Seventeenth Annual
University of Oregon
Eugene Luks Programming Competition

Saturday, April 13, 2013

Problem Contributors

Jim Allen
David Atkins
Gene Luks
Chris Wilson

Food and prizes provided by Pipeworks

T-shirts provided by Emberex

GO DUCKS

If you have ever been to a UO football game, then you know that we have an exceptionally fit mascot. Every time that the UO team scores, the crowd counts while Ducky does as many push-ups as the UO team's total score at that point in the game. Of course, this means that with each scoring play, Ducky does even more push-ups.

For example, if the first UO score is a touchdown with a 2 point conversion, then the UO score would be 8, and Ducky would do 8 push-ups. If the next UO score is a field goal for 3 points, then the UO score would be 11 and Ducky would do 11 push-ups. If the next score is another field goal, Ducky would do 14 push-ups. If the final UO score is a touchdown with the kicked extra point, then 7 points would be added and now Ducky would do 21 push-ups. For the game Ducky would have done 8+11+14+21 = 54 push-ups.

For this problem, you are to calculate the final UO score for the game and the total number of push-ups that Ducky does during the game. The first line in input specifies the number of games for which you are to calculate totals. Each game is specified on a line as a list of the UO scoring plays. Each play is one of five strings: "TD", "TD+1", "TD+2", "FG", "S", which represent a touchdown with zero, one, or two extra points; a field goal; or a safety. The corresponding scores are 6, 7, 8, 3, and 2. For each game, you must output the number of push-ups and the total score in the format shown.

```
TD      6
TD+1    7
TD+2    8
FG      3
S       2
```

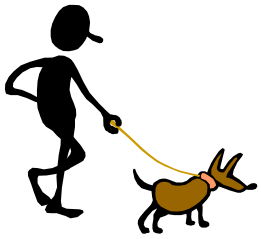
Sample Input

```
3
TD+2 FG FG TD+1
FG TD+1 TD+2 S TD
TD TD+1 TD+1 TD+1 TD+1
```

Sample Output

```
Game 1: final score 21, total push-ups 54
Game 2: final score 26, total push-ups 77
Game 3: final score 34, total push-ups 100
```

WALKING THE DOG

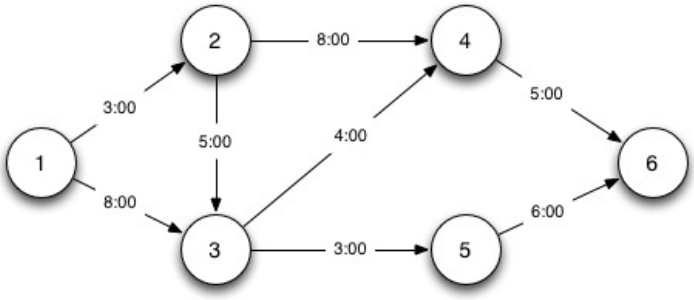


You have been hired to walk your employer's dog across the city each morning. The route through the city is expressed by a graph with n nodes, and you will start at position 1 and walk to position n. Since you get paid by the hour, you want to take the longest possible route. Also, your contract says that you get a bonus if you take a different route each day, so you want to know how many long routes there are.

The routes are described by a special kind of graph: an acyclic digraph. It has the following properties:

1. Nodes are numbered 1,2, …, n.
2. All edges are directed and specified by two nodes i and j with i<j.
3. There is at least one path from node 1 to node n.
4. The weight on an edge (i,j) corresponds to the expected amount of time it takes to walk from location i to location j, given in hours and minutes.
5. All edge weights are greater than zero.
6. Streets are one-way, you cannot reverse direction.

Consider the example graph to the right. The longest route from node 1 to node 6 takes 17 hours and 0 minutes. There are four paths that take that long: 1-3-5-6, 1-3-4-6, 1-2-3-5-6, and 1-2-3-4-6. For this problem, your program must determine the length of the longest path and the number of distinct paths of that length for each of a series of graphs.



The first line of input will be the number of graphs to solve. The first line of each subsequent graph problem will consist of two numbers *n*, the number of nodes and *e*, the number of edges. Following this will be *e* lines, one for each edge, and consisting of four numbers *i*, *j*, *h*, and *m*. The directed edge is from node *i* to node *j*. The weight of the edge is *h* hours and *m* minutes. For each graph your program must output the length of the longest path from node 1 to node *n*, and the number of distinct paths of that length in the form shown.

