Third Annual Ohio Wesleyan University Programming Contest

March 28, 2008

Rules:

1. There are **six** questions to be completed in **four** hours.
2. All questions require you to read the test data from standard input and write results to standard output. You cannot use files for input or output. **Do Not** include any prompts, other debugging information, or any other output except for exactly what is specified by the problem.
3. The allowed programming languages are C, C++ and Java
4. All programs will be re-compiled prior to testing with the judges' data.
5. Non-standard libraries cannot be used in your solutions. The Standard Template Library (STL) and C++ string libraries are allowed.
6. Programming style is not considered in this contest. You are free to code in whatever style you prefer. Documentation is not required.
7. All communication with the judges will be handled in the PC^2 environment.
8. Judges' decisions are final. No cheating will be tolerated.

Problem A: Pack the House

Companies like Ticketmaster sell tickets to events and need to ensure that no seat is sold twice. Additionally, people usually buy “groups” of tickets, and would like all tickets in the group to be adjacent.

In this problem, you will allocate groups of tickets to a fictional arena. The arena can be imagined as a rectangle with *R* rows, and *S* seats per row. Tickets will be requested as groups, each group requesting a number of tickets *N*. If a request is to be satisfied, the *N* tickets must be allocated to adjacent empty seats in the same row (people in groups don’t like to sit in front or behind their friends).

Seats are allocated in a first-come, first-served basis. As a request arrives, tickets are allocated to the *first* row (closest to the stage) with enough seats to hold all N members of the group. If there is no row with enough seats, the customer is told that the arena is too full to hold the group. We’re interested in finding out how many groups are rejected in a given situation.

Input

There will be several test cases. Each test case will start with values for *R*, *S*, and *G* (the number of groups) on the first line. These values will be non-negative (≤10,000), and values of *R*, *S,* and *G* equal to zero signify the end of the input.

If the values are positive, there will follow a line with *G* values on it, each line representing a request. Each request will list its value for *N* (the number of seats requested) on a separate line.

Output

For each test case, output a line of the form:

```
Case c: X of the G groups did not get tickets.
```

or

```
Case c: All groups got tickets.
```

depending on which situation applies. *c* is the case number (starting from 1), and *X* is the number of groups that could not be seated according to the algorithm described above

Sample Input:

```
4 5 6
1 2 3 4 4 3
10 10 10
1 1 1 1 1 1 1 1 1 1
0 0 0
```

Sample Output

```
Case 1: 1 of the 6 groups did not get tickets.
Case 2: All groups got tickets.
```

Problem B: Volleyball Scoring

In “traditional” volleyball scoring (sometimes called “side-out scoring”, used in the Olympics before the 2000 games when they changed to “rally scoring”), a team can only score a point when they are serving. If a team serves and wins, they add a point to their score. If a team serves and does not get a point, there is no change to the score, and the serve goes to the other team (who then has a chance to score points). These rules mean it is possible to deduce the score of a game based on the knowledge of who won the various serves.

Input:
The first line will be a positive number, *N*, indicating the number of input cases. There will then follow *N* lines. Each line will start with a positive number *S* (<= 100), indicating the number of serves in the test case. There will then follow *S* values. Each value will either be the letter A (indicating Team A won the point) or the letter B (indicating Team B won the point). Team A always starts out serving. The game will go to 15 points, but a team has to win by at least two points to win the game, so it is possible for scores to be higher than 15.

Output:
For each input case, output a line:

```
Match m: The score is x-y.
```

Where *m* is a number denoting the match (starting from 1), and *x* and *y* are Team A’s and Team B’s points, respectively, in the current game.

It is possible for a match to end (because a team has reached 15 points or more and is ahead by at least two points). If this happens, disregard any remaining data in the test case, and output the line:

```
Match m: Team A/B has won the match with a score of x-y.
```

(You’ll say which team of A and B has won, of course)

Sample Input:

```
4
10 A B A A A B B B A A
15 A A A A A A A A A A A A A A A
4 B A B A
```

```
5 B B B A B
```

Sample Output:

```
Match 1: The score is 4-2.
Match 2: Team A has won the match with a score of 15-0.
Match 3: The score is 0-0.
Match 4: The score is 0-2.
```

Problem C: Turkey Irish

"Turkey Irish" is a language similar to Pig Latin, where English words are replaced with a simple substitution. In Turkey Irish, a word is modified by placing the letters "ab" in front of every vowel (the vowels are A, E, I, O, and U). So, for example, the word "contest" would be replaced by "cabontabest". Your task is to write a program that converts English words into their Turkey Irish equivalent.

