Computer Science Competition
State 2016
Programming Problem Set

I. General Notes

- Do the problems in any order you like. They do not have to be done in order from 1 to 12.

- All problems have a value of 60 points.

- 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.

- Your program should not print extraneous output. Follow the form exactly as given in the problem.

- A penalty of 5 points will be assessed each time that an incorrect solution is submitted. This penalty will only be assessed if a solution is ultimately judged as correct.

II. Names of Problems

Number	Name
Problem 1	Abigail
Problem 2	Benjamin
Problem 3	Carlos
Problem 4	Dara
Problem 5	Emiliano
Problem 6	Fujita
Problem 7	Guadalupe
Problem 8	Heng
Problem 9	Ignacio
Problem 10	Jasmine
Problem 11	Kamalika
Problem 12	Luna

1. Abigail

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

Abigail has just learned to simplify square root radicals and needs you to write a program for her so she can check her work. The rules she learned in the lesson were simple:

1. A square root radicand may not contain a perfect root factor.
2. A square root radicand may not be negative.

Her current homework has several expressions in the form A + B * sqrt(C), where A, B, and C can be any integers, positive or negative, but only A might be zero. The assignment is to simplify the radical in the same form.

The program is to input the initial values of A, B, and C, and then output the simplified values of A, B, and C, any of which could be zero.

Her teacher has guaranteed that all answers have possible values of A, B, and C, and that when B or C are equal to zero, it means there is no radical in the final solution.

For example, the data set 5 2 20 represents 5 + 2 * sqrt(20), which can be simplified as such:
5 + 2 * sqrt(4 * 5) = 5 + 2 * 2 * sqrt(5) = 5 + 4 * sqrt(5), resulting in the output values 5, 4 and 5.

For the data set 5 2 -20, the final output is 5 4i 5, with ***i*** representing the square root of -1.

Input - Several sets of three integer values, each set on one line. All three integers will be in the range -100...100, but only A might be zero. Each set of three integers A, B, and C represents the radical expression A + B * sqrt(C).

Output - Simplify each expression according to the rules listed above, and output the final values of A, B, and C. It is possible for any of the final three values to be zero.

Sample data:
```
5 2 20
5 2 -20
0 3 9
3 4 -25
0 9 3
```

Sample Output:
```
5 4 5
5 4i 5
9 0 0
3 20i 0
0 9 3
```

2. Benjamin

Program Name: Benjamin.java　　　　Input File: benjamin.dat

Benjamin has decided to expand the stack and queue process he just learned in class by adding some new processing features. Besides the normal push and pop, he adds low, high, and middle processes, either inserting a value or removing a value using these concepts. For working with stacks of digits, he decides on eight operations, which are as follows:

1. P - Pop a digit from the top of the stack.
2. P(X) - Push X onto the top of the stack.
3. L - Remove the lowest valued digit from the stack.
4. L(X) - Insert X immediately below the lowest valued digit in the stack.
5. H - Remove the greatest valued digit from the stack.
6. H(X) - Insert X immediately above the greatest valued digit in the stack.
7. M - Remove the digit at the middle* of the stack.
8. M(X) - Insert X so that it is positioned as the new middle* of the stack.

For queues of digits, he will use the same eight operations, as defined below:
1. P - Pop a digit from the front of the queue.
2. P(X) - Push X onto the back of the queue.
3. L - Remove the lowest valued digit from the queue.
4. L(X) - Insert X immediately in front of the lowest valued digit in the queue.
5. H - Remove the greatest valued digit from the queue.
6. H(X) - Insert X immediately behind the greatest valued digit in the queue.
7. M - Remove the digit at the middle* of the queue.
8. M(X) - Insert X so that it is positioned as the new middle* of the queue.

*NOTE: For a stack or queue with an even number of digits, Benjamin observes that the "middle" could be one of two different digits. He decides that, for his purposes, "middle" shall refer to the digit from the middlemost pair that is closer to the bottom of the stack or the front of the queue. For example, in the list of digits below, the "middle" digit is the 6 (not the 1), whether the digits are in a stack or a queue:

```
Bottom/Front --> [5, 3, 7, 6, 1, 9, 8, 7] <-- Top/Back
```

Input - An initial string of digits in the range 0-9, all on the first line, with single space separation. The values in this list are to be used to create both a stack and a queue by adding each digit in order, from left to right. On the next several lines will be commands that will either affect the stack or the queue, indicated by either the letter S or Q, followed by one or more commands to be evaluated left-to-right as defined above. When an L command is given, you may assume that the stack or queue contains only 1 digit that is strictly less than all other digits. Similarly, you may assume that when an H command is given, the stack or queue contains only 1 digit that is strictly greater than all others. No commands will result in attempting to remove a digit from an empty stack or queue.

