# 2014 Cornell University
# High School Programming Contest

# RULES

**Team formation:**
To compete you should be a team of three students enrolled in the same institution. You may bring with you as much printed material as you like (books, old codes you have developed, course notes, etc.), but no electronic device will be allowed. That means that any laptop, phone, smartwatch, iphone, USB drive or any other form of media will not be allowed on the competition floor.

**The problem set:**
The problem set will be formed of 6 to 12 problems, varying in difficulty. **The problems are NOT presented in order of difficulty.** You may solve your problems in any order you want, and each problem awards you with the same amount of points. More about grading later.

**Submitting:**
After reading a problem statement, if you decide to approach it, you should code your solution in one of the accepted languages (C, C++, Java or Python). After you test your solution in your computer, and think that it is correct, you may submit your code for judging. Upon receiving your code the automatic system will:

1. Compile your code (if not in Python)
2. Execute your code with a prepared set of hard input (that are hidden from the contestants)
3. Compare the output your program generated with the expected output.

Note that this will be done by an automatic judge, no human will read your code. After the judging (usually a few seconds, but may be more depending on the judging queue) you will receive a verdict which can be:

- **NO - Compilation Error** - Your code didn't compile
- **NO - Time Limit Exceeded** - Your code took longer to run than the allowed time limit
- **NO - Wrong Answer** - Your code ran in time, but it output a wrong solution for at least one of the test cases.
- **NO - Runtime Error** - Your code ran into a runtime error, for instance, a division by 0, or a syntax error in the case of Python.
- **YES** - Your code compiled and ran correctly, and output a correct answer.
- **NO - Other (Please see staff)** - unpredictable things can happen and we have to be prepared.

We will be using the system PC2 for controlling the contest. You are encouraged to read their manual at http://www.ecs.csus.edu/pc2/doc/v9/PC2V9TeamGuide.pdf

**Scoring:**

The first criterion for scoring is the total number of problems with a **YES** verdict. If team A solved 8 problems and team B solved 6 problems, team A will be ahead of team B in the ranking, regardless of which problems they solved, and of the time it took them.

The tie breaking is done by considering the "penalty". Penalty has two components:

1. You will receive a penalty of 20 for each "NO" submission **of a problem you eventually solve.** If you don't eventually solve the problem, you won't be penalized for wrong submissions.
2. If you receive a "YES" on a problem **X** minutes after the competition started, you will receive a penalty of **X**.

In case of ties in number of problems, teams will be ranked by smallest penalties. In the unlikely case of ties in both number or problems and penalties, the judges may decide to make a (subjective) determination based on the quality of the submitted code.

**Clarifications:**

In case of ambiguity in a problem statement, you may ask for a clarification through the PC2 system. Clarifications go to the judge anonymously and are completely free. As a rule of thumb, in case of doubt, ask for a clarification. In the worst case, you will receive back something that is not helpful.

**At your station you will have at your disposal:**

- One computer (note that there will be only one computer per team of three people)
- Three copies of the problem set, printed. These copies are yours to keep, and you can write on them as you will.
- One calculator
- Sheets of paper and pens.

Apart from that, you will be able to print anything you want (for instance, to analyze code while another team member is working on another problem on the computer).

**Some Quick Hints**

- Log in with the username and password provided to you.
- For editing, you have many choices, including
    - gedit: a simple to use graphical editor. Can be started from a terminal window.
    - Eclipse: a complete programming environment. Should be in sidebar.
    - vi: old school…

  You can ask a staff member for help with these.
- If you need to print something, please ask one of the attendants.
- If you need to go to the bathroom, please ask one of the attendants to accompany you.
- Clarifications can be submitted through pc2team. This is preferred over asking attendants.
- All programs should read from standard input and write to standard output.
- There is a running time limit of 5 seconds on each solution, which should be plenty.
- See the last page for an intro to modular arithmetic.
- Until 30 minutes before the end of the contest, you will get a balloon for each solved problem, in a color that corresponds to the problem you solved. The colors of the problems are not disclosed, so unless you have a balloon of the same color you cannot tell which problems other teams have solved.