Input:
The first line of the input will be a number, *N*, indicating the number of test cases. Then *N* words will follow, one per line. All words will be entirely in lower-case, and have no punctuation or any other non-letter symbols. Each word will be at most 100 characters.

Output:
For each input word, output the corresponding Turkey Irish word on a separate line.

Sample Input:
```
5
contest
happy
saint
patricks
day
```

Sample Output
```
cabontabest
habappy
sabaabint
pabatrabicks
dabay
```

Problem D: Minglers

One problem with large parties is that people tend to clump into small groups. As a professional mingler, you've been hired to help bring different groups together into one large social group by standing adjacent to groups. The only problem is that the host of the party is concerned about his status and has requested that you do not create any group larger than the largest group already present (the one he is in). This means that you cannot increase the size of his group either (he's already in the largest group he can handle)

Imagine that the party is represented by a two-dimensional grid. A "P" in a position means that the position has a partygoer, a dash ("-") in a position means that the position is empty.

P	P	P	X	P
-	-	-	-	P
P	P	-	-	-
P	P	-	-	-
P	P	P	P	-

The spot "X" is initially empty, but if you were to stand there and mingle, you would join the two separate groups into one large group of 6 people (including yourself). This is the best way to minimize the total number of groups without making any group larger than the host's group of 8 people (in the bottom left). If you are adjacent to any two (or more) groups in any direction, including diagonally, you have merged them into one large group.

You are to determine the best place to position yourself to mingle groups together, under the constraint that no new group is larger than the current largest group. The best place is one that minimizes the total number of groups. If there are several positions that do this, you should position yourself such that the new group that is formed is as large as possible.

Input:

There will be several cases. The first line will be two integers *L* and *W*, representing the length and width of the room(*L* and *W* will both be ≤ 100). Values of *L* and *W* that are less than zero will represent the end of the input. Otherwise, there will be *L* lines of *W* characters. Each character will either be a 'P' (meaning that location is occupied by a partygoer), or a '-' (meaning that location is empty). There will be one space between each character. Each grid will always have at least one empty location for you to stand at without violating the constraints of the problem.

Output:
For each input case *c* (starting from 1), output a line of the form:
Case *c*: I can make there be *g* total groups by forming a group of size *s*.

Where *g* is the number of groups that remain after you join the party, and *s* is the size of the group you’re in after you join it.

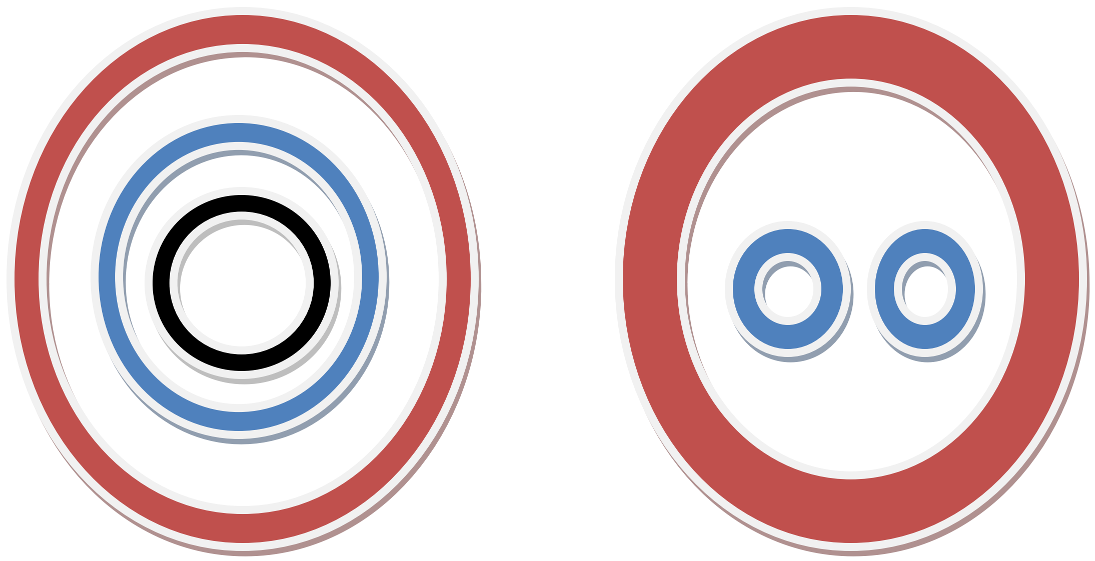
Sample Input:
```
5 5
P P P - P
- - - - P
P P - - -
P P - - -
P P P P -
4 6
P P P - P P
- - - - - -
P P - - - P
P P P - - -
5 5
P - P - P
P - - - -
P - P – P
P - - - -
P - - - -
4 5
P – P – P
P - - - -
- – P – P
- - - - - -
0 0
```

