*Problem 1*

# Tweening

You've been using jQuery for all your web development needs lately, and you were saddened to discover that despite all of jQuery's bells and whistles, jQuery doesn't allow color animation. For many CSS properties, like opacity, height, width, etc., you can specify a start and stop point for the animation and a time it takes to do the animation and jQuery will handle the rest. With colors, this just isn't the case.


Being the resolute programmer that you are, you've decided to write your own jQuery extension to do color in-between. You've been reading the API, and you've discovered that jQuery uses linear interpolation. So given two hex colors and a duration, you need to take each color component separately and interpolate linearly. Then, you'll need to combine them back again into an HTML color of the same format as the originals. However, the result of interpolation might not be an integer, so you have decided to always truncate to an integer.

Consider the case with `#ff0000` (red) and `#00ff00` (green) with a duration of 5000 milliseconds in length, and the middle time to interpolate for was 2500 milliseconds, the color would be `#808000`. This is because you are halfway between the start and stop time, so the red component will be halfway between `ff` (255) and `00` (0) which is 127.5 (which we truncate to 127, giving `7f` in hex), and the green will be halfway between `00` and `ff` giving the same result. The blue remains 0 because both endpoints are 0.

**Input**

Input will consist of a single line with an integer *n*, telling how many cases will follow. The next *n* lines will each have four tokens separated by a single space: a start hex color, a stop hex color, a duration time, and a middle time to calculate for. The duration time is guaranteed to be positive, and the middle time between 0 and the duration time.

**Output**

For your output, you should print the interpolated hex color on a single line. If during interpolation, the values become fractional, you should round to the nearest integer.

**Sample Input**

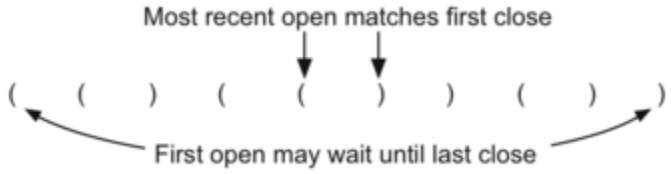
```
3
#ff0000 #00ff00 5000 2500
#ffffff #000000 100 30
#ff6666 #6666ff 1000 800
```

**Output Corresponding to Sample Input**

```
#7f7f00
#b2b2b2
#8466e0
```

*Problem 2*

# Balanced




We want to write a program that takes a string of opening and closing parentheses and checks to see whether it's *balanced*. We have exactly two types of parentheses for this problem: round brackets: () and square brackets: []. Assume that the string doesn't contain any other characters than these. This means no spaces, digits, letters, or other symbols.

*Balanced* parentheses require that there are an equal number of opening and closing parentheses. It requires that every opening parenthesis be closed in the reverse order opened. For example, `([])` is balanced, but `([)]` is not. It also requires that no closing parenthesis appears before an opening parenthesis.

**Input**
The file contains a positive integer *n* and a sequence of *n* strings of parentheses, one string per line. Each string is of length *n*, ($2 \leq n \leq 80$). You may assume each string contains only parentheses of type `()` and `[]`, and no other characters.

**Output**
Output each input string in the format below indicating whether or not it is balanced.

**Sample Input**
```
8
()
[]
([])
([()[]()])()
(([()]))
)(
((()(()))())
([)]
```

**Output Corresponding to Sample Input**
```
() is balanced
[] is balanced
([]) is balanced
([()[]()])() is balanced
(([()])) is not balanced
)( is not balanced
((()(()))()) is balanced
([)] is not balanced
```

*Problem 3*

# A Mathematical Exercise

Professor X has made a New Year's resolution to get into better shape. To that end, he decides to spend his lunch hour running on a treadmill. Prof. X soon discovers that running in place is quite boring and he searches for a math problem to reduce the tedium. One day while running, he notices that sometimes the time elapsed and miles traveled are similar. For example, after running a minute and sixteen seconds, the digital readout for time elapsed is 1:16, while the distance traveled reading is 0.116 miles. He wonders how often this type of similarity occurs during his runs. He defines the readings as similar if:

```
(minutes elapsed) + (seconds elapsed)/100 = 10 × (miles traveled)
```

where `miles traveled = speed × (time elapsed)`. Your task is to find all similar readings.

**Input**

The first line of input is an integer $T$, ($2 \leq T \leq 20$), representing the number of test cases. Each test case will begin with $n$, ($1 \leq n \leq 20$), the number of segments during the run. The next $n$-lines of input will each correspond with one segment. A segment will consist of a line with two inputs, $s$ and $d$, separated by a single space. The first $s$ is a floating-point value representing Prof. X's running speed $s$ in miles per hour for duration of $d$ seconds where $d$ is a positive integer. Prof. X never runs at the same speed for more than 30 minutes.

**Output**

For each test case, output the run number followed by, in ascending order, the times rounded to four decimal places when a similar reading occurred. If the readings are similar throughout an interval, only report the beginning time followed by an asterisk. Note that there is a special speed which is 3.6 mph. If the readings are similar and the speed of the treadmill is 3.6 mph, then the readings will remain similar until the start of the next minute. Always assume the clock resets to zero when we start a new run.

