Computer Science Competition
Region 2016
Programming Problem Set

I. General Notes

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

2. All problems have a value of 60 points.

3. 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.

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

5. 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	Magda
Problem 2	Nicole
Problem 3	Oleg
Problem 4	Paulina
Problem 5	Quincy
Problem 6	Raj
Problem 7	Sakshi
Problem 8	Thiago
Problem 9	Veronika
Problem 10	Wayne
Problem 11	Yaroslav
Problem 12	Zoe

1. Magda

Program Name: Magda.java **Input File: magda.dat**

Magda knows how to convert between bases with integer values, but has wondered about converting floating point values in different bases. She found a tutorial that shows her how to convert from base ten floating point values to binary, and needs your help writing a program to do this process. The tutorial says to repeatedly multiply a base ten floating point value by 2, taking the whole number value each time, leaving behind the fractional value, until the final answer is 1.0.

For example, to convert .140625 to its floating point equivalent in base 2, which is .001001 base 2, you do this:

.140625 * 2 equals 0.28125, which yields 0
.28124 * 2 equals 0.5625, which yields 0
.5625 * 2 equals 1.125, which yields 1
.125 * 2 equals 0.25, which yields 0
.25 * 2 equals 0.5, which yields 0
.5 * 2 equals 1.0, which yields 1

resulting in .001001 as the base two floating point value.

Input – Several base ten floating point values, each on a separate line.

Output - The first ten digits (or less, if the process terminates before that) of the equivalent base two floating point value, each one a separate line. Do not print any digits to the left of the decimal point.

Sample data:
```
.140625
.111
```

Sample Output:
```
.001001
.0001110001
```

2. Nicole

Program Name: Nicole.java Input File: nicole.dat

Nicole has been studying the Fibonacci sequence and has decided to come up with her own sequence, the "Nicole" series, but different each time based on new values. She decides to see what it would look like if she started with any three numbers and create a fourth one as the sum of the first three, minus 3 each time.

For example, if the first three values are 1, 0 and 4, the fourth value would be 1+0+4-3, which equals 2. The next value would be the sum of 0, 4 and 2, less 3, which would be 3. The next one would be 4+2+3-3 which is 6, and so on and on. The first 9 values in the series would be: 1, 0, 4, 2, 3, 6, 8, 14, 25, ...

Input - Several data sets of four integers A, B, C, and N, all on one line with single space separation. Each of the set of integers A, B, and C which start a "Nicole" series can be any value within the range -100...100. The fourth integer N (2<=N<=20) indicates the value of that series to output.

Output - The Nth value of the "Nicole" series for the three given values.

Sample data:

```
1 0 4 9
1 1 2 5
```

Sample Output:

```
25
1
```

3. Oleg

Program Name: Oleg.java Input File: oleg.dat

Oleg loves board games, especially chess and tic-tac-toe. After seeing an old Star Trek episode showing Spock and his 3D chess set, he thought about writing an AI program to play 3D chess, but decided to start with something easier, like 3D tic tac toe.

However, he needs your help to write this program. He has decided to input a string of characters to represent the three boards, like this one, "**O-O--X-XOO-X-X-OO-X-X-X---O**". Each level has 3 rows and 3 columns, as you can see in the diagram below: the first board with positions numbered 1 through 9, the second 10 through 18, and the third 19 through 27, with the entire diagram representing the given string.

1 O	2 -	3 O
4 -	5 -	6 X
7 -	8 X	9 O

10 O	11 -	12 X
13 -	14 X	15 -
16 O	17 O	18 -

19 X	20 -	21 X
22 -	23 X	24 -
25 -	26 -	27 O

To start with, he just wants to know, given a game already underway, if an additional move by X will win the game.