Sample Output:
```
Case 1: I can make there be 2 total groups by forming a group of
size 6.
Case 2: I can make there be 3 total groups by forming a group of
size 4.
Case 3: I can make there be 2 total groups by forming a group of
size 5.
Case 4: I can make there be 5 total groups by forming a group of
size 2
```

Problem E: Nested Ring Stacking

The ring stack is a child's toy where rings of different sizes are placed on top of each other:

A variation on this task is the nested ring stack, where rings are allowed to be placed inside each other (if they fit). The challenge now is to form the smallest possible stack, using all rings. The rings must either nest inside one another, or be placed such that one nested collection of rings is placed on top of another nested collection. (So you can't place two small rings side-by-side inside a large ring).



Acceptable Nesting: A smaller ring fits inside a larger one if its diameter is smaller than the diameter of the larger ring's hole	**Bad Nesting:** Each ring can have only directly enclose one ring.

Input:
The first line of the input will be a positive integer *N*, indicating the number of test cases. There will then follow *N* lines. Each line will start with an integer *R* ($1 \leq R \leq 100$) listing the number of rings in this test case. The line will then have *R* more entries, all separated by spaces, holding the diameters of the various rings, in inches. Each diameter will be a positive integer between 2 and 10,000.

Each ring has the same height, and is one inch thick. So the diameter of the "hole" in the ring is the diameter of the ring -2 (one inch on both sides) . A ring can fit inside another ring if its diameter is ≤ the size of the hole it is trying to fit into.

Output:
For each input case, output a line:
`Stack c is r rings tall.`

Where *c* is the current test case (starting from 1), and *r* is the minimum height (in rings) needed to stack all of the rings.

Sample Input:
```
3
5 2 3 4 5 6
6 12 2 8 6 4 10
5 3 3 3 3 3
```

Sample output:
```
Stack 1 is 2 rings tall.
Stack 2 is 1 rings tall.
Stack 3 is 5 rings tall.
```

Problem F- Necklace

You have a bunch of random beads lying around your house, and you'd like to use them to make a necklace by stringing the beads together. The beads are of different sizes and colors, and you'd like to construct an aesthetically pleasing necklace. The properties you've decided to enforce are:

1) Each pair of adjacent beads must match in either size or color.
2) No two adjacent beads can have the same size and color (so you can't have two adjacent beads that are exactly the same).
3) The necklace will form a loop, so the first and last beads are considered adjacent.
4) All beads must be used.

You're curious about whether you can construct a necklace from a given set of beads.

Input

There will be several test cases. Each test case will start with a number *T*, indicating the number of types of beads in the set. *T* will be at most 10, and a value of *T*=0 will denote the end of the input.

If *t* is greater than zero, there will then follow *T* lines. Each line will have three numbers separated by spaces:

C S Q

C is an integer ID for the color, *S* is an integer for the size of the bead (in millimeters), and *Q* is the quantity of the bead (how many of them we have). All values will be between 1 and 20, inclusive, and there will be at most 10 beads total.

Output:

For each test case *c*, output a line:

```
Case C: We can make a necklace from this set of beads.
```

or

```
Case C: We cannot make a necklace from this set of beads.
```

Sample Input:

```
4
1 5 4
1 10 4
3 10 4
3 5 4
2
1 5 4
2 3 4
0
```

Sample Output:

```
Case 1: We can make a necklace from this set of beads.
Case 2: We cannot make a necklace from this set of beads.
```