## 1. Seek-A-Word

You will be given traditional word search puzzles. Each puzzle will have a grid of letters and you will be asked to locate a given word. The letters of the word must be arranged consecutively in a straight line but in any direction. Your word may be going up, down, diagonally, forwards, or backwards. Your task will be to locate the word by the positions of the first and last letters. Your word may occur more than once, but you only need to locate one occurrence. If your word does not occur at all, you must display the appropriate message.

### Input/Output Description:

The input will start with a line containing an integer giving the number of puzzles. Each puzzle will consist of a line containing the word for which to search, a line containing the respective numbers of rows and columns of the grid, and the grid itself.

The output should start with a line containing the number of puzzles. Then, for each puzzle, the following output should be produced. If your word is found, your output should consist of one line displaying the respective row and column numbers of the first letter and one line displaying the respective row and column numbers of the last letter. (If there are multiple occurrences, only one should be output.) If your word is not found, you must output "word not found".

### Sample Input:

```
2
BET
3 4
A B C T
E F E T
A B Z T
TREAT
3 5
Z Y X W V
U T S R Q
P O N M L
```

### Corresponding Output:

```
2
1 2
3 4
word not found
```

## 2. Roman Numerals

Roman numerals, the numeric system used in ancient Rome, uses combinations of letters from the Latin alphabet to signify numerical values. From the 14th century on, Roman numerals began to be replaced by the more convenient Hindu-Arabic numerals that we use today. Nonetheless, Roman numerals are still used in some situations.

The basic Roman numerals are

| | |
|---|---|
| I | 1 |
| V | 5 |
| X | 10 |
| L | 50 |
| C | 100 |
| D | 500 |
| M | 1000 |

Numerals are placed left-to-right in order, largest first, and added together. I, X, C, and M can be repeated up to three times. The largest Roman numeral is MMMDCCCLXXXVIII = 3888. To avoid four identical numerals to appear in a row, the following exceptions apply: IV = 4, IX = 9, XL = 40, XC = 90, CD = 400, CM = 900.

Write a converter from Roman numerals to positive integers and back.

### Input/Output Description:

The input starts with a line containing the number of problems. For each problem, the input is a line containing either a Roman numeral or an (Hindu-Arabic) integer in the range [1 - 3888]. You are to output the number of problems followed by the corresponding conversions.

### Sample Input:

```
3
2013
MMXIV
2015
```

### Corresponding Output:

```
3
MMXIII
2014
MMXV
```

**3. Can You Build It?**

Some regular polygons can be constructed with just a straightedge and a compass. It has been proven that a regular *n*-gon is constructible if and only if the prime factors of *n* include only powers of two and at most one each of the *Fermat primes*. Fermat primes are prime numbers that are equal to $2^{2^{\wedge m}} + 1$ for some $m \geq 0$. For example, a regular decagon (10-gon) is constructible because $10 = 2 * 5$ but a regular nonagon (9-gon) is not constructible because $9 = 3 * 3$. Furthermore, a regular heptagon (7-gon) is not constructible because 7 is neither a power of two nor a number of the form $2^{2^{\wedge m}} + 1$, but a regular dodecagon (12-gon) is constructible because $12 = 2^2 * 3$.

Besides powers of two, for a regular *n*-gon to be constructible, the only possible factors of *n* less than one million are 3, 5, 17, 257, and 65537. Your job is to determine whether or not a regular polygon with a given number of sides is constructible with just a straightedge and compass.

**Input/Output Description:**

The input starts with a line containing an integer listing the number of problems. For each problem, a line contains a value for *n*. The output should start with a line containing the number of problems. For each problem, there should be a line containing the number of prime factors and a list of the prime factors in ascending order including repeats, and finally tell whether or not a regular polygon with that many sides is constructible by adding the token "constructible" or "non-constructible".

**Sample Input:**

```
3
120
45
786444
```

**Corresponding Output:**

```
3
5 2 2 2 3 5 constructible
3 3 3 5 non-constructible
4 2 2 3 65537 constructible
```

**4. Student Identifiers**

At Occam High School, students have 8-digit identifiers, the first digit cannot be 0. So, for example, 25473803 is a student identifier. No two students have the same identifier. However, teachers at Occam find those numbers too large to remember. Write a program that generates, for each class at Occam, the smallest positive integer M such that the list of student identifiers in that class, modulo M, still renders unique identifiers. See Appendix for a description of modular arithmetic.