For example, the display shows the current moves for X and O, with the string "O-O--X-XOO-X-X-OO-X-X-X---O" representing those moves. If X plays on position 5 (row 2, col 2 of the top board), will X win the game? The answer is yes, since positions 14 and 23 immediately below position 5 already have Xs on them, winning with three Xs in a row. In fact, any "three in a row" answer will win, whether it vertical, horizontal, or diagonal on the original board, or some series that is in a straight line on all three boards. For example, if an X had been played in position 20, the winning series would be 8, 14, and 20.

Input - Several lines of data, each with two items: a 27 character string containing X, O and - indicating played and open positions on the three boards, followed by an integer representing the next move by X.

Output - All of the possible wins for X with this last move, showing all three position values (in ascending sorted order) that cause the win, or the words "NO WIN" if there is no win with this move. If the move results in several possible "wins", list each combination in order by 1st number, then if necessary by second number. A blank line will follow the output result for each data set.

Sample data:

```
O-O--X-XOO-X-X-OO-X-X-X---O 5
XO--X-O--O---X-XXO--X---O-O 11
X-X-X-O-OO-O-O-X-XX-X---O-O 23
```

Sample Output:

```
5 14 23

1 11 21
11 14 17

NO WIN
```

4. Paulina

Program Name: Paulina.java **Input File: paulina.dat**

In the last weekend campout with her Scouting Venture Crew, Paulina learned about orienteering, but still needs a little bit of help. In the training, instead of the traditional value of zero being north, zero was considered to be east. The direction 360 was also east, as was -360. Due west was the positive or negative 180. Due north was either 90 or -270. To make it easier, the orienteering course only dealt with directions that were multiples of 30 or 45, which made it easy for her to use the special 45-45-90 and 30-60-90 triangle formulas she had just learned in algebra.

The purpose of the course was to predict an ending position on the coordinate grid, with (0,0) being the home base location, and all other positions relative to home base. The instructions given were in numeric pairs, each representing a vector, the first value indicating the distance traveled in miles, and the second the direction traveled. The task was to predict the ending location of the course.

Several vectors could be given, representing multiple movements in one trek. For example, the vector (5, 45) meant to go 5 miles in a north-east direction, which would end up at position (3.5355, 3.5355) relative to home. An additional vector of (4, -30) in the same course would start from (3.5355, 3.5355), go 4 miles in the direction 30 degrees below east, ending up at (6.9996, 1.5355).

Input - Several sets of data, each representing one trek, and each on one line with single space separation. Each data set consists of an initial value N, followed by N vector integer pairs, each pair consisting of a distance in miles, and a positive or negative direction (-360 <= direction <= 360) relative to East, as described above. Each direction is guaranteed to be a positive or negative multiple of 30 or of 45.

Output - An ordered pair representing the final **(x, y)** position upon completion of the trek as designated by the data set, enclosed within parentheses, with a single space after the comma, each coordinate value output rounded to a precision of 4 decimal places. A tolerance of +-0.0001 will be accepted for either value.

Sample data:

```
2 5 45 4 -30
4 2 90 2 180 2 270 2 0
3 2 120 2 -330 2 60
```

Sample Output:

```
(6.9996, 1.5355)
(0.0000, 0.0000)
(1.7321, 4.4641)
```

5. Quincy

Program Name: Quincy.java Input File: quincy.dat

Quincy loves floating point numbers, especially the classic irrational values like PI and E, because they go on forever! He has found some limited precision representations of these numbers, as well as several others with precision values of over 1000 places, and has decided to do some analysis research on them. He wants to know how many instances of each digit there are in these values, and then will represent those counts in a histogram. He also wants to know how the frequency changes when the values are halved or doubled. He starts with some easy values, just to test his process for accuracy, and then tries it on the longer ones.