Output - After processing each line of commands, print the modified stack or queue, with the corresponding letter (S or Q) followed by its list of contents, as shown below. Stacks should be printed with the bottom of the stack to the left and the top of the stack to the right. Queues should be printed with the front of the queue to the left and the back of the queue to the right.

Sample data:
```
5 3 7 6 1 9 8 7
S P
Q P(4)
Q L M(2)
S L(0) H(2)
S L M
```

Sample Output:
```
S [5, 3, 7, 6, 1, 9, 8]
Q [5, 3, 7, 6, 1, 9, 8, 7, 4]
Q [5, 3, 7, 6, 2, 9, 8, 7, 4]
S [5, 3, 7, 6, 0, 1, 9, 2, 8]
S [5, 3, 7, 1, 9, 2, 8]
```

3. Carlos

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

Carlos has decided to write a program to simulate the poker game Five Card Draw, where he stores each card using a unique number, with the diamonds numbered from 1-13, hearts from 14-26, spades from 27-39, and clubs from 40-52. Card #1 is the Ace of Diamonds, and card #13 is the King of Diamonds, #14 is the Ace of Hearts, and so on. Card #52 is the King of Clubs.

His program will input the ten numbers representing the cards alternately dealt to players A and B, and will tell him the best hand for each player, and which player wins the hand. According to the rules of poker, the hand rankings are as follows, in order of best to worst.

FOUR OF A KIND - all four of the same card from each suit, like four 2s or four Kings.

FULL HOUSE - three of one kind and two of another, like three 8s and two Aces.

FLUSH - five cards, all of the same suit, like five spades or five diamonds, in no particular order.

STRAIGHT - five sequentially numbered cards (dealt in any order) of at least two different suits, like Ace, 2, 3, 4, 5, or 10, Jack, Queen, King, Ace, or any sequence somewhere in the middle of the suit. With the exception of the Ace, which can count as either a "low" card (less than 2) or a "high" card (greater than King), wraparound sequences, like Queen, King, Ace, 2, 3, do not count as a straight.

THREE OF A KIND - three of one kind, like three 7s or three Jacks, but not a full house.

TWO PAIRS - 2 different pairs of the same kind, like two 5s and two 9s.

PAIR - one pair of the same kind, like two 4s or two Kings.

DUD HAND - Nothing good, for sure.

Input: Several poker hands of ten cards each, with the 1st, 3rd, 5th, 7th and 9th cards going to player A, and the others (2nd, 4th, 6th, 8th and 10th) to player B. The ten cards will all be on one line, separated by single spaces, according to the description above.

Output: In all uppercase letters and separated by " - " as shown in the sample output below, print the best poker hand for player A, the best poker hand for player B, and "PLAYER x WINS" (where x is the player with the winning hand).

Sample input:
```
32 18 6 44 19 7 22 21 23 38
1 34 14 13 27 26 29 39 42 52
```

Sample output:
```
THREE OF A KIND - PAIR - PLAYER A WINS
FULL HOUSE - FOUR OF A KIND - PLAYER B WINS
```

4. Dara

Program Name: Dara.java Input File: dara.dat

Dara has just learned about derivatives, is very confused, and needs your help with her homework, which for now only works on simple powers (no negative or fractional exponents).

She barely understands the rules of finding a derivative, which are:

- If the expression is a constant value, the derivative is zero.
- If the expression is any power in the form AX^N, where is A is a positive or negative integer, and N is any positive integer, the derivative is equal to A times N, times X raised to the power of N-1, or ANX^(N-1).
- The derivative of a sum of multiple operands is the sum of the derivatives of each operand.

For example, the derivative of 8 is just 0, since 8 is a constant value. The derivative of X is 1, since A is implied to be 1, and the power of X is implied to be 1. N-1 is zero, and so X^0 is 1, therefore 1(A) times 1(N) times 1(X^0) is equal to 1.