**Sample Input**

```
3
2
5.0 90
8.0 90
2
6.0 60
3.6 300
1
8.0 900
```

**Output Corresponding to Sample Input**

```
Run #1: 0:00.0000, 1:34.0909, 2:06.8182
Run #2: 0:00.0000, 1:00.0000*
Run #3: 0:00.0000
```

*Problem 4*

# Curve Killer

Bob has finally graded his assembly tests and is passing them back out to his class. Unfortunately (but unsurprisingly), the grade average is much lower than it should be. Being the generous professor he is, he informs his students that he will curve the grades. This is how he will do so. Given the set of grades, we can calculate the average grade as follows:

$$\mu = \frac{\sum_{i=0}^{n} x_i}{n}$$

We can also calculate the standard deviation of the set in this manner.

$$\sigma = \sqrt{\frac{\sum_{i=0}^{n} (x_i - \mu)^2}{n-1}}$$

Given a new standard deviation and mean, we can scale the points such that the data set's mean and standard deviation change to this new value. Your job is to take the data set, and alter it such that the standard deviation and mean are changed to the new provided values.

**Input**

The input will consist of multiple test cases of the problem beginning with a line containing a number *n*, ($0 \leq n \leq 20$), representing the number of students for that case. The next line will contain two floating point numbers separated by a single space, the new mean and standard deviation in that order. This is followed by *n* lines each containing a student's integer grade. The last line will contain an *n* value of 0. Do not process this case.

**Output**

For each test case, you should print out each new grade on a new line. Print the grades in sorted order for each case. The sorted lists for each case should appear next to each other. Furthermore, print them as integers without rounding, i.e., 98.9 truncates to 98. Lastly, grades with values below zero are not permitted. If a grade falls below zero, it must be printed as zero.

**Sample Input**

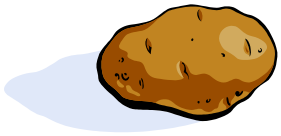
```
3
75.0 10.0
42
55
57
3
80.0 5.0
23
90
27
0
```

**Output Corresponding to Sample Input**

```
63
79
81
76
77
85
```

*Problem 5*

# Josephus



The Josephus problem is the following game. There are *n* people, numbered 1 to *n*, sitting in a circle. Starting at person 1, a hot potato is passed. After *m* passes, the person holding the hot potato is eliminated, the circle closes ranks, and the game continues with the person who was sitting after the eliminated person picking up the hot potato. The last remaining person wins.

The Josephus problem arose in first century A.D., in a cave on a mountain in Israel, where Jewish zealots were besieged by Roman soldiers. The zealots voted to form a suicide pact rather than surrender to the Romans. Josephus suggested the game, and rigged the game so he would get the last lot. That is how we know about this game; in effect Josephus cheated!

Write a program that takes as input the number of people in the circle, *n*, and the number of passes, *m*. Display the sequence of people removed. The final person printed is deemed the winner. Note that person 1 is always the designated first person and starts with the hot potato. You may assume the number of passes remains constant for each test case, and once person number *k* is eliminated, then the next round begins with person *(k+1)* holding the hot potato.

**Input**

The first line of input is an integer *T*, ($2 \leq T \leq 20$), representing the number of test cases. This is followed by *T* lines, each containing a test case with two integers separated by a single space. The first integer represents *n*, the number of people where ($2 \leq n \leq 40$), and the second represents *m*, the number of passes where ($0 \leq m \leq 50$).

**Output**

Your program should print the sequence of *n* people removed. Each person should be separated by a single space.

**Sample Input**

```
5
5 0
5 2
7 3
10 10
6 8
```

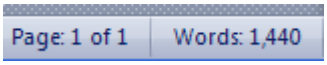
**Output Corresponding to Sample Input**

```
1 2 3 4 5
3 1 5 2 4
4 1 6 5 7 3 2
1 3 6 10 8 9 5 2 4 7
3 1 2 6 4 5
```

*Problem 6*

# Word Count


*Microsoft Word* now automatically displays the number of words in your documents on the status bar at the bottom of the workspace.




You can even click on Words from the status bar to find the total characters and lines in your file too. This is nothing new on the *Linux* operating system which has always included a word count command called `wc` that displays the number of *lines*, *words*, and *bytes* in a text file to the screen. These are each defined as follows.

- *lines* — total end of line characters in the file.
- *words* — total *tokens* in the file. A *token* starts and ends with a printable, non-whitespace character. A *whitespace character* is a space, tab, or end of line character.
- *bytes* — total characters in the file including all *whitespace characters*.

You have been asked to implement the classic `wc` *Linux* system command to calculate the total lines, words, and bytes as defined above.

**Input**

Your program should accept a single test case consisting of one or more strings of length *n*, ($2 \leq n \leq 80$). Input is terminated by the end of file, and the last character in the file will always be an end of line character.

**Output**

Output the total number of lines, words, and bytes exactly as shown below.

**Sample Input**

```
"Your time is limited, so don't waste it living someone else's life.
Have the courage to follow your heart and intuition.
They somehow already know what you truly want to become.
Everything else is secondary."
-Steve Jobs, 2005 Stanford Commencement
```

**Output Corresponding to Sample Input**

```
Lines = 5
Words = 40
Bytes = 250
```