Easy # 1

**N-CHOOSE-K**

Write a program which repeatedly reads in a non-negative integer *N* followed by a second non-negative integer *K*, and prints the number of ways you can choose *K* items from among *N* total items. We call this “N-choose-K”, and the formula for “N-choose-K” is:

Input Format:

Input contains *N* and *K* respectively.

Output Format:

Single integer indicating the result given the integers *N* and *K.*

Constraints:

0 <= K <= N

Sample input: Sample output:

7 2 21

**Solution: *Java/NchooseK.java*** *[10 lines]*

***C++/NChooseK.cpp*** *[16 lines]*

***C/NChooseK.c*** *[15 lines]*

***Python/nchoosek.py*** *[7 lines]*

Easy # 2

**WEIRD NUMBER**

(Shortest possible solution)

Given an integer N, check the following:

* If N is odd, print “Weird”
* If N is even and in between 2 and 5 (inclusive), print “Not Weird”
* If N is even and in between 6 and 20 (inclusive), print “Weird”
* If N is even, and N > 20, print “Not Weird”

Constraints:

1 <= N <= 100

Sample Input: Sample Output:

3 Weird

Sample Input: Sample Output:

24 Not Weird

**Solution: *Java/WeirdNumber.java*** *[6 lines]*

***C++/weirdnumber.cpp*** *[6 lines]*

***C/WeirdNumber.c*** *[7 lines]*

***Python/weirdnumber.py*** *[2 lines]*

***Shell/weirdnumber.sh*** *[8 lines]*

Easy # 3

**SMOKING CIGARETTE**

Peter has *N* cigarettes. He smokes them one by one, keeping all the butts. Out of K > 1 butts, he roll a new cigarette. How many cigarettes can Peter have?

Constraints:

T <= 10 2 <= N, K <= 1 000 000 000

Sample Input: Sample Output:

3 5

4 3 14

10 3 124

100 5

**Solution: *Java/Cigarettes.java*** *[14 lines]*

***C++/cigarette.cpp*** *[19 lines]*

***C/cigarette.c*** *[21 lines]*

***Python/cigarette.py*** *[9 lines]*

***Shell/cigarette.sh*** *[17 lines]*

Easy # 4

**FILLING JARS**

Animesh has *N* empty jars, numbered from 1 to *N*, with infinite capacity. He performs *M* operations. Each operation is described by 3 integers, *a*, *b*, and *k*. Here, *a* & *b* are indices of the jars, and *k* is the number of candies to be added inside each jar whose index lies between *a* and *b* (both inclusive). Can you tell the average number of candies after *M* operations?

Input Format:

First line contains two integers, *N* and *M*, separated by single space.

*M* lines follow; each of them contains three integers, *a*, *b,* and *k*, separated by spaces.

Output Format:

A single line containing the average number of candies across *N* jars, rounded down to the nearest integer (floor of the number).

Constraints:

3 <= N <= 107 1 <= M <= 105 1 <= a <= b <= N 0 <= k <= 106

Sample Input: Sample Output:

5 3 160

1 2 100

2 5 100

3 4 100

**Solution: *Java/FillingJars.java*** *[14 lines]*

***C++/FillingJars.cpp*** *[15 lines]*

***C/FillingJars.c*** *[15 lines]*

***Python/fillingjars.py*** *[6 lines]*

***Shell/FillingJars.sh*** *[18 lines]*

Easy # 5

**PINGPONG PAUL**

Paul Eigon recently got a new job at a famous company in town. They have all sorts of cool stuff at the office. Video games, a swimming pool, a sauna, and even a ping-pong table! Paul really loves his enjoyable new workplace.

Mr. Eigon plays a lot of ping-pong at work. While he is playing, he enjoys having a nice conversation with his opponent. Discussions are jolly fun to have, but they can also a bit distracting, enough for Paul to sometimes completely forget whose turn it is to serve the ball the upcoming round!