**Input/Output Description:**

The input starts with a line containing the number of classes at Occam High School. This line is followed by a line for each class. Each such line starts with an integer containing the number of students in the class. This integer is followed by the list of student identifiers. The output should start with a line containing the number of classes. For each class, there should be a line containing the corresponding integer M.

**Sample Input:**

```
2
2 90980967 84959483
3 86632431 74363749 92938245
```

**Corresponding Output:**

```
2
3
5
```

**5. Eating M&Ms**

Little John likes to eat M&Ms. Everyone knows that the correct way to eat M&Ms is:

- Eating exactly one M&M at a time (since little John is trying to enjoy them for the maximum amount of time).
- Eating an M&Ms from a color that has the most number of M&Ms left.
- If there are ties on colors, follow your preference.

For instance, suppose that Little John likes red M&Ms the most, then green, and then blue, and that originally he has the following amount of each of these colors: Red - 3, Green - 4, Blue - 2. In this case, Little John will eat the M&Ms in the following order: Green, Red, Green, Red, Green, Blue, Red, Green, Blue. Little John loves a number X, and wants to know in advance what is the color of the $X^{th}$ M&Ms he will eat.

**Input/Output Description:**

The input will start with a number T, which indicates the number of test cases to follow. Each of the next T lines will contain a number N (1 <= N <= 50000) which is the number of colors, followed by N numbers, the number of M&Ms of each color. Colors are presented in order of preference, and are numbered from 1 to N (1 being the first color, and the favorite). Finally, there will be a number X.

The output should start with a line containing the number T, followed by one line per test case. For each test case you should output the color of the Xth M&Ms eaten. It is guaranteed that X is between 1 and the total number of M&Ms. Note that X can be very big (up to a billion) and simply "simulating the M&M eating process" would yield to a very slow solution. The sample inputs correspond to the example given in the problem statement.

**Sample Input:**

```
2
3 3 4 2 7
3 3 4 2 1
```

**Corresponding Output:**

```
2
1
2
```

**6. Great Expectations (Bullseye)**

Pip is tired of playing games with dice and has taken up darts. Your job is to help Pip determine how many points he can expect each time he throws a dart. You are to use the formula for expected value where you first find the product of each possible value and its probability of occurring and then add up all of these products. This formula may be written as follows:

$$E = x_1*p_1 + x_2*p_2 + x_3*p_3 + \ldots + x_i*p_i + \ldots + x_n*p_n$$

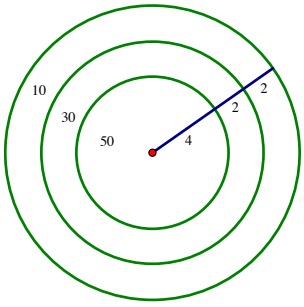
where $n$ stands for the number of possible values, each $x_i$ stands for one possible value, each $p_i$ stand for the corresponding probability, and $i$ ranges from 1 to $n$. The other formula you will need is the traditional formula for the area of the circle, $A = \pi r^2$. Lastly, please note that the probability of the dart landing in a certain region is the area of that region divided by the area of the smallest region containing all possible landing points. To keep it simple, we will stick to dartboards consisting of concentric circles where higher values are found closer to the center and we will ignore the possibility that Pip misses the dartboard entirely.

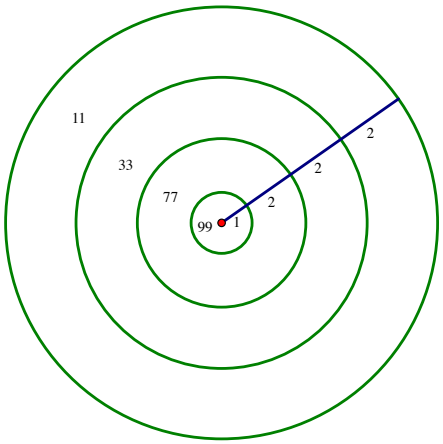
**Input/Output Description:**

The input starts with a line containing an integer containing the number of problems. For each problem, your program should read data representing the dartboard from the standard input. The first piece of data will be an integer representing the number of successive rings on the dartboard, including the bullseye. After that, your program should read in integers representing the respective radius and associated point value of each circle. Please note that the point value is equal to the point value associated with the smallest circle in which the dart lands (the largest possible point value). Your output should start with a line containing the number of problems, followed by the expected values, to three decimal places, each on a separate line.