The derivative of X^5 is 5 times 1 times X^4), or 5X^4.

The second derivative is simply the derivative of the first derivative. For example, in the case of X^5, if the first derivative is 5X^4 (as shown above), then the second derivative is 20X^3 (i.e., (5 times 4) times X^(4-1)).

Input - Several polynomial expressions, one per line. Each polynomial consists of one or more terms separated by " + " or " – ", in descending order of degree, with X as the only variable. Terms with a degree of zero (i.e., "AX^0") will be expressed as "A". Terms with a degree of one (i.e., "AX^1") will be expressed as "AX". All numeric values (coefficients or constants) will be positive integers. There will be no fractional input values.

Output - The first and second derivative of each expression, with " : " separating the two derivative expressions. Coefficient values of 1 or -1 are not allowed, and only one sign is allowed between terms. For example, 2X + -6 is not correct and should be expressed as 2X - 6. Unless the overall derivative is 0, any individual terms that evaluate to 0 should not be displayed. For example, 9X^2 - 5 + 0 should be expressed as 9X^2 - 5. There will be no fractional output values.

Sample data:
```
8
X
X^5
5X^4
3X^3 - 5X + 8
```

Sample Output:
```
0 : 0
1 : 0
5X^4 : 20X^3
20X^3 : 60X^2
9X^2 - 5 : 18X
```

5. Emiliano

Program Name: Emiliano.java   Input File: emiliano.dat

Emiliano loves morphological derivation and the most fundamental and fascinating to him is compound words. He is building a program to identify every compound word in a given dictionary! Given a list of words in this dictionary and a list of test words, determine which of the test words are compound words. A test word is considered a compound word if it is completely made up by exactly two dictionary words. But if a single test word can qualify as a compound word in multiple ways, it is known as a multi-compound word. For example, the word NOTABLE is a multi-compound word:

```
NOTABLE is a compound word of NO and TABLE.
NOTABLE is a compound word of NOT and ABLE.
```

Thus, the test word notable will count as two compound words.

For each of the test words, determine if it is a compound or multi-compound word made up by a pair of dictionary words. For each way that the test word qualifies as a compound, increment the count of the number of compound words. In the case of multi-compound words, each compound pairing is counted.

Input: The first integer will represent the number of data sets to follow. The next line will be an integer, a, representing the number of dictionary words. The next a lines are single words (with no spaces) representing the dictionary words, in an arbitrary order. The following line will be an integer, b, representing the number of test words. The next b lines are single words (with no spaces) representing the test words.

Output: For each test word, determine if it is a compound word, and if it is, add one to the count. For each data set, print the number of compound words (and multi-compound word pairings).

Assumptions: Language is very complex, and thus the number of words in the dictionary can be very, very large. There is a 3 minute execution time limit and a 1GB space limit. Note the empty string is not a dictionary word. Prefix dictionary word and suffix dictionary word need not be unique.

Sample Input:

```
3
3
car
Bar
STAR
5
barcar
CARSTAR
BarStar
Starsnbars
Nobarcar
3
a
aa
beans
4
aaa
abeans
abeansa
beansa
2
no
yes
1
yesno
```

Sample Output:

```
3
4
1
```

6. Fujita

Program Name: Fujita.java   Input File: fujita.dat

Fujita is fascinated with number bases, specifically with the log values of different numbers in different bases.
He decides to experiment by writing a program that will output bar graphs using this idea, but needs your help. He wants the program to produce a horizontal bar graph whose length represents the log of a value N given a certain base B. For example, for the value 1000 in base 2, the bar produced will have ten "*"s, since the log, base 2, of 1000 is approximately 10.

One way Fujita thinks about finding the log value of a number is to consider the powers of 2 close to 1000, which are 512 (2^9), just less than 1000, and 1024 (2^10), just greater than 1000. For the purposes of this exercise, he decides to use the log of the value just greater than N.

For another value, 3671, and a base of 4, the bar graph produced contains six stars, indicating the approximate log value of 6, base 4, of the value 3671. He understands that this also is an approximation, since 4 to the power of 6 is actually 4096, but it is the closest log value that just exceeds 3671. The power of 4 just less than 3671 is 4^5, or 1024.

Input - Several pairs of positive integer values N and B (N>B), with N having up to 20 significant digits, and B having a value no greater than 10.