For example, with the value 123456.7089, it is easy to see one instance of each digit. If he halves this value to get 61728.35445, the digit frequencies change a bit, and if he doubles the value, they change again. He decides to output all three versions of the value - the original, half the value, and twice the value - and then calculates and displays the digit frequency for each one. Below are several results of his program. By the number of stars, you can see one zero digit in the original number, but none in the half value or double value. The number of 2s, 3s, 6s, 7s, and 8s remain the same, and the frequencies for the rest of the digits change a bit.

```
123456.7089
61728.35445
246913.4178
0 *||
1 *|*|**
2 *|*|*
3 *|*|*
4 *|**|**
5 *|**|
6 *|*|*
7 *|*|*
8 *|*|*
9 *||*
```

Then he found a limited precision version of PI, containing 50 digits, and processed that value the same way, with the following results.

```
3.14159265358979311599796346854418516159057617187 5
1.57079632679489655799898173427209258079528808593 75
6.28318530717958623199592693708837032318115234375 0
0 *|****|****
1 *********|**|******
2 *|*****|*****
3 ****|***|*********
4 ****|**|*
5 *********|*******|*****
6 *****|***|***
7 *****|********|*****
8 ****|*******|******
9 *******|*********|*****
```

After searching some more, he found a version of PI with over 200 digits, and decided to modify the histogram output, showing up to 25 stars for each count, or just a value followed by "*s" for any counts beyond 25, so that his output would stay on the screen and not wraparound. He also limited the value output to no more than 60 digits for the same reason. Here is the outcome for that more precise value of PI.

```
3.14159265358979323846264338327950288419716939937510582097 49
1.57079632679489661923132169163975144209858469968755291048 74
6.28318530717958647692528676655900576839433879875021164194 98
0 ********************|********************|*****************
1 **********************|26*s|************************
2 ************************|************************|************************
3 ********************|******************|****************
4 ************************|27*s|******************
5 **********************|**********************|********************
6 ******************|********************|29*s
7 *************|*********************|******************
8 27*s|*************|***********************
9 ************************|************************|************************
```

Write a program that will show Quincy's process as described above.

Input - several floating point values, each on one line, each containing at least 3 digits (including the .) and no more than 1000 digits.

Output - Three versions of each value - the original value, half the value, and twice the value - showing no more than 60 digits of each, followed by a histogram showing the digit frequency counts as described and shown above.

Sample input:

```
123456.7089
3.14159265358979311599796346854418516159057617187 5
3.14159265358979323846264338327950288419716939937510582097494459 (...over 200 total digits)
```

Sample output:

See output examples above. One blank line will follow each output set.

6. Raj

Program Name: Raj.java Input File: raj.dat

Raj is trying to build a better calculator. Instead of a typical calculator which reads in-order expressions and evaluates them, Raj wants to build a calculator that takes in numbers and *assigns* the operations to yield the highest value expression. The current operations on Raj's calculator are pretty simple; he can only assign pairs of parentheses, and the operators for addition, subtraction, multiplication, and division. His calculator **can rearrange the numbers** if it helps him find the optimal value.

For example, the list **`4.0 5.0 3.0 2.0 1.0`** using Raj's new calculator should find the optimal expression yielding the maximum value to be:

`4.0 * 5.0 * 3.0 * (2.0 + 1.0) = 180`

Please help Raj design his calculator!

Input: The first integer N will represent the N data sets to follow. Each line will be a list of **n** floating point values.

Output: Each data set should return a single floating point value rounded to a precision of two decimal places representing the highest value expression that Raj could create by inserting the symbols **`( ) + - * /`** into the given expression.

Assumptions: Each decimal will fit into a Java double, and there will be at least one number in each of Raj's lists. **`Infinity`** is a valid largest value and output.

Sample Input:

```
3
4.0 5.0 3.0 2.0 1.0
1.0 1.0 1.0 1.0 -1.0
-1000.33 0.33 0.45 10.1
```

Sample Output:

```
180.00
5.00
30629.93
```

7. Sakshi

Program Name: Sakshi.java **Input File: sakshi.dat**

After completing a recent math class study unit on powers, Sakshi is experimenting with various bases and exponents to make powers, and needs your help writing a program for her research. For example, when she uses the base value 10 and exponent 2, she gets the value 100, and for 100 and 0.5, the result is 10, since 0.5 is the exponent that gives the square root of a value. But she is also interested in other values for bases and exponents, and is curious what powers they produce.

