *with chocolate chips*, you would get pancake 4 *with chocolate chips*, Tim would get pancake 5 *with chocolate chips*, Mike would get pancake 6 *with chocolate chips*, Jarod would get pancake 7, and you would get pancake 8 (the one on the bottom of the shuffled stack), so in the end you'd end up with one chocolate chip pancake.

Your task is to determine, based on the number of friends, number of pancakes, availability of chocolate chips, and sequence of partial stack flips, how many chocolate chip pancakes you'll end up with.

Details of the Input

The input will consist of multiple cases and begin with a line containing the number of cases to follow.

Each input case will begin with a line containing 3 integers representing the number p of pancakes ($0 < p \leq 1000$), the number k of people eating breakfast ($0 < k \leq p$, with k as a factor of p), and the number of c pancakes that have chocolate chips ($0 \leq c \leq p$). The next line begins with a nonnegative integer $f \leq 100$ indicating the number of pancake stack flips followed by f integers in the range $[1, p]$ representing how many (topmost) pancakes are involved in each flip, in order.

Details of the Output

For each case, print the number of chocolate chip pancakes you get to eat as shown in the Sample Output.

Sample Input

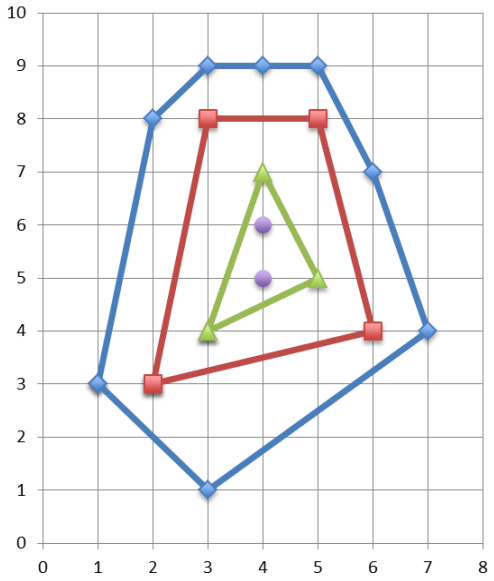
```
2
10 5 4
7 1 4 4 10 5 8 7
10 5 3
7 1 4 5 10 5 8 7
```

Sample Output

```
Case 1: Yay, I get 1!
Case 2: Aww, no chocolate chips.
```

Problem B *Zombie Defense*

Zombies are coming! And they're coming to get you… very slowly. Slowly enough that you have some time to prepare barricades to ensure that you stay alive as long as possible, just in case a helicopter flies by with a rope ladder to rescue you. You probably don't stand a chance of surviving that long, but it's worth a shot.



You decide to run into the woods behind yourself with lots of rope. Zombies can be stopped for 5 minutes by a rope loop around a set of non-collinear trees. Their vertical vision is terrible so they don't realize they could duck under or hop over the rope, but their horizontal vision and path-finding abilities are superhuman, so they can tell if a rope loop is incomplete and quickly move around the partially roped off area. Once they've started chewing through rope, their appetite for human braiiiinnnsss gives them the ability to move arbitrarily quickly through open spaces to get to you, and they only slow down for 5 minute time spans to chew through rope loops.

In order to survive as long as possible, you tie a rope around all of the outermost trees on your property (while you wish to stay alive as long as possible, you also do not want to be sued so you only use trees on your property). This gives you 5 minutes to live from the time the zombies reach the outermost border. As soon as they eat through that rope (the one wrapped around 8 trees in the figure above), the entire loop falls to the ground and the zombies immediately begin working on the next rope loop standing in their way (the one wrapped around 4 trees in the figure above). This gives you 5 more minutes to live, totaling 10 minutes. In the figure above there were enough trees to make three complete rope loops, giving you a total of 15 minutes to survive (there are two more trees available in the middle, but since they are collinear you cannot create another loop out of them).

Each tree can only touch one rope loop, including trees that don't really bear any of the load from the loop. For example, the outermost loop above touches the tree at (4, 9) so that tree cannot be used for the next loop inside. Rope loops also cannot intersect at any point; if the loop structure were viewed from above there would need to be no intersections, not even at a single point.

Details of the Input

The input will consist of multiple cases and begin with a line containing the number of cases to follow.

