November 2018

**Computer Science Competition**

Hands-On Programming 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. Point Values and Names of Problems**

|  |  |
| --- | --- |
| **Number** | **Name** |
| Problem 1 | Bean Trapezoids |
| Problem 2 | Bean Doodle |
| Problem 3 | Bean Game |
| Problem 4 | Bean Parser |
| Problem 5 | Bean Search |
| Problem 6 | Bean Transformations |
| Problem 7 | Bean Printer |
| Problem 8 | Bean Flow |
| Problem 9 | Bean Maze |
| Problem 10 | Bean Vault |
| Problem 11 | Bean Checker |
| Problem 12 | Bean Manager |

**1. Bean Trapezoids**

# Program Name: BeanTrapezoids.java Input File: beantrapezoids.dat

You’ve gotten pretty bored, so you’ve decided to practice geometry. However, you don’t have a pen or a pencil. The only thing you have are beans. You decide to make trapezoids out of beans. Given the length of the two bases and height of a trapezoid calculate its area. The formula for the area of a trapezoid is as follows (where A is the top base, B is the bottom base, and H is the height):

**Input**

The first line will contain a single integer n that indicates the number of data sets that follow. Each dataset will consist of 3 positive floating-point values on a line. Respectively, these values represent A (the top base), B (the bottom base), and H (the height). Given these values, calculate the area of the trapezoid.

**Output**

For each data set, output the area of the trapezoid. The answer for each dataset should be on a new line.

**Example Input File**

3

4.00 4.00 1.00

3.21 6.43 9.75

1.80 10.01 1.10

**Example Output to Screen**

4.00

47.00

6.50

**2. Bean Doodle**

# Program Name: BeanDoodle.java Input File: beandoodle.dat

Unsurprisingly, you’re still bored. Math did not help at all. What else is there to do? How about doodling? Using the beans you have from making trapezoids, make some doodles. Specifically, use the beans to make bullseye/target patterns of different sizes.

**Input**

The first line will contain a single integer n that indicates the number of data sets that follow. Each data set consists of a single positive integer x on a line. This x represents the number of rings to print in each target/bullseye pattern.

**Output**

For each data set, output the target, bullseye pattern. The patterns consist of alternating rings of “#” and “ “ (spaces). After each target/bullseye pattern, print a blank line ***EXCEPT*** after the last test case.

**Example Input File**

2

1

3

**Example Output to Screen**

#

#########

# #

# ##### #

# # # #

# # # # #

# # # #

# ##### #

# #

#########

**3. Bean Game**

# Program Name: BeanGame.java Input File: beangame.dat

Yikes, clearly doodling and math aren’t the most entertaining things in the world. What else is there to do? You could play that game your old grandpa Pinto used to play with you! Bean Game! The object of the game is simple. Grandpa gives you a handful of beans. These beans are all different shapes, sizes, and weights. Next, grandpa asks you to use those beans to create a pile of beans weighing a certain weight. In this problem, you are given a finite number of beans with varying weights. It is your job to decide whether or not you can use these beans to make a pile of a given weight.

**Input**

The first line will contain a single integer n that indicates the number of data sets that follow. Each data set will start with 2 integers on a line, x and g. x represents the number of beans that grandpa gives you. g represents the weight of the pile grandpa asks you to make. On the following line, there will be x integers. Each integer represents the weight of one bean given to you by grandpa.

**Output**

For each data set, simply output “YES” or “NO” on a new line if it is possible to make a pile weighting g given the beans grandpa gives you.

**Example Input File**

2

5 10

1 2 3 4 5

3 3

4 5 6

**Example Output to Screen**

YES

NO

**4. Bean Parser**

# Program Name: BeanParser.java Input File: beanparser.dat

Bean game was probably the worst suggestion yet. You are now more bored than ever before. You decide to read the daily paper. However, you have been driven mad with all this bean nonsense that you decide to count the number of times the word bean appears in a string of words.

**Input**

The first line will contain a single integer n that indicates the number of data sets that follow. Each data set will be a single line of text. Each line will consist of a series of characters terminated by a newline.

**Output**

For each data set, output the number of times the string “bean” (***IGNORING CASE***) occurs in the string on a new line.

**Example Input File**

2

beanbangbongbambiffbowwowhowbeanpopcrashsmashbean

clapclapclapsnapsnapsnap

**Example Output to Screen**

3

0

**5. Bean Search**

# Program Name: BeanSearch.java Input File: beansearch.dat

Well, reading the newspaper and searching for the word bean wasn’t the sanest pastime, so you decide to do the word search on the back of the newspaper. However, instead of doing it manually, you decide to do some bean programming on your bean computer. Given a word search grid and a list of words, determine whether or not each word can be found in the word search. Like a traditional word search, a word must be linear across the grid. In addition, a word can be in any of the 8 possible cardinal directions (vertical, horizontal, diagonal).