Input - Several pairs of non-negative real numbers, each pair on one line, representing the base and exponent of a power P. All data values will be no greater than 10,000.

Output - The resulting value P, shown rounded to a precision of three decimal places. A tolerance of +- 0.001 will be accepted. All output values are guaranteed to be within range of standard data types.

Sample data:

```
10 2
100 .5
4.5 3
1.1 5
```

Sample Output:

```
100.000
10.000
91.125
1.611
```

8. Thiago

Program Name: Thiago.java **Input File: thiago.dat**

Thiago has lots of fun playing with encryption processes, and is experimenting with encoding his friends' names, first counting how many names are on the list, then adding up the ASCII values of the capital letters in each person's name, and finally multiplying each person's letter sum by the number of people on the list to find each person's unique number code for that group of friends. Help him with this interesting encryption process by writing a program to help him out.

For example, in the data sample below there are three names, the first of which is Magda. The values for the letters M, A, G, and D are 77, 65, 71 and 68, for a total sum of 346 for all five letters. The product of this value and 3 (number of friends in the list) is 1038, the value Thiago assigns as the code number for Magda among this group of friends.

Input - A list of names, each on one line, with no symbols or spaces contained in any of them.

Output - Each capitalized name, followed by a single space, and then followed by the calculated integer code for that person, as described above.

Sample data:

```
Magda
Nicole
Oleg
```

Sample Output:

```
MAGDA 1038
NICOLE 1326
OLEG 885
```

9. Veronika

Program Name: Veronika.java Input File: veronika.dat

Creating letter patterns with a program is one of Veronika's favorite things to do. Lately she has been playing around with a layered pattern, where she takes any word and makes a "concentric square" with the letters, where the first letter of the word is the outside layer of the square, the second is the next layer, and so on until the last letter is in the middle of the square.

For example, with the word **UIL**, the square would look like this:

```
UUUUU
UIIIU
UILIU
UIIIU
UUUUU
```

or this example using the word **REGION**:

```
RRRRRRRRRR
REEEEEEEER
REGGGGGGER
REGIIIIIGER
REGIOOOIGER
REGIONOIGER
REGIOOOIGER
REGIIIIIGER
REGGGGGGER
REEEEEEEER
RRRRRRRRRR
```

Input - Several words, all in caps, each on one line, and each with a length of at least 2 and no more than 25.

Output - A concentric square for each word, as shown above, each output followed by a blank line.

Sample data:

```
UIL
REGION
```

Sample Output:
See above

10. Wayne

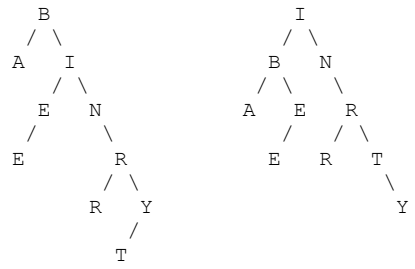
Program Name: Wayne.java **Input File: wayne.dat**

After completing a study unit on binary search trees, Wayne was considering how to keep these search trees better balanced. He had learned that the IPL (internal path length) of a tree is a good indicator of how balanced it is. He recalled that the **internal path length** of a tree is the sum of the depths of each node in the tree, and the better balanced a tree is, the lower the IPL value is. His idea was to take a data set, build a normal binary search tree by starting with the first element as the root node, and inserting the others in order, allowing duplicate values and **sliding ties to the left**. He would calculate the IPL of the normal tree, and then use a "two-ahead" technique in an attempt to build a more balanced BST (binary search tree).

This "two-ahead" process works like this. Consider the first three elements of the data set, arrange them in order, and insert the middle element of those three as the root. After that, add another element from the data set to the two remaining elements, arrange them in order again, and insert the middle element of those three. Continue this process until the last data element is added to the group, and again, insert the middle of those last three. When only two elements remain, insert the lesser of those two, and then insert the last one standing.