Each input case will be contained on one line with a single positive integer n <= 100 indicating the number of trees on the property in no particular order followed by n pairs of non-negative integers describing the x and y values of each tree ($x, y \leq 1000$). Each tree will be at a distinct location.

Details of the Output

For each case, print the case number and the number of minutes you can survive as formatted below.

Sample Input

```
2
17 3 1 1 3 2 3 3 4 6 4 7 4 4 5 5 5 4 6 4 7 6 7 2 8 3 8 5 8 3 9 4 9 5 9
8 0 0 0 10 10 10 10 0 2 5 4 5 6 5 8 5
```

Sample Output

```
Case 1: Survive for 15 minutes.
Case 2: Survive for 5 minutes.
```

Problem C *Construction Superstitions*

Due to triskaidekaphobia, many buildings in the United States do not have a floor 13. Well, all buildings with at least 13 floors have a 13th floor (we'll always start numbering floors at 1), but the floor labels purposely skip 13. Thus, the 13th floor is labeled as floor 14 with normal numbering following from there. We will refer to such buildings as triskaidekaphobic buildings. For this problem, these buildings do not physically skip a floor (leaving it empty or using it for storage), but rather the 13th physical floor is simply relabeled as floor 14.

In China, 13 is sometimes considered a lucky number, so most of their buildings with at least 13 floors have a floor labeled as floor 13. Due to tetraphobia, however, many buildings in China will skip all floor numbers containing *any* digit 4 in its decimal representation (we have no clue why they aren't concerned about representations in other number bases). We will refer to such buildings as tetraphobic buildings. Just like triskaidekaphobic buildings, floors with a 4 in their decimal representation are not physically skipped, but rather such floor labels are skipped.

A new skyscraper was planned to be built in Cleveland, OH in the United States, so the building plans skipped 13 when labeling the floors. Due to some extremely bizarre zoning laws, the new building can only be constructed in China and all of the floors need to be relabeled to change it from a triskaidekaphobic building to a tetraphobic building. Your job is to write a program to help with this conversion.

Details of the Input

The input will consist of multiple cases and begin with a line containing the number of cases to follow.

Each input case will consist of a single positive integer $f \leq 257$ ($f \neq 13$, of course) indicating the floor number to be converted (as of the date of this contest, the largest building in the world, Burj Khalifa in Dubai, has a total of 211 floors; we believe 256 floors ought to be enough for anybody and the 256th floor of a triskaidekaphobic building is labeled as 257).

Details of the Output

For each case, print the case number and the floor number conversion from a triskaidekaphobic building to a tetraphobic building as shown in the Sample Output.

Sample Input

```
4
3
4
18
58
```

Sample Output

```
Case 1: Floor 3 becomes 3
Case 2: Floor 4 becomes 5
Case 3: Floor 18 becomes 19
Case 4: Floor 58 becomes 73
```

Problem D *Randomized Playlist*

Congratulations! You've just received an internship at Macrosift for the summer doing QA on the Zuhn music player integration into Windoughs Phoan Seveghn. Since you listen to music on your Windoughs Phoan Seveghn all the time, you decide to get started early to really impress your boss. Somewhat embarrassingly, you only ever listen to the same album – the new Justen Beebar disc! Sure, your friends make fun of you for this, but it's not your fault that a kid with tons of real musical talent is also some kind of teenage heartthrob. You're not about to deny his musicality over that little detail, unlike all the haters out there.

Anyway, listening to just these few songs over and over on "random" has you thinking about various random playlist implementations. One such implementation would be to simply keep picking songs at random, one at a time, over and over. However, since the Justen Beebar disc only has 8 songs and it's the only album on your Windoughs Phoan Seveghn, there is a 1 in 8 chance of a chosen song being the same as the previous song, and even if it's distinct, there's then a 1 in 4 chance the next song will be a duplicate of the previous two songs!

A better approach might be to apply a random ordering to the entire playlist at the beginning (using a Fisher-Yates shuffle), play through the songs in that new order, and then once every song has been played just create a new random ordering and repeat the process. This is better because you'll get to hear every Justen Beebar song at least once before you start hearing repeats, but there's still a chance of his hit song *Baby Baby Baby Oooooh* being the last song in the first random ordering and the first song in the second random ordering, resulting in an immediately repeated instance of that song. You decide that a solution that utilizes repeated random orderings with one additional check to make sure no song is ever played twice in a row is a sufficiently good solution to the problem.

