**Java program:** Prob12.java

**Input File:** Prob12.in.txt

**Output:** Your output needs to be directed to stdout (i.e., using System.out.println())

**Introduction**

When computer data disruption and corruption cannot be tolerated, such as in scientific or financial computing applications, Error-Correcting Code (ECC) memory can be used to limit the impact of electrical and magnetic interference inside of a computer. One of the most popular ECC techniques is implemented through a hamming code. A hamming code includes a number of extra parity bits in precise locations that allow for the detection and correction of errors. The number of parity bits that you need to add depends on the length of the data you are trying to encode. The general algorithm for creating a hamming code is as follows:

1. Number the bits from left to right starting with the number 1.
2. Label the bit positions with their numbers converted to binary:
   1. Bits whose position numbers are powers of 2 (1, 2, 4, etc.) are the parity bits. Notice that these are the positions where there is a single 1 in the binary representation of the bit position number. This is the key to calculating the value of the parity bits.
   2. All other bits are the data bits.
3. Fill in the data bits with the original data.
4. Calculate the value of the parity bits. For this problem, we will be using even parity.
   1. Parity is calculated by adding the value of each checked bit. If the value is even, the parity is 0; if the value is odd, the parity is 1.
   2. Parity bit 1 checks all bit positions which have the least significant bit set to 1 (1, 3, 5…)
   3. Parity bit 2 checks all bit positions which have the second least significant bit set to 1 (2, 3, 6…)
   4. Parity bit 4 checks all bit positions which have the third least significant bit set to 1 (4-7, 12-15…)

**Example hamming code:**

Suppose we had an original input of 1101, and we wanted to create the hamming code for this set of bits. We would do the following:

1. Calculate the number of parity bits we need. Remember that the parity bits are in the bit positions that are a power of two. You can notice from the table below that adding a parity bit at position M gives us an additional M-1 data bits that we can encode. For this example, we have 4 data bits. Parity bit 1 does not give us any data bits. Parity bit 2 gives us one data bit, and parity bit 4 gives us three more. Thus, we need 7 bits total: 3 for parity, and 4 for data.
2. Calculate the values of the parity bits using the algorithm above. The table below shows the bits considered when calculating each parity bit.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Bit Position (int): | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| Bit Position (binary): | 0001 | 0010 | 0011 | 0100 | 0101 | 0110 | 0111 |
| Bit Label: | P1 | P2 | D1 | P4 | D2 | D3 | D4 |
| Original Data: |  |  | 1 |  | 1 | 0 | 1 |
| P1: | 1 |  | 1 |  | 1 | 0 | 1 |
| P2: |  | 0 | 1 |  | 1 | 0 | 1 |
| P4: |  |  | 1 | 0 | 1 | 0 | 1 |
| Encoded Data: | 1 | 0 | 1 | 0 | 1 | 0 | 1 |

Table 1: Hamming Code Example

For each parity bit calculation, the parity bit’s position is highlighted black, and the bits used to calculate it are highlighted grey. Looking at parity bit 2, its position number in binary is 0010, so it only cares about positions that fit the pattern xx1x, where the x could be either a 0 or a 1. Positions 3, 6, and 7 fit this pattern. Two of the three of those data bits are 1, so the parity bit’s value is 0 because there are an even number of 1 values in the data for that parity bit.

**Program Input**

The first line of the file Prob12.in.txt will contain a positive integer T denoting the number of test cases that follow. Each test case will have the following input:

* The first line of each test case will contain a positive integer N denoting the number of binary numbers that follow.
* The next N lines will contain binary numbers, one per line. You will not know the length of each number beforehand.

**Example Input:**

2

1

1101

3

01001101

11011101

10011010

**Program Output**

For each binary number read, your program should output the hamming code for that number.

**Example Output:**

1010101

010010011101

011110111101

011100101010

**Java program:** Prob13.java

**Input File:** Prob13.in.txt

**Output:** Your output needs to be directed to stdout (i.e., using System.out.println())

**Introduction**

Sky scrapers are the tallest man-made structures in the world; the tallest being BurjKhalifa in Dubai. They are expertly designed to stretch high into the sky and resist wind and seismic activity. Your task, should you choose to accept it, is to build the tallest sky scraper possible from a set of bricks. But like any good construction project there are design requirements:

* You will be given a set of rectangular bricks of varying sizes. You can use each brick only once.
* You must stack them on top of one another to build the tallest structure possible.
* You can only stack a brick on top of another brick if the dimensions of the lower brick’s top are greater than or equal to the base of the brick you are stacking on top.
* You can rotate the brick so that any of the sides can function as its base.
* Brick surfaces must stay parallel to the x, y, and z axes, so only 90 degree rotation is possible in any direction.

**NP-Complete**

This problem is an example of an NP-complete type problem. What that means is that there is no slick algorithm that will cut your program’s running time down. It also means that we promise to keep the judging input set small. In our testing, even 50 blocks made us go over the 2 minute run time allotted for problem submissions. The judging data will have a maximum of 25 blocks, and our solution will run for a maximum of 30 seconds. The judging software will allow your program two minutes to run before timing out. Good luck!



**Program Input**

The first line of the file Prob13.in.txt will contain a positive integer T denoting the number of test cases that follow. Each test case will have the following input:

* The first line of each test case will contain a positive integer N denoting the number of blocks.
* The next N lines will contain the description of a single block in the form LxWxH, where L is the block’s length, W is the block’s width, and H is the block’s height.