Supposedly, Wayne thought, this would make for a more balanced tree, which in turn would be more efficient. He decided to research this idea with several sets of data, consisting of capitalized words with no symbols or spaces, calculating the IPL of the standard BST for each word, and then the IPL of the "two-ahead" BST, and then determining if there was any improvement, and by how much. The IPL of the "improved" tree should be a lower value, but not always, as his research discovered. Here is how the two trees would look, using the characters in the word: **BINARYTREE**

Standard BST "Two-ahead" BST



The internal path length of the standard tree is 25 (two nodes 'A' and 'I' at depth 1, two at depth 2, two at 3, two at 4, and 1 at 5). The root node has a depth of zero, and does not affect the IPL. The "two-ahead" tree has an IPL of 21 (the two nodes 'B' and 'N' at depth 1, three at 2, three at 3, and one at 4). The difference shows a better IPL value by 4, which indicates a more balanced, and in turn, more efficient binary search tree. His report for the word "BINARYTREE" would be "`25 21 4 BETTER`", indicating the IPL for the standard tree, then the IPL for the "two-ahead" tree, the positive difference, and the word word `BETTER`, since 21 shows a more balanced tree.

Write a program to simulate Wayne's research, according to the process described above.

Input - Several words, all in caps, containing no symbols or spaces, each word at least three characters in length.

Output - For each word, build a standard binary search tree, with duplicate values allowed, and ties going to the left, and then output the IPL for that tree. Do the same using the "two-ahead" BST as described above and output the IPL for that tree. Finally, output the analysis, indicating the value and word that indicate how much "BETTER" or "WORSE" the second tree is from the first. All output items for one line will have single space separation.

Sample data:

```
BINARYTREE
RESEARCH
TWOAHEAD
```

Sample Output:

```
25 21 4 BETTER
16 16 SAME
19 21 2 WORSE
```

11. Yaroslav

Program Name: Yaroslav.java Input File: yaroslav.dat

Yaroslav is currently alone in his spacecraft, engines off due to a malfunction, drifting in space. You are a programmer at NASA tasked with predicting Yaroslav's location as he drifts so that you can help others determine if he is in any danger! You know Yaroslav's current position in space (using x,y,z coordinates) and you know his trajectory (the velocity he has in the x-direction, the y-direction, and the z-direction. You also have a catalog of all the celestial bodies whose gravitational effects will affect Yaroslav's rocket. Luckily, you have a physicist friend who has given you some kinematic equations to help you determine how gravity will affect Yaroslav.

F = GMm/r^2
Force_x = GMm/r^2*(Position_x - Planet_x)/r
Force_y = GMm/r^2*(Position_y - Planet_y)/r
Force_z = GMm/r^2*(Position_z - Planet_z)/r
Acceleration_x = Force_x/m
Acceleration_y = Force_y/m
Acceleration_z = Force_z/m
Velocity_NEW_x = Acceleration_x * t + Velocity_x
Velocity_NEW_y = Acceleration_y * t + Velocity_y
Velocity_NEW_z = Acceleration_z * t + Velocity_z
Position_NEW_x = Velocity_NEW_x * t + Position_x
Position_NEW_y = Velocity_NEW_y * t + Position_y
Position_NEW_z = Velocity_NEW_z * t + Position_z
r = distance between Yaroslav and the Body.
m = 160000 kg (Mass of spacecraft)
M = mass of the Body (in kg)
G = 0.0000000006674 N m^2/kg^2 (Newton's gravitational constant)
t = timestep faction (1/1000 s)

For this problem, use a t value of 1/1000. This means that if you want the values to all of these equations at 20 seconds in the future, that you will need to update these equations 20 * 1/t = 20000 times.