You decide to write a tool that reads in a list of songs played while in "random" mode and checks it against the rules described above in order to flag obviously illegal playlists. It's difficult to judge whether a shuffle looks "random," so you simply check to make sure every song gets played once (in some order) before the songs start being played for the second time (and they all get played a second time before they start being played for the third time, etc.) and that the last song of one iteration is not the same as the first song from the next iteration, thus preventing back-to-back plays.

Details of the Input

The input will consist of multiple cases and begin with a line containing the number of cases to follow.

Each input case will begin with a line containing positive integers *m* and *s* ($2 \le m \le 1000$ and $2 \le s \le 1000$) representing the number of known music files (numbered 1 through *m*) and the number of songs played during a session, followed by *s* song identifiers in the range [1, *m*] in the order in which the *s* songs were played.

Details of the Output

For each case, print the case number and whether or not the playlist was generated according to the rules described above in the format shown in the Sample Output.

Sample Input

```
3
3 7 1 2 3 1 2 3 1
4 10 1 2 3 3 4 1 2 4 1 2
3 6 1 2 3 3 2 1
```

Sample Output

```
Case 1: Legal playlist
Case 2: Illegal playlist
Case 3: Illegal playlist
```

Problem E *Hedge Mazes*

Hedge mazes are constructed from tall, closely spaced shrubs that can form a wall-like structure. One way of constructing a hedge maze is to purposely grow these shrubs according to a pattern that results in the empty spaces forming pathways from an entrance point to an exit point. Another way of constructing them is to build a large rectangular area completely packed with these shrubs and then carve pathways through it from an entrance point to an exit point. This is the option you've chosen for constructing the *A-maze-ing Hedge Maze Marvel Attraction of Awesomeness* attraction in your amusement park.

You've provided the dimensions of your rectangular shrub area to your employees and created a contest where they can submit their designs and if their design is chosen they get a $0.25/hour raise! The final design will be chosen based on a number of criteria measuring the interestingness of the maze, but *first* you need a way of weeding out all the illegal mazes, so you decide to write a program to help you with this task.

The rules of legal hedge mazes are fairly simple:

- The maze is actually a grid of squares, each of which may either keep its original shrubs or have its shrubs removed.
- There must be exactly one entrance and exit, and they must be on the outside border (for obvious reasons).
- There must exist *at least* one path from the entrance to the exit (for obvious reasons). When the customer reaches the exit, they immediately leave the maze (they can't keep going or go back another path).
- There will be an actual wall around the outside of the maze, so you can use paths that remove shrubs around the outside border of the described maze area without worrying about the customers wandering off as though they had found the exit.
- Along *at least* one path from the entrance to the exit, there must exist *at least* one "choice" for the customer to make (otherwise it would be quite boring). The choice doesn't have to be *convincing*, there merely has to be a point at which they could decide to go one of two ways (not counting turning around). This includes a choice that can be made *at* the starting position to head in one direction or another if such an option exists.
- Any spot in which the shrubs are removed must be connected to the entrance via some path that does not go through the exit. That is, do not chop down shrubs at a particular location if it is impossible to reach that location. We don't like wasted effort and we've already built this huge rectangle of shrubs.
- We want this attraction to be fun even if it's raining, so we need to be able to place a (huge) tarp over the entire maze, but we don't want the tarp to droop down and hit people on the head (lawsuits are no fun!). So, at every intersection of four of the squares in the grid, there must be shrubs in one of the four squares sharing that corner to help support the weight. Do not worry about the extreme outside edge – there is a wall there which will support the tarp.

Given hedge maze proposals from your employees, determine whether they follow these rules.

Details of the Input

The input will consist of multiple cases and begin with a line containing the number of cases to follow.

Each input case will begin with a line describing the height and width of the rectangular shrub region (in “squares” as described above), each no greater than 10. There will then follow the description of a proposed maze design which may contain the following characters:

- X: indicates that the shrub is still there and this position cannot be used as part of a path
- +: indicates that the shrub has been removed
- s: indicates that the shrub has been removed and this is the entrance point of the maze
- e: indicates that the shrub has been removed and this is the exit point of the maze

Details of the Output

For each case, print the case number and whether or not the proposed maze is valid according to the rules above using the formatting provided in the Sample Output.