Output - For each pair of values, output the number of stars ("*") in a horizontal bar graph that represents the log value of N, base B, as described and demonstrated above.

Sample data:
```
1000 2
3671 4
9182736453 5
```

Sample Output:
```
**********
******
***************
```

7. Guadalupe

Program Name: Guadalupe.java Input File: guadalupe.dat

Many rumors have been flying around at school. Guadalupe has been trying to figure out how certain people have been keeping up with the gossip. To do this, Guadalupe has set up a camera by the water cooler and noticed that when two or more people are at the water cooler at the same time, they exchange all the gossip they have heard that day with each other. Guadalupe wants your help to model the findings from the time people are at the water cooler to the information they know.

Thanks to Guadalupe's camera, she knows exactly who came and went from the water cooler, and what times each person arrived and left. When two or more people are at the water cooler, they tell each other all of the gossip that they knew at the start of the day as well as all of the gossip they have been told by others during the day. For example, if Amelia sees Juan at the water cooler in the afternoon, she will tell him everything she knew at the start of the day, plus everything she learned from José earlier in the day. Likewise, Juan will tell Amelia everything he knows or has learned up until that point as well. At the end of the day, Amelia will know José's, Juan's, and her own gossip. Juan will know Amelia's, Jose's, and his own gossip. But Jose will only know Amelia's and his own gossip.

Now, given the information from Guadalupe's camera, determine who, at the end of the day, has heard whose gossip, assuming that all gossip is exchanged at the water cooler and only at the water cooler.

Input: The first integer will represent the number of data sets to follow. The first line of each data set will consist of an integer, N, representing the number of people Guadalupe caught on camera during the current day. Each of the next N lines will begin with the firstname (containing no spaces or punctuation) of the person that the camera saw, followed by a pair of times (represented in 24-hour time). Each of these pairs of times represents when the person arrived at the water cooler and when they left, respectively.

Output: Each data set should print a line of output for each person Guadalupe saw on camera in the order that they first arrived at the water cooler for the day. This string should read: "<NAME> has heard gossip from: <GOSSIPS>" where <GOSSIPS> is the alphabetized list of names whose stories <NAME> has heard either directly or indirectly throughout the day. Note: A person, by default, has heard their own gossip. Therefore, each person's name should be included in their own list of<GOSSIPS>. Each data set should be followed by a blank line.

Assumptions: All times expressed will be valid times. All meetings will have an end time at least one minute later than their start time. All times given for a person's schedule will be in chronological order and will not wrap around the day (meaning the earliest time will be midnight -- 0:00 -- and the latest will be 23:59). Times are represented up to but not including the end-time. Thus, if someone arrived at 14:01 and left at 14:02 and someone arrived at 14:00 and left at 14:01, these two people did not overlap at the water cooler and did not exchange information.

Sample Input:
```
3
4
AMELIA 1:20 1:25 2:35 2:50
JOSE 2:30 2:35
JUAN 2:34 2:38
JESUS 2:37 2:40
4
NOLAN 1:20 1:25 2:35 2:50 23:50 23:59
DULCE 2:30 2:45
DESERAY 2:34 2:38
PEDRO 2:37 2:40
5
SHARON 8:30 8:50
IRFAN 8:45 9:10
STEPHANIE 9:00 9:23
KARIBU 9:20 9:30
MURICA 9:29 9:50
```

Sample Output:

```
AMELIA has heard gossip from: AMELIA JESUS JOSE JUAN
JOSE has heard gossip from: JOSE JUAN
JUAN has heard gossip from: AMELIA JESUS JOSE JUAN
JESUS has heard gossip from: AMELIA JESUS JOSE JUAN

NOLAN has heard gossip from: DESERAY DULCE NOLAN PEDRO
DULCE has heard gossip from: DESERAY DULCE NOLAN PEDRO
DESERAY has heard gossip from: DESERAY DULCE NOLAN PEDRO
PEDRO has heard gossip from: DESERAY DULCE NOLAN PEDRO

SHARON has heard gossip from: IRFAN SHARON
IRFAN has heard gossip from: IRFAN SHARON STEPHANIE
STEPHANIE has heard gossip from: IRFAN KARIBU SHARON STEPHANIE
KARIBU has heard gossip from: IRFAN KARIBU MURICA SHARON STEPHANIE
MURICA has heard gossip from: IRFAN KARIBU MURICA SHARON STEPHANIE
```