**Input**

The first line will contain a single integer n that indicates the number of data sets that follow. Each data set will begin with 2 integers, r and c, on a line that represent the rows and columns of the word search grid. The next r lines will contain c characters. Each character represents one cell of the word search grid. On the following line, there will be a single integer x. The following x lines will contain one word each. For each word, determine whether or not it can be found in the word search grid.

**Output**

For each word in each data set, output “FOUND” or “NOT FOUND” depending on whether or not it can be found in the word search grid. Output each answer on a new line.

**Example Input File**

1

5 5

ABCDE

FGHIJ

KLMNO

PQRST

UVWXY

4

ABCDE

AGMSY

APPLE

KITES

**Example Output to Screen**

FOUND

FOUND

NOT FOUND

NOT FOUND

**6. Bean Transformations**

# Program Name: BeanTransformations.java Input File: beantransformations.dat

Enough word searches. Let’s just do some weird bean math. Using the beans you have, you form them into a number. Next, you change the beans to represent that number in binary. You decide to count the number of ones in every nyble (a nyble is 4 bits). Using the number of ones in each nyble you form a new number. The new number is simply the number of ones in each nyble lined up. For example, in the binary string 10100001, the corresponding nyble count is 21 (1 one in the 4 least significant bits 0001 and 2 ones in the 4 2nd 4 least significant bits 1010). To be even more fun, you decide to take the nyble count and interpret it as a base 16 number. Finally, you must covert the nyble count from base 16 to base 10. In the previous example, 2116 = 3310.

**Input**

The first line will contain a single integer n that indicates the number of data sets that follow. Each data set will consist of a single integer x. X is the integer that you need to transform.

**Output**

For each data set, output the number after it undergoes transformation on a new line.

**Example Input File**

4

15

1

1024

11034

**Example Output to Screen**

4

1

256

4882

**7. Bean Printer**

# Program Name: BeanPrinter.java Input File: NONE

Using the beans you used when transforming numbers, you decide to use them to spell out the word bean. Very simply, print out the word bean as shown below.

**Input**

There is no input for this problem.

**Output**

Output exactly the same text as shown in the Example Output to Screen.

**Example Output to Screen**

BBBBB-EEEEE-------A-------N-----N

B---B-E----------A-A--- --NN----N

B---B-E---------A---A-----N-N---N

BBBB--EEEEE----AAAAAAA----N--N--N

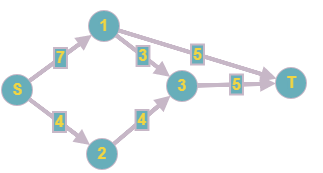
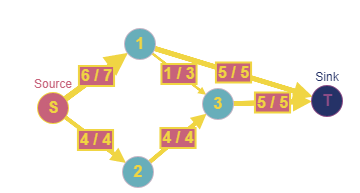
B---B-E-------A-------A---N---N-N

B---B-E------A---------A--N----NN

BBBBB-EEEEE-A-----------A-N-----N

**8. Bean Flow**

# Program Name: BeanFlow.java Input File: beanflow.dat

Because bean art isn’t the most challenging problem, you are in search of a harder problem. In addition to being the creator of the bean game, your grandpa was also a plumber. Around the house there is a plethora of various pipes. You begin connecting them to create one extensive network of pipes. Once you are done, you wonder how many beans you can get from the start of the pipe network to the end of the pipe network. Each pipe has a “bean flow rate.” The bean-flow-rate of a pipe is simply the number of beans that can flow down it in a unit of time. Your job is to determine the total max bean-flow-rate of the pipe network from the network’s source (start) to the network’s terminal (end). You will be given the pipe network in the form of an adjacency matrix (example below). Each row contains the connections for the rth node to the cth node. In the example, the location (row, column) [0, 1] represents that there is a directed (one-way) pipe connecting the source node to node 1 with a flow rate of 7.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | S | 1 | 2 | 3 | T |
| Source | 0 | 7 | 4 | 0 | 0 |
| 1 | 0 | 0 | 0 | 3 | 5 |
| 2 | 0 | 0 | 0 | 4 | 0 |
| 3 | 0 | 0 | 0 | 0 | 5 |
| Terminal | 0 | 0 | 0 | 0 | 0 |

**Input**

The first line will contain a single integer n that indicates the number of data sets that follow. Each data set will begin with a single integer, x, on a line. The next x lines will contain x space-separated integers representing the adjacency matrix of the pipe network. The very first row represents the source node, the very last row represents the terminal node, and the rows in the middle represent the rest of the nodes.

**Output**

For each data set, simply output the maximum flowrate of the pipe network on a new line.