Sample Input

```
8
6 6
XX+sXX
XX+XXX
++++++
+XXXX+
+X++++
+XeXXX
6 6
XX+sXX
XX+XXX
++++++
+XXX++
+X+++X
+XeXXX
6 6
XX+sXX
XX+XXX
++++++
XXXXX+
+X++++
+XeXXX
6 6
XX+sXX
XX+XXX
XX++++
XXXXX+
XX++++
XXeXXX
2 5
XX+XX
s+++e
3 3
+++
sXe
+++
3 3
+++
sXe
+X+
1 3
sXe
```

Sample Output

```
Case 1: Valid
Case 2: Invalid
Case 3: Invalid
Case 4: Invalid
Case 5: Valid
Case 6: Valid
Case 7: Invalid
Case 8: Invalid
```

Problem F — *Image Resizing*

There are many algorithms available for resizing images to make them smaller. Most of them involve removing pixels proportionately throughout the image – if the resulting image needs to have half the width, just delete every other column. This approach to resizing results in an image that is (roughly) evenly scaled in each dimension.

In this problem, we consider a different approach that involves repeatedly finding and removing the "least interesting" horizontal or vertical "path" through an image. For our purposes, an image is simply a 2D array of integers that represent pixels, and our images always begin as squares and end as smaller squares.

A *horizontal path* must begin at some cell in the leftmost column and move to the right, up-right diagonal, or down-right diagonal until it reaches the right-most column.

A *vertical path* must begin at some cell in the topmost row and move down, down-left diagonal, or down-right diagonal until it reaches the bottom-most row.

The *interestingness of a path* is the sum (for all pixels in the path) of the absolute value of the differences between the pixel and its neighboring pixels in the direction perpendicular to that of the path to which it belongs. If you are constructing a horizontal path, then compare each pixel to the pixel above and below it; if you are constructing a vertical path, then compare each pixel to the pixel to the left and right of it. For pixels on the border which are missing one of these pixels to compare with, simply double the absolute value of the difference between it and the appropriate pixel which isn't missing. This should be made clear in the following example where we reducing a 4x4 image to a 3x3 image. The least interesting path in each direction is highlighted.

Original Image | **Finding Horizontal Path** (using vertical deltas) | **Finding Vertical Paths** (using horizontal deltas) | **Transformed Image**

First, reduce from 4x4 to 3x4 or 4x3 (based on whether the least valued path is horizontal or vertical).

6	5	8	8
8	7	4	7
3	3	3	6
3	3	9	5

4	4	8	2
7	6	5	2
5	4	7	2
0	0	12	2

2	4	3	0
2	4	6	6
0	0	3	6
0	6	10	8

5	8	8
7	4	7
3	3	6
3	9	5

We choose to delete the least interesting vertical path because its interestingness is less than that of the least interesting horizontal path. Next, reduce the 3x4 image to 3x3 image (if we were shrinking to 2x2, we could consider resizing to either 3x3 or 2x4 at this point)

5	8	8
7	4	7
3	3	6
3	9	5

4	8	2
6	5	2
4	7	2
0	12	2

Cannot remove vertical path because the width is already 3

5	8	8
7	4	6
3	9	5

To shrink the image from *n* x *n* to *k* x *k*, we repeatedly find the least valued horizontal or vertical path through the image (if either dimension has reached *k* pixels then paths in the opposite direction are no longer considered. That is, if the width has reached *k*, then vertical paths are no longer considered because they would cause the width to drop below *k*). An image may have, for instance, lots of horizontal paths removed before a single vertical path gets removed (imagine a picture of a sunset on a beach with a lot of empty sky). If the least valued horizontal and least valued vertical path have the same value, choose the horizontal path. If there are multiple paths in the same direction with the same least value (in our first case above there are three distinct paths with a sum of 9), prefer the path which makes the leftmost decision as you move top to bottom in vertical paths or the path which makes the topmost decision as you move left to right in horizontal paths.

(Incidentally, this is a real approach to context-aware image resizing – it's well worth taking the time to check out the video at http://bit.ly/UAContestImageResizingVideo **after** the contest.)

Details of the Input

The input will consist of multiple cases and begin with a line containing the number of cases to follow.