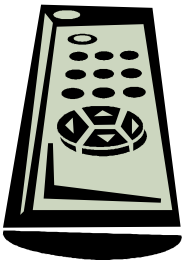
Sample Input

```
2
6 8
1 2 3 0
1 3 8 0
2 3 5 0
2 4 8 0
3 4 4 0
3 5 3 0
4 6 5 0
5 6 6 0
3 2
1 2 2 45
2 3 1 25
```

Sample Output

```
17 HOURS, 0 MINUTES, 4 PATHS
4 HOURS, 10 MINUTES, 1 PATHS
```

JUST A CLICK AWAY



Many devices connected to a TV have functionality which requires text entry, but don't have a full keyboard available. For example, my Blu-Ray DVD player has built in applications to play You Tube and Netflix videos, but only has a simple TV style remote controller. This makes it difficult to search for a topic or title. For text input, the player displays a virtual keyboard on the TV screen. In order to input some text, I must use the remote to move to each letter of input, select it, and move to the next. The remote control only uses four arrow buttons for the directions to move from one character to the next (one position up, down, left, or right), and a select button (SEL).

a	b	c	d	e	f	1	2	3
g	h	i	j	k	l	4	5	6
m	n	o	p	q	r	7	8	9
s	t	u	v	w	x	.	@	0
y	z	OK	SP	#	/	-	_	,

The virtual keyboard is shown to the left. The starting position is the upper left character 'a', and with the arrow buttons, you can move to a horizontally or vertically adjacent character. However, the keyboard does implement wrap around. For example, if the cursor is on 'a' and you press Up, then the cursor moves to the 'y'. Likewise, if the cursor is on 'g' and you press Left, the cursor moves to '6'. In some cases, wrap around could reduce the number of presses needed to get from one character to another. The actual space character in a string may be entered using the position labeled 'SP' on the keyboard. To complete the entry of a string, the special 'OK' position on the keyboard must be selected.

Inputting a text string may require many button pushes. For example, consider how to input the string "abc". The default character at the beginning of text selection is 'a' in the upper left corner. So just pressing the SEL button will select it. Then you must press Right to move to the 'b', and press SEL to select it. Then you must press Right to move to 'c' and press SEL to select it. Finally, to complete the text entry, you must move the cursor to the 'OK' character and press SEL. The quickest way to do that is to use wrap around and move Up from the 'c' to get to 'OK'. Thus, a total of 7 button presses is required (2 Rights, 1 Up and 4 SELs).

For this problem, you are to determine the smallest number of button presses that can be used to input a given text string. The cursor on the virtual keyboard always starts in the upper left at 'a'. The only movements are Up, Down, Left, and Right. The SEL button must be pressed to select a character. At the end of the string, the 'OK' pseudo-character must be selected. The first line of input is the number of text strings for which you are to calculate the smallest number of presses. Each subsequent line is a string formed from the characters shown on the keyboard. ('SP' on the keyboard represents the actual space character, which may appear in the strings.) Your program must output the smallest number of key presses for each string.

Sample Input	**Sample Output**
3	7
abc	7
ayz	25
deja vu	

THE ANSWER IS

You work on the Watson team at IBM. After its success on the English version of Jeopardy, your boss wants to develop other versions that can win on foreign language versions of Jeopardy. As the most recent employee, you are assigned to the team working on the Onami version, a recently discovered remote language spoken in the Amazon Rain Forest. Watson works by using an ontology to suggest possible interpretations of the question. The possible interpretations must be validated against the language grammar before determining which interpretation is most likely. Your unit is assigned to do this validation by taking a sequence of parts of speech and deciding if it is a valid sentence in Onami.

The parts of speech of Onami which will appear in input are:

Nouns	`nom`	Can be used for sentence subjects
	`acc`	Can be used for direct objects
	`dat`	Can be used for indirect objects
Verbs	`verb`	verb conjugations are handled with suffixes in Onami
Adverbs	`time`	adverb describing when
	`place`	adverb describing where
	`manner`	adverb describing anything other than when or where
Other	`conj`	word used to join two or more ideas (Conjunction)
	`adj`	word used to describe a noun (Adjective)

The general rules for the Onami grammar are:

1) Adverbs can go anywhere, but any `time` adverbs must precede any other type of adverbs, and any `place` adverbs must come after any other type of adverbs within the same simple clause.
2) The verb must be in the second position in a simple clause.
3) The subject (`nom`) must come immediately before or immediately after the verb.
4) Direct and indirect objects (`acc` and `dat`) are optional. If both occur within a simple clause, the indirect object must come after the direct object.
5) Adjectives (`adj`) must occur in front of the noun they describe with no intervening words except other adjectives. For purposes of rules 2 and 3, adjectives don't count as taking up a position in the clause separate from the noun's position
6) Each simple clause contains at most one subject, one direct object and one indirect object.
7) Subjects, direct objects or indirect objects may be a single noun, possibly preceded by adjectives, or a conjunction-separated list of adjective-noun groups. Such a compound noun counts as only occupying one position for purposes or rules 2 and 3.
8) Sentences in Onami can be either a simple clause as described in rules 1-7 or a conjunction-separated list of clauses.

The input will begin with a line with a single number $0 < n \leq 10000$. This will be followed with n problems, each on a line by itself. Each problem will consist of one or more parts of speech. For each problem in the input, your program should output either "accept" or "reject" on a line by itself. Printing "accept" means the proposed assignment of parts of speech is within the

grammar rules of Onami. Printing "reject" means the assignment of parts of speech violates the grammar rules of Onami.

Sample Input

```
15
time verb nom
place verb nom
manner verb adj nom
nom verb acc
nom verb time acc
nom verb acc time
nom verb dat acc
nom verb time place
nom verb time conj place
nom verb place time
nom conj adj nom verb dat
nom verb conj verb
time time verb nom
nom verb conj nom conj nom verb
```

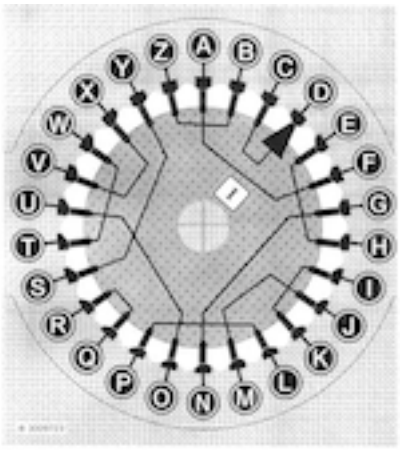
Sample Output

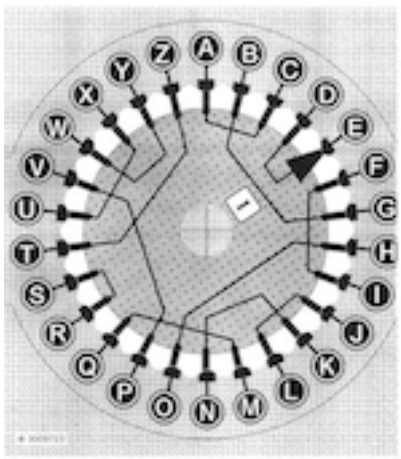
```
accept
accept
accept
accept
accept
accept
reject
accept
reject
reject
accept
reject
reject
accept
```

POCKET ENIGMA

This problem is based on the POCKET ENIGMA, a toy that is functionally equivalent to primitive military ciphering machines. The Pocket Enigma consists of a fixed outer circular disk, marked with the alphabet in clockwise order, and a replaceable inner rotor with 13 wires that pair up the letters for encryption/decryption.

The outer disk and two examples of inner rotors are shown on the reverse. The device is shown here with Rotor I in successive positions (see arrow) **D**, **E**.





The disk moves one notch clockwise *before* encrypting each letter of plaintext. Thus, with Rotor I starting in position **D**, "CIS" is encrypted to "APO".

In this problem, you are given a ciphertext (encrypted message) that used an unknown rotor but are in possession of a "crib" (a segment of the source plaintext). Your job is to determine where the crib appeared in the plaintext.

The first line of the input specifies the number ***n*** of ciphertexts to follow. Each of the following ***n*** lines consists of two strings of capital letters, of maximum lengths 50 and 10, respectively, separated by a single space:

`CIPHERTEXT CRIB`

The ***n*** ciphertexts would all have been created with different inner rotors, and *not those shown on the reverse*.
For each input, output the number ***m*** if the crib starts at the m^{th} character, There will be always be just one feasible position.

(Sample I/O on reverse)

Sample Input	Sample Output
4	
AHA A	2
KFOP CAT	1
ZAORQAS DOG	4
VFKHEY DUCK	3

The outer disk and two possible rotors for the Pocket Enigma