8. Heng

Program Name: Heng.java   Input File: heng.dat

Heng just learned about CRC (cyclic redundancy check) in networking class and has looked up what Wikipedia says about it, which is:

> **"A cyclic redundancy check (CRC) is an error-detecting code commonly used in digital networks and storage devices to detect accidental changes to raw data. Blocks of data entering these systems get a short check value attached, based on the remainder of a polynomial division of their contents."**

Essentially, when a message is sent, two other things are sent: a key, and a checksum. The receiver performs an algorithm on the message using the key and determines if the result of that algorithm matches the checksum that is sent.

Heng decides to find a simple way of doing this, just for practice, and discovers this simple algorithm. First he starts with a binary string represented by the hex value 098, which represents some data, and decides on a key value of 5. The equivalent binary strings for these two hex values are:

000010011000 and 0101

The process he discovers uses long division modulo 2 in binary, but instead of subtracting it uses the bitwise XOR process to reduce the message. When XOR is applied in a bitwise manner, if the two bits being considered are the same, the resulting bit is zero, otherwise the result is 1. The process for the data example given would look like this:

`000010010000` divided by `0101`

```
000010011000
    101
    00111
      101
      0100
       101
       00100
         101
          01
```

He carefully notes that only the remainder after the XOR process is important, and that the final remainder should be expressed using the number of significant digits of the key, minus 1.
In the example to the right, since the key, 0101, has three significant digits (the leading zero does not count), the final remainder must be expressed in two SD, or 01.

Input - Several pairs of hex values, each pair on one line, separated by a single space.

Output - For each pair of input values, calculate and output the final checksum based on the algorithm described and demonstrated above.

Sample data:
```
098 5
F 3
AC 5
71C A
8E6B 6
```

Sample Output:
```
01
0
11
000
01
```

9. Ignacio

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

Ignacio was looking through a math puzzle book and encountered a very cool process to make a magic square. Magic squares, he learns, always have the same sum for any row, column, or main diagonal. The process described in his book works with squares of odd length and width, like 3 by 3, or 5 by 5, and so on. He needs your help to write a program to simulate this process so that he can identify any value in the square, given the row and column.

Here is an example of a magic square that is 3 by 3.

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

The top row sum is 4+9+2, or 15. The sum for the leftmost column is 4+3+8, also 15. Even the diagonals have a sum of 15: 4+5+6 and 8+5+2. "This is fascinating!", he thinks to himself.

The rules given in the puzzle book to build any size magic square, as long as it is an odd length and width, are as follows:

- Start with the value 1 in the middle column of the bottom row of the square.
- When possible, place the next number in the empty box down one row and across one column to the right.
- If the move described above in rule 2 is not possible because the box down one, right one is occupied, put the next number directly above the current box.
- If you have reached the bottom row, but not the last column, the next number goes to the top of the next column over.
- If you have reached the last column on the right, but not the bottom row, the next number goes to the first column of the next row down.
- If you have reached the bottom right corner of the square, put the next number directly above the current box.

Ignacio decides to build the 3 by 3 size box, shown above, following the rules.
1. He follows Rule 1 to place the 1 in row 3, column 2.
2. Since he's at the bottom row of the box, but not the last column, he follows Rule 4 and puts the 2 at the top of the next column.
3. Since he's now at the right column of the box, with no place to go "down one, across one", he follows Rule 5 and puts the 3 at the leftmost column of the next row down.
4. Since the box "down one, across one" is occupied by the 1, he puts the 4 directly above the 3, following Rule 3.
5. He puts the 5 and 6 using the "down one, across one" process described in Rule 2.
6. The 7 is placed using Rule 6, the 8 using Rule 5, and the 9 using Rule 4.

He now sees how the rules work, and can identify any value, given the row and column, such as row 3, column 2, which contains the 1, and row 2, column 1, which contains the 3.

Input - Several sets of three integer values, N, R, and C, all on one line, with single space separation.

Output - After building the magic square of size N, according to the rules listed and demonstrated above, output the value in row R, column C, of the completed magic square.