Paul's ping-pong games are played with simple rules. The game is played by two players. The game starts with one of the players serving the ball, and that player does so during the *N* first rounds. After that, the other player serves the ball for *N* rounds, and so on. Each round is won by exactly one of the players, and a player gets one point for each won round (there can be no draws). When one of the players has reached a certain number of points, that player wins the game.

Paul is in desperate need of a new way to figure out which player should serve the ball the upcoming round, and he wants you to help him. You should write a program that given *N*, and the current score of the game, prints out whose turn it is to serve the ball the upcoming round. His plan is to run your program on his secret high technology pocket computer, in order to impress his opponent. And by the way, Paul always serves the first ball (if that makes it easier for you).

Input Format:

Input consists of three integers **N**, **P**, and **Q** on a single line. **N** denotes the number of serves a player will perform in a row before it is the other player's turn to serve the ball. **P** denotes Paul's current score in the ongoing ping-pong game, and **Q** denotes his opponent's score.

Output Format:

Output should consist of a single word on a single line. If it is Paul's turn to serve the ball, your program should output “paul”. Otherwise, “opponent”.

Constraints:

1 <= N <= 109

0 <= P <= 109

0 <= Q <= 109

Sample Input: Sample Output:

5 3 7 paul

**Solution: *Java/PingPongPaul.java*** *[7 lines]*

***C++/PingPongPaul.cpp*** *[12 lines]*

***C/PingPongPaul.c*** *[12 lines]*

***Python/pingpongpaul.py*** *[6 lines]*

***Shell/pingpongpaul.sh*** *[14 lines]*

Average # 1

**ARRAYS.** Insert, Delete, and Search element/s in Arrays

**Solution: *Java/Arrays.java*** *[71 lines]*

Average # 2

**iPhone PASSCODE GENERATOR.** Generate all possible pass codes in an Apple iPhone device with four-pin pass code system.

**Solution: *Java/PasscodeGenerator.java*** *[17 lines]*

***Python/PasscodeGenerator.py*** *[9 lines]*

Average # 3

**ARCTAN SERIES**

The Maclaurin series for arctan(x) is a formula which allows us to compute an approximation to arctan(x) as a polynomial in x. The formula is:

arctan(x) = x – x3/3 + x5/5 - x7/7 + x9/9 - x11/11 + …

Write a program which reads in a double x, and a positive integer k, and prints out the partial sum from the first k terms in this series and also prints out the value of Math.atan(x), which computes the arctangent directly.

Hint #1: If the user enters 0.5 for x, and 3 for k, you should compute:

0.5 – 0.53/3 + 0.55/5 = 0.464583...

That is, the first 3 terms of the series with x = 0.5. This turns out to be a good approximation, even with only 3 terms, as:

arctan(0.5) = 0.4636475...

Input Format: The input contains two integers X and K respectively.

Output Format: The result of the equation given an integer X and K.

Sample Input: Sample Output:

0.5 3 0.464583333333333

**Solution: *Java/ArctanSeries.java*** *[17 lines]*

***C++/ArctanSeries.cpp*** *[24 lines]*

***Python/ArctanSeries.py*** *[14 lines]*

Average # 4

**NOTEBOOK**

Emma is writing bits (1's and 0's) in her notebook. Every second, she writes more bits.

In the 1st second, she writes two bits in her notebook. She starts with 0 and alternates between 0 and 1. They look like this:

0 1

In the 2nd second, she writes three more bits in her notebook. This time, she starts with 1 and keeps alternating between 0 and 1. Now, they look like this:

0 1

1 0 1

After 6 seconds, her notebook looks like this:

0 1

1 0 1

0 1 0 1

1 0 1 0 1

0 1 0 1 0 1

1 0 1 0 1 0 1

If you count carefully, you can see that Emma wrote 15 1's in her notebook after 6 seconds. She wants to know how many 1's she can write in T seconds.

Input Format: The input contains a single integer T, indicating the time in seconds.

Output Format: Print the number of 1's Emma can write in T seconds.