**Example Input File**

1

5

0 7 4 0 0

0 0 0 3 5

0 0 0 4 0

0 0 0 0 5

0 0 0 0 0

**Example Output to Screen**

10

**9. Bean Maze**

# Program Name: BeanMaze.java Input File: beanmaze.dat

You’ve had enough games. You want to get out of the house and get some more beans. Why do you need more beans? No one knows. The one thing we do know is that there’s a bean bank about 15 minutes away from your house. Inside the bank is a crazy maze. However, at the end of the maze there is a giant vault full of beans. Your job is to write a program to solve the maze in the bank’s interior. Inside the maze you can move up, down, left, and right. However, you can not move through walls.

**Input**

The first line will contain a single integer n that indicates the number of data sets that follow. Each data set will start with 2 integers on a line, r and c. R and C represent the number of rows and columns within the maze. The next R lines will contain C characters which represent the maze. “S” represents the start of the maze. “B” represents the end of the maze (the bean vault). “#” represents a wall (you cannot move through or onto walls). “.” represents an open space that you may freely pass through.

**Output**

For each data set, simply output “POSSIBLE” or “NOT POSSIBLE” on a new line based on whether or not it is possible to reach the end of the maze from the start.

**Example Input File**

2

2 3

S..

..B

5 5

S#...

.#...

..#..

.#...

.#..B

**Example Output to Screen**

POSSIBLE

NOT POSSIBLE

**10. Bean Vault**

# Program Name: BeanVault.java Input File: beanvault.dat

You have reached the bean vault. Inside there are various types of beans that are different weights and values. Sadly, you can only carry so much weight inside your burlap sack. You must decide which beans you want to take home in order to maximize the value of beans that you steal from the bank. Your job is to write a program that will determine the maximum value you can steal from the bank given a capacity for the burlap sack.

**Input**

The first line will contain a single integer n that indicates the number of data sets that follow. Each dataset will start with a single integer on a line, c. C represents the capacity of the burlap sack you have to carry items. The next line will contain a variable amount of numbers (the amount is not given to you, but all of the numbers will be on one line). These numbers represent the weights of each of the items. The following line will contain the same number of integers as the previous line. These numbers represent the values of each of the items on the previous line. For example, the first item in the example input has a weight of 1 and a value of 2. All items are finite. All bag capacities will be positive.

**Output**

For each data set, output the maximum value you can steal from the vault given a bag capacity c. The answer to each data set should be on a new line.

**Example Input File**

2

5

1 2 3 4 5

2 3 4 5 1

10

1 1 1 1 1 2 3 4 5 6

35 6 7 8 9 10 15 16 17 18 19 20

**Example Output to Screen**

7

90

**11. Bean Checker**

# Program Name: BeanChecker.java Input File: beanchecker.dat

You’ve stolen a ton of beans from the bean vault. Now you need to check whether these beans are truly valuable. A bean is valuable if both the bean’s name contains an “e” (***INGNORING CASE***) and the bean’s weight is odd.

**Input**

The first line will contain a single integer n that indicates the number of data sets that follow. Each data set will consist of a string and integer, separated by a space, on a line.

**Output**

For each data set, output either “YES” or “NO” depending on whether or not a bean’s name contains an “e” (***INGNORING CASE***) and the bean’s weight is odd.

**Example Input File**

2

Bean 1

Ban 1

**Example Output to Screen**

YES

NO

**12. Bean Manager**

# Program Name: BeanManager.java Input File: beanmanager.dat

Now that you’ve checked whether beans are valuable or not, you need to sort them. Given a list of beans with names, colors, prices, and weights, sort them in alphabetical and increasing order. The sorting is based on name, then color, then weight, and then finally price.

**Input**

The first line will contain a single integer n that indicates the number of data sets that follow. Each data set will start with a single integer x on a line. X denotes the number of beans in each data set. The next x lines will contain the name, color, price, and weight of the bean. The name and color of the bean will be strings containing only letters. The price of the bean will be a “$” followed by a floating-point value. The weight will be an integer followed by a “g”

**Output**

For each data set, output the beans, sorted based by name, color, weight, and price. Each bean should be printed on a new line. The bean’s information should be printed in the same format as the input, but the weight should be listed before the price. Each piece of information should be separated by a space. The weight of the bean should be an integer followed by a “g”. The price should be a “$” followed by the bean’s price rounded to two decimal places. After each data set, print a new line ***EXCEPT*** for the last data set.

**Example Input File**

2

3

Pinto Brown $0.12 1g

Black Black $0.10 3g

Lima Green $0.03 4g

2

Pinto Brown $0.2 1g

Pinto Brown $0.1 1g

**Example Output to Screen**

Black Black 3g $0.10

Lima Green 4g $0.03

Pinto Brown 1g $0.12

Pinto Brown 1g $0.10

Pinto Brown 1g $0.20