**Example Input:**

2

4

1x2x3

2x2x4

5x4x1

2x1x2

5

7x7x7

6x6x6

40x40x1

5x5x5

4x4x4

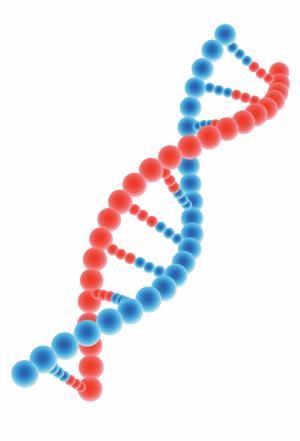
**Program Output**

Your program should print out the height of the tallest tower you can make with the blocks you were given.

**Example Output:**

12

40

**Java program:** Prob14.java

**Input File:** Prob14.in.txt

**Output:** Your output needs to be directed to stdout (i.e., using System.out.println())

**Introduction**

Last month Bobby reported to the police that someone broke into his house and stole his favorite pair of jeans. He was extremely distraught and asked the police to do whatever it takes to find the burglar. The police looked for DNA samples from around the house. Luckily they found only one, and they believe it belongs to the burglar. Unfortunately, the police’s database searching program is on the fritz so they want you to create a program to search their database of all the people in the town and find the closest DNA match.

DNA samples are rarely perfect – parts of a sample could be missing or contaminated. Therefore, a direct match is not usually found. The police use the “longest common subsequence” method to find the most likely suspect. Two strings share a common subsequence if they have the same set of letters in the same order, but the letters in the subsequence do not necessarily have to be adjacent to one another.

For example:

* The string ABCDEFGHIJKLMNOPQRSTUVWXYZ has subsequences KNOT and MOW, but not PAT.
* The longest common subsequence between the strings ABAFCDEF and BCFEDE is BCDE.

**Program Input**

The first line of the file Prob14.in.txt will contain a positive integer T denoting the number of test cases that follow. Each test case will have the following input:

* The first line of each test case will contain a string representing the DNA sequence found by the police.
* The second line of each test case will contain a positive integer N denoting the number of items in the town’s database.
* The next N lines of each test case will contain one item in the town’s DNS database. Each item will be in the form NAME=DNA\_STRING. The DNA sequence letters is made up of the letters A, C, G, and T (A: Ademine, C: Cytosine, G: Guanine, T: Thymine).

**Example Input:**

1

TTTCAGTCTTCGAAACGT

5

David=ATGGCCATCGGGGTCGGCCGTCGCTGGC

Bobby=TTTCAGTCTTCGACGT

Brian=TTGAATGGCGTCTGGCAAACTGGCTT

Jose=TTGACCATGACGTGCCCACTGGC

Kyle=TTGACCAGGGGAATAAACTTTCT

**Program Output**

Your program’s output should display the name of the person whose DNA was the closest match to the sample according to the length of the longest subsequence algorithm. If the length of the longest subsequence is shared by two or more people, output the names in alphabetical order separated by commas.

**Example Output:**

Bobby

**Java program:** Prob15.java

**Input File:** Prob15.in.txt

**Output:** Your output needs to be directed to stdout (i.e., using System.out.println())

**Introduction**

Reverse Polish Notation (RPN, also called postfix notation) is a mathematical notation where the operator (+, -, \*, /, etc.) follows the operands. For example, 2 + 3 in postfix notation is 2 3 +. This notation has the advantage that it’s easy to evaluate in a computer program and it doesn’t require parenthesis to handle order of operations.

Your task is to write a program that will convert expressions from algebraic (also called infix) notation to RPN.

**Some Notes on RPN**

* Operators use the two operands to their left if you’re looking at the final RPN equation. Once used, the result becomes available as an operand for another operation.
  + A + B – C \* D --> A B + C D \* -

In this example, A and B are added together and saved for later. C and D are multiplied together and saved for later. The minus sign at the end acts on both saved values and subtracts the two.

* Operators are used from left to right. Note the difference between these two lines because we know multiplication has to happen before addition. The order of the operators is switched in the second example to account for the order in which the operations must happen.
  + A \* B + C --> A B \* C +
  + A + B \* C --> A B C \* +
* Exponentiation is right-associative (meaning if two exponent operations are next to each other, they happen from right to left instead of left to right):
  + 2 ^ 2 ^ 3 is the same as 2 ^ (2 ^ 3), not (2 ^ 2) ^ 3
* The operands should stay in the same order in your output. For example, if you were given the expression A + B, the answers A B + and B A + are mathematically equivalent because of the commutative property of addition. However, for the purposes of this problem, the only acceptable answer would be A B +.

**Program Input**

The first line of the file Prob15.in.txt will contain a positive integer T denoting the number of test cases that follow. Each test case will have the following input:

* The first line of each test case will contain a positive integer N denoting the number of input lines that follow.
* The next N lines will each contain an expression in regular algebraic notation for you to convert. Each character in the expression will be separated by a space.
* The only non-operand characters you will encounter are PEMDAS characters:
  + ( and ) for grouping
  + ^ for exponentiation
  + \* and / for multiplication and division
  + + and – for addition and subtraction
* If you are unfamiliar with PEMDAS, it is a pneumonic device for remembering the order in which operations are performed:
  + Parentheses first (innermost to outermost)
  + Exponentiation second
  + Multiplication and Division next (they have the same precedence)
  + Addition and Subtraction last (they have the same precedence)

**Example Input:**

2

2

a \* b + c

2 ^ 2 ^ 3

3

( a - b ) \* c / d + e / f

( ( a + b ) - ( c - d ) ^ x ) + q \* w

A + B \* C - D

**Program Output**

Your program should convert the expressions in the order they are read to RPN. All your output characters should be separated by a single space.

**Example Output:**

a b \* c +

2 2 3 ^ ^

a b – c \* d / e f / +

a b + c d - x ^ - q w \* +

A B C \* + D -