Constraints: 1 <= T <= 1 000 000 000

Sample Input: Sample Output:

3 5

**Solution: *Java/NotebookBits.java*** *[11 lines]*

***C++/Notebook.cpp*** *[10 lines]*

***C/Notebook.c*** *[10 lines]*

***Python/notebook.py*** *[7 lines]*

***Shell/Notebook.sh*** *[16 lines]*

Average # 5

**HELIOCENTRIC**

Nicolaus Copernicus, born in Torun in the Kingdom of Poland in 1473, was the first to collect sufficient evidence to show that the heliocentric view of the solar system (that the Sun was at the center of the planetary orbits) could compete with the Ptolemaic view (that the Earth was at the center). Copernicus viewed each planet as revolving in a circular orbit with the Sun as the common center of each circle. This new model, which was later improved by many other astronomers, made it much simple to understand and predict motions of the planets.

Consider two of the planets in Copernicus' orbital system: Earth and Mars. Assume the Earth orbits the Sun in exactly 365 days, and Mars orbits the Sun in exactly 687 days. Thus, the Earth's orbit starts at day 0 and continues to day 364, and then starts over at day 0. Mars orbits similarly, but on a 687 day time scale. We would like to find out how long it will take until both planets are on day 0 of their orbits simultaneously. Write a program that can determine this.

Input Format: The first line of input contains an integer N indicating the number of test cases. N lines follow. Each test case contains two integers E and M. These indicate which days Earth and Mars are at their respective orbits.

Output Format: For each case, display the case number followed by the smallest number of days until the two planets will both be on day 0 of their orbits. Follow the format of the sample output.

Constraints: 0 <= E < 365 0 <= M < 687

Sample Input Sample Ouput:

5 Case 1: 0

0 0 Case 2: 1

364 686 Case 3: 5

360 682 Case 4: 239075

0 1 Case 5: 11679

1 0

**Solution: *Java/Heliocentric.java*** *[12 lines]*

***C++/Heliocentric.cpp*** *[18 lines]*

***Python/Heliocentric.py*** *[8 lines]*

Difficult # 1

**POSITIVE PENTAGON**

Five random integers are placed at the corners of a pentagon. Typically, some of these numbers will be negative, but their sum is guaranteed to be positive. The goal is to get rid of all the negative numbers through a balanced process of subtraction and negation.

Starting with the lowest of the negative numbers, we negate the number (thus making it positive), and then subtract that value from each of its two neighbors. The sum of the new numbers will remain the same as the original pentagon, so the pentagon is still “balanced”. This process (finding the lowest of the negative numbers, negating it, and subtracting it from its neighbors) is then repeated until all of the numbers are non-negative.

During any step, if the lowest negative number appears at more than one corner, use the one that would be found first, if you started at the top corner and traversed in clockwise direction.

Given the original five numbers at the corners of a pentagon, output the Positive Pentagon that can be created by following this process. You may assume this process will always make a pentagon “pentastic” in at most 1000 steps.

Input Format: An input contains a single pentagon description. A pentagon description will consists of exactly 5 integers, which are in the range of -999 to 999 (inclusive), and which sum up to a positive number less than 1000.

There will be exactly one space between numbers, and no leading or trailing spaces on the input lines. Positive numbers in the input will not have leading '+' sign. The numbers are given in a clockwise order around the pentagon, starting from the top. This means that the 1st and 3rd numbers are neighbors of the 2nd number, the 5th and 2nd numbers are neighbors of the 1st number, and so on.

Output Format: For the input pentagon, output the Positive Pentagon that results from applying the process described above. Output the numbers for each corner using the same ordering and method used in the input, with number for the top corner first, and the others following a clockwise order. Output one space between output numbers.

Constraints: -999 <= a1, a2, … ,a5 <= 999

Sample Input: Sample Output:

2 -1 5 7 -4 1 2 2 3 1

**Solution: *Java/Pentagon.java*** *[73 lines]*