Each input case will begin with a line containing two integers *n* and *k*, with *n* representing the height/width of the original square image and *k* representing the height/width of the final resized image (one integer), with $3 \le n \le 50$ and $2 \le k < n$. The following *n* lines each contain *n* positive integers no greater than 1000 which together describe the original image.

Details of the Output

For each case, print the case number and the final image as shown in the Sample Output.

Sample Input

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

Sample Output

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

Problem G *Queen Anne's Puzzling Riddle*

Queen Anne's riddle is typically posed as a series of statements. For example:

> Kittens are, but cats are not.
> Butter is, but margarine is not.
> Twitter and Facebook are, but MySpace is not.
> Busses are, but trains are not.

The point of this riddle is not to drive you crazy, but it may annoy you. The solution to which objects are acceptable according to Queen Anne is neither intuitive nor obvious, but we would argue that it is pretty silly.

Objects that are acceptable are those whose string representation contains a double letter (like the two adjacent 't' characters in "kittens"). Triple letters are extremely infrequent in English, but letters that are part of a triple letter chain (like the 's' characters in "governessship") are NOT eligible to be part of a double letter. Hence, the word "governessship" is not acceptable according to Queen Anne. This rule also holds for all other k-letter chains where $k > 2$.

Details of the Input

The input will consist of multiple cases and begin with a line containing the number of cases to follow. Each input case appears on a single line of the form:

$$n \ w_1 \ \ldots \ w_n$$

Each word w_i will contain up to 50 lowercase alphabetic characters and $0 < n \leq 100$. The words are not guaranteed to be real words in any language.

Details of the Output

For each case, print the case number and the number of words in the case that would be acceptable according to Queen Anne as shown in the Sample Output.

Sample Input

```
2
3 queen annes riddle
4 programming contest governessship noooope
```

Sample Output

```
Case 1: 3 of 3
Case 2: 1 of 4
```

Problem H *House Hunting*

You and a friend are looking to rent a house together in the Akron area. You agree that there are three important places that you want to be able to visit relatively quickly: the grocery store, the electronics store, and the mall. You disagree about the relative importance of having short driving distances to these, so you decide to each allocate 10 "importance points" amongst the three destinations and come up with a formula to find the house that best suits your collective needs.

The score of a house is equal to the sum of your individual dissatisfaction ratings for that house, and you agree (in advance) to rent the house with the least such score. The dissatisfaction rating of a particular house for a particular individual is equal to the sum of that individual's dissatisfaction ratings for the commutes to the three destinations. The dissatisfaction rating of a commute to a particular destination for a particular person is the product of the shortest distance to that destination (in miles) and the importance points assigned to the destination by that person. The more important points assigned to a destination, the unhappier a person will be with a longer commute.

Out of all the houses for rent near Akron, you've been able to find no more than 50 that you agree are equivalent in every way except for physical location. Phew! Now that the hard part is done, simply calculate which of those houses should be chosen based on the rules above.

Details of the Input

The input will consist of multiple cases and begin with a line containing the number of cases to follow.

Each input case will begin with two integers, a positive integer n <= 50 indicating the number of houses and a positive integer $k \leq 2000$ representing the number of roads connecting the houses to each other, connect houses to the three destinations, or connect the destinations to each other. The houses are numbered in the range [1, n] and the destinations are labeled with special characters in the set {'G', 'E', 'M'} representing the grocery store, electronics store, and mall.

The following two lines will represent the importance points assigned by each person to these destinations (one set of points per line) in the order shown above (G, then E, then M). The sum of the three integers on each of these lines will be exactly 10, and each individual value will be non-negative.

Finally, there will be k lines of the form

a b d

indicating that there exists a bidirectional road between locations a and b (each represented by integers if they represent houses or uppercase alphabetic characters if they represent one of the three destinations) of distance d ($1 \leq d \leq 10000$) miles. Each house will be able to reach each destination through some path.

Details of the Output

For each case, print the case number and the best house as shown below. If there are multiple best houses, choose the one with the lowest numeric representation.

Sample Input

```
2
3 5
8 2 0
3 3 4
1 2 4
2 3 3
3 G 5
3 E 3
3 M 2
2 6
2 3 5
5 5 0
1 G 4
1 E 6
1 M 9
2 G 5
2 E 1
2 M 4
```

Sample Output

```
Case 1: House #3
Case 2: House #2
```