Sample data:

```
3 3 2
3 2 1
5 1 1
7 7 5
9 2 9
```

Sample Output:

```
1
3
11
10
25
```

10. Jasmine

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

Jasmine loves to mess around with her friends' names by changing around the letters to make crazy sounding new names. One thing she tried recently, even writing a program for it, was to divide each name into three parts, then finding the longest part and putting that first in all caps, followed by the other two parts, the first lowercased and reversed, and the other part uppercased and alphabetized.

She decided to divide each full name length by 3, and use that value as the length of the first two parts, which means the last part could possibly be longer than first two, but that was OK with her.

For example, since "Abigail" has 7 letters, the first two parts - "Ab" and "ig" - will be 2 characters long, and "ail" will be the third part. Since "ail" is the longest part, which she puts it first in all caps, followed by "ba", the reversed and lowercased next part, and finally "GI", the alphabetized and uppercased final part, making a new crazy name of "AILbaGI".

For Carlos, all three parts are the same length, so she just keeps them in order, capitalizing the first part, doing the "reverse/lowercase" thing with the second part, and the "alphaOrder/uppercase" thing with the last part, producing "CAlrOS" as his new crazy name.

Input - Several names of friends, each on one line, as shown below.

Output - For each name, output the three parts, with single space separation, followed by " ==> ", and then the new crazy name according to the criteria specified above.

Sample data:
```
Abigail
Benjamin
Carlos
```

Sample Output:
```
Ab ig ail ==> AILbaGI
Be nj amin ==> AMINebJN
Ca rl os ==> CAlrOS
```

11. Kamalika

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

Kamalika has always loved numbers and how different operations act on them. She has decided to do some simple research and needs your help writing the program to help with this. Her idea is to see how the resulting values from seven basic operations behave on two real number values A and B, specifically the resulting order in ascending value of the seven results.

For example, using the values 3 and 4, the sum is 7, difference is -1, product is 12, modulus is 3, quotient is 0.75, power A^B is 64, and B^A is 81. Those values in ascending sorted order are:

```
|Dif -1.00|Div  0.75|Mod  3.00|Sum  7.00|Prd 12.00|B^A 64.00|A^B 81.00|
```

However, for values less than 1, like 0.3 and 0.4, the order changes, like this:

```
|Dif -0.10|Prd  0.12|Mod  0.30|A^B  0.62|Sum  0.70|Div  0.75|B^A  0.76|
```

Input - Several pairs of real number values A and B, each in the range **`-5 <= value <= 5`**, excluding zero.

Output - Output the ascending order result of seven different mathematical operations as shown in the examples above, formatted EXACTLY as shown, with appropriate labels, followed by the corresponding resulting values rounded to the hundredths place. All results are guaranteed to be real, and to fit within the formatted spaces as shown in the examples.

Sample data:
```
3 4
.3 .4
3 .4
.3 4
-3 4
-4 3
```

Sample Output:
```
|Dif -1.00|Div  0.75|Mod  3.00|Sum  7.00|Prd 12.00|B^A 64.00|A^B 81.00|
|Dif -0.10|Prd  0.12|Mod  0.30|A^B  0.62|Sum  0.70|Div  0.75|B^A  0.76|
|B^A  0.06|Mod  0.20|Prd  1.20|A^B  1.55|Dif  2.60|Sum  3.40|Div  7.50|
|Dif -3.70|A^B  0.01|Div  0.08|Mod  0.30|Prd  1.20|B^A  1.52|Sum  4.30|
|Prd-12.00|Dif -7.00|Mod -3.00|Div -0.75|B^A  0.02|Sum  1.00|A^B 81.00|
|A^B-64.00|Prd-12.00|Dif -7.00|Div -1.33|Sum -1.00|Mod -1.00|B^A  0.01|
```

12. Luna

Program Name: Luna.java Input File: luna.dat

Luna has been studying the classic data structure, the binary search tree, and has learned that it is great for storing information that allows efficient searching and/or retrieval of the data. It is designed as a collection of nodes, each with a single left node/child and a single right node/child, where at each node, the **left child is always less than or equal to the parent value and the right node is always greater than the parent value**.

She has learned that to find any item in a well-balanced tree has a best case of O($\log_2$N) retrieval, a very quick process! Despite this optimistic thinking, Luna is dismayed to learn that binary search trees have a worst case retrieval scenario of O(N). This can happen if you have a binary search tree of the form shown here. To find the E in the search tree, we have to visit all 5 nodes, not very efficient.

```
A
 \
  B
   \
    C
     \
      D
       \
        E
```

Fortunately, she learns that it is possible to guarantee that binary search trees never end up in this unfortunate form, so that there is a better chance of having this quick O($\log_2$N) retrieval from all binary search trees.