For example, if Yaroslav is at position (0, 0, 0) with an initial velocity of (0, 0, 0), and there is only one celestial body: a planet with mass 59721986000000000000000000.0 kg at position (100000000.0, 100000000.0, 100000000.0), then the first calculation will be to determine the forces generated by this planet in all directions affecting Yaroslav's spacecraft. This calculation is shown below for the x variable, and would be similar for the y and z variables.

Force_x = G * (59721986000000000000000000.0 kg) (160000 kg) / ((173205000.0 m)^2) * (100000000.0 m)/(173205000.0 m) = 1227 N.

Next we use this Force to determine the acceleration:

Acceleration_x = (1227 N) / (160000 kg) = 0.007669 m/s^2

Then we use this Acceleration to determine the new Velocity:

Velocity_NEW_x = (0.007669 m/s^2) * (1/1000 s) + (0 m/s) = 0.000007669 m/s

Finally we use this new Velocity to determine Yaroslav's new Position:

Position_NEW_x = (0.000007669 m/s) * (1/1000 s) + (0 m) = (0.000000007669 m)

Now we have new values for Yaroslav's position & velocity, thus his distance to all the celestial bodies has changed, so you must do all of these calculations again. This is how much his position has changed in 1/1000 th of a second, so, you must repeat this n*1000 times to determine his position in n seconds.

Please help ensure Yaroslav's safety by writing a program to do these crucial calculations!

Input: The first integer will represent the number of data sets to follow. For each data set, the first line will be Yaroslav's initial position. The second line with be Yaroslav's initial velocity in the x, y, and z directions. The third line will be how many seconds in the future we need to know Yaroslav's position and total velocity. The fourth line will be the number of celestial bodies, p, whose gravity are going to affect Yaroslav. The next p lines will be sets of 4 doubles, the first representing the mass of the object (in kg) the next 3 representing the x, y, and z position of the object in space.

Output: Each data set should display the sentence "**`YAROSLAV IS AT POSITION (x, y, z).`**" where x, y, z represent the x, y, and z coordinates of Yaroslav.

Assumptions: Yaroslav will never fall within 500 m of a celestial body. All positions are given in meters. You'll never be asked to predict his safety more than 2 minutes in the future.

Sample Input:

```
3
0.0 0.0 0.0
2.0 2.0 2.0
60
1
597219860000000000000000.0 100000000.0 100000000.0 100000000.0
0.0 0.0 0.0
0.0 0.0 0.0
60
2
597219860000000000000000.0 100000000.0 100000000.0 100000000.0
597219860000000000000000.0 100000000.0 -100000000.0 -100000000.0
1.0 7.0 6.0
5.0 5.0 5.0
30
3
59721986000000000000.0 100000.0 200000.0 300000.0
59721986000000000000.0 3000000.0 2000000.0 1000000.0
59721986000000000000.0 200000.0 300000.0 100000.0
```

Sample Output:

```
YAROSLAV IS AT POSITION (-18.08, -18.08, -18.08).
YAROSLAV IS AT POSITION (-478.31, -0.00, -0.00).
YAROSLAV IS AT POSITION (-14164.83, -17573.83, -10698.49).
```

12. Zoe

Program Name: Zoe.java Input File: zoe.dat

Zoe has been practicing a long time for the upcoming State UIL programming contest, which she knows is on May 25, 2016 this year. She wanted to keep track of the dates that were so many days out from that important event, and needs you to write a program that will help her out with that, plus, she needs help with calendar type programs anyway.

She wants the program to calculate and output the date that is a certain number of days before the state contest date.

For example, 10 days before that day is May 15, 2016, which she wants to be in the format mm/dd/yy, which would be 05/15/16.

Input - Several integers N (0<=N<=500), each on a separate line.

Output - The date in **mm/dd/yy** format that represents the given number of days N before May 25, 2016.

Sample data:

```
10
1
86
```

Sample Output:

```
05/15/16
05/24/16
02/29/16
```