**Sample Input:**

```
2
3
4 50
6 30
8 10
4
1 99
3 77
5 33
7 11
```




**Corresponding Output:**

```
2
26.250
30.755
```

## 7. Pancakes

You are working at The European Bakery, where one can get a stack of pancakes of any height. Each stack is served nicely sorted by diameter, the largest on the bottom. The chef produces a stack of pancakes in arbitrary order of diameter. It is your job to sort the pancakes, largest on the bottom, but you are not allowed to touch them. Instead, using a spatula, you repeatedly insert the spatula under one of the pancakes and flip the entire stack above it until the stack is sorted.

**Input/Output Description:**

The input starts with a line containing a number indicating the number of stacks of pancakes. This is followed by a line for each (unsorted) stack. Each such line starts with a number containing the number of pancakes in the stack. This is then following by the (integer) diameters of the pancakes, bottom-most first. The output should start with a line containing the number of stacks, followed with a line for each solution. Such a line consists of a list of numbers, one for each flip. Each number indicates the position where to insert the spatula, with position 0 meaning the bottom of the stack. That is, 0 means that the entire stack is flipped. 1 means that the bottommost pancakes stays in place, but the stack above it is flipped. And so on.

In the example below there are two stacks, one consisting of three pancakes, the other of four. The first stack is upside down, so the solution is to flip the entire stack (position 0). To sort the second stack, you first insert the spatula midway the stack (position 2) and flip it. Then you flip the entire stack (position 0) and you're done. Note that multiple solutions exist. You only have to produce a single solution. Evidently, a shorter solution is generally preferred over a longer one (but this is only taking into account by the judges if two or more teams are tied). If the stack is already sorted, then no flips are necessary, and so the solution would consist of an empty line.

**Sample Input:**

```
2
3 1 2 3
4 2 3 5 4
```

**Corresponding Output:**

```
2
0
2 0
```

## 8. Postfix Calculator

In “postfix notation”, an operation follows the operands. For example, “3 4 +” means add 3 and 4, while “3 4 + 5 *” means add 3 and 4, and then multiply the result by 5. We want you to write a “postfix calculator”. Such a calculator maintains a stack of numbers. Initially the stack only contains the number 0. A number in the input is pushed onto the stack. Operations in the input operate on the top of this stack:

| | |
|---|---|
| +: | replace the top two numbers on the stack with their sum |
| *: | replace the top two numbers on the stack with their product |
| =: | print (but do not remove) the number that is on the top of the stack |
| quit: | terminate the calculator |

+ and * do nothing if there is only one number on the stack. As an additional requirement, we want all arithmetic to be modulo 256 (see appendix). So "128 128 + =“ should output “0” because 128 + 128 = 0 (modulo 256).

**Input/Output Description:**

The input consists of a list of tokens separated by white space such as spaces or newlines. The last, and only the last, token is “quit”. The other tokens are numbers in the range [0, 255], and operators { +, *, = }. The numbers printed by the ‘=‘ operation should all be on a separate line.

**Sample Input:**

```
3 4 + = 100 * = quit
```

**Corresponding Output:**

```
7
188
```

# About “modulo” arithmetic

Modular arithmetic is arithmetic where numbers “wrap around” in a range. A well-known example is the “clock”. When it’s 9 o’clock, then 6 hours later it is 3 o’clock. Computers use modular arithmetic all the time, usually in a range [0, $k$ - 1] for some positive $k$, called the “modulus”. The modulus is often a power of 2. So, if the modulus is 256, then the range is [0, 255]. For example, 255 + 1 = 0 in modulo 256 arithmetic. Similarly, 250 + 10 = 4 in modulo 256 arithmetic, because 260 = 256 + 4.

Languages such as C, C++, Java, and Python provide a “modulo” operator, denoted

“$x$ % $y$”

where $x$ is a non-negative integer and $y$ is a positive integer. It yields the remainder of the integer division $x$ / $y$, which is an integer in the range [0, $y$ – 1].
For example, (7 + 5) % 10 = 2, because if you divide 12 by 10, the remainder is 2.
Similarly, (250 + 10) % 256 = 4.