This process is called *search tree balancing*, which Luna wants to learn about, and needs your help. One of these techniques is to use rotations. **Rotations** take a specific child node, turn it into a parent, and rotate all of the dependencies necessary around that node to keep the binary search tree property (as described above). There are 4 different kind of rotations that can work.

The SINGLE LEFT ROTATION
A tree is right heavy if its right subtree is larger than the left subtree in height by two or more AND the height of its right subtree's right subtree is greater than its right subtree's left subtree. In the example below, the height of A's right subtree is 2, and is zero for the left subtree. Furthermore, the height of B's right subtree is 1, and its left subtree zero. This fits the criteria, and she can perform a **single left rotation** about A to make B the new root, which looks like this:

From this:

```
A
 \
  B
   \
    C
```

To This:

```
  B
 / \
A   C
```

The SINGLE RIGHT ROTATION
If the tree is left heavy - its left subtree is larger than the right subtree in height by two or more, AND the height of its left subtree's left subtree is greater than its left subtree's right subtree - then she can perform a single right rotation about C to make B the new root, which looks like this:

From this:

```
    C
   /
  B
 /
A
```

To This:

```
  B
 / \
A   C
```

The DOUBLE LEFT ROTATION

If Luna has a situation where a tree is right *heavy* - its right subtree is larger than the left subtree in height by two or more -- AND the height of its right subtree's left subtree is greater than its right subtree's right subtree, then we must perform a double left rotation, which would look like this:

From this:

```
A
 \
  C
 /
B
```

To This:

```
A
 \
  B
   \
    C
```

To This:

```
  B
 / \
A   C
```

This is a single right handed rotation about C to move B to the right child of A and then a standard single rotation about A to make B the new root.

The DOUBLE RIGHT ROTATION

If we are in the situation that a node is right *heavy* -- its right subtree is larger than the left subtree in height by two or more -- AND the height of its right subtree's left subtree is greater than its right subtree's right subtree, then we must perform a double right rotation, which would look like this:

From this:

```
  C
 /
A
 \
  B
```

To This:

```
A
 \
  B
   \
    C
```

To This:

```
  B
 / \
A   C
```

This is a single left handed rotation about A to move B to the left child of C, **and then** a single left rotation about C to make B the new root.

Armed with these rebalancing techniques, Luna needs your help to write a program that will detect when these rotations are necessary about the root node of a binary search tree. Given a string of words to put in binary search tree (using alphabetical order to determine where they should go in the tree), display when the input of a node makes the tree unbalanced at the root node. You should input the nodes into the binary search tree in the order they are given to determine when the tree becomes unbalanced. When the tree becomes unbalanced, print which word it was that caused the imbalance, and which rotation should be used to correct it. If, in the order given, the words can all be placed into the binary search tree and remain completely balanced, then print that the tree is balanced.

Input: The first integer will represent the number of data sets to follow. Each line will be a list of words to be placed into the binary search tree.

Output: Each data set should return the line "`UNBALANCED AFTER  <unbalancing input>, A <rotation type> AT ROOT TO REBALANCE.`" If all of the input is consumed and the root of the tree remains balanced, then output the phrase "`TREE IS BALANCED.`"

Sample Input:

```
6
A B C
C B A
B A C
C A B
ALLEGORY BREAD CHARMANDER DANDELION ELEPHANT
MAYFLOWER HAIR RETINA ECLIPSE IGUANA QUACK RODEO OCTOBER SOUP
```

Sample Output:

```
UNBALANCED AFTER C, NEED SINGLE LEFT ROTATION AT ROOT TO REBALANCE.
UNBALANCED AFTER A, NEED SINGLE RIGHT ROTATION AT ROOT TO REBALANCE.
TREE IS BALANCED.
UNBALANCED AFTER B, NEED DOUBLE RIGHT ROTATION AT ROOT TO REBALANCE.
UNBALANCED AFTER CHARMANDER, NEED SINGLE LEFT ROTATION AT ROOT TO REBALANCE.
TREE IS BALANCED.
```