Robotique
CRC
Robotics

CRC

PROGRAMMING CHALLENGE

ARCANUM 2022

BLOCK A

PROBLEM BOOKLET

A FEW NOTES

- The complete rules are in section 5 of the rulebook
- This challenge was made for small teams of programmers, we do not recommend having more than 4 players on your team
- The all-around bonus applies only if you complete one/two problem(s) in every category of BOTH block A AND block B
- You will NOT be able to retry these problems during the 2nd day (block B) of the competition, so plan your strategy accordingly
- Anything submitted after the time limit will not be counted
- The judges should be able to simply copy paste the inputs all in ONE bunch and receive the outputs all in ONE bunch for correction all in the console
- Program smart: making an input reader should only be done once and then tweaked for other problems
- Your team is not expected to be able to complete everything easily so don't be shocked if you see a large amount of problems
- You are also free to come ask us questions on the problems!

STRUCTURE

Each problem contains a small introduction like this about the basics of the problem and what is required to solve it. Before every problem, in the title, will appear both its category and its index within that category. The number in parentheses corresponds to the number of points given for a successful solve.

Input and output specification:

In these two sections, we specify what the inputs will be and what form they will take, and we also say what outputs are required for the code to produce and in what format they shall be.

Sample input and output:

In these two sections, you will find a sample input (that often has multiple entries itself) in the sample input and what your program should produce for such an input in the sample output.

Explanation of the first output:

Sometimes, the problem might still be hard to understand after those sections, which is why there will also be a usually brief explanation of the logic that was used to reach the first output from the first input.

Table of Contents

TEXT TREATMENT (TT)

TT1: Cost of grammar (15)

Recently, one of the robots on the ship has learned how to play vocabulary board games, but can't seem to understand the logic between different letter values. The robot doesn't like the lack of consistency between the letters that would be considered rare in french and those that would be considered rare in english, for example. To replace it, the robot creates his own scoring system that lets the latin alphabet do the work. You will have to compute the new values of a few words according to this system. It starts with the letter A being worth 1. Then, the value is incremented by 1 for every further letter in the alphabet. This way, we have A=1, B=2, C=3 and so on until Z=26. The values of the letters add up together to give the total value of a word.

Input specification:

The first line contains an integer *n* that corresponds to the number of entries needing to be transformed. Every one of the *n* lines below contains a word for which the total value will need to be calculated. All words are in capital letters. Given words will not contain any accents or special symbols.

Output specification:

For every word in the input, you will have to give the total value corresponding to that word.

Sample input:

```
5
ALVEOLE
BOUTEILLE
PEPPER
YEET
CRC
```

Sample output:

```
72
101
76
55
24
```

Explanation of the first output:

For ALVEOLE, we get 1+12+22+5+15+12+5=72.

TT2: cApiTaLs (20)

The main computer has problems with the MAJ key: it activates and deactivates randomly. To make any section of text legible, you will have to change capital letters for lowercase ones and vice versa. If it's the first letter before a final punctuation mark, (.!?)you will have to make sure that it is capital. Except for the first letter of each word, you will have to make sure every capital letter becomes lowercase. The first letter of a word will not be modified if it is capitalized as you do not know if it is a proper noun or not.

Input specification:

The first line contains an integer *n* that corresponds to the number of entries needing to be transformed. Every one of the *n* lines below contains the string to correct. Given words will not contain any accents or special symbols.

Output specification:

For every string in the input, you will have to give the corrected string with the right capital letters.

Sample input:

```
2
lorem ipSUm dolor sit AMet, consectetur aDIpiscing ELIT. Nulla hendrerit maXImus erAT, sIT amet consectetur aNte hendrerit id. nulla nON auCtor diam.
la cRC eSt UnE ComPeTItIOn dE rOBoTiQue. tRop cOOL!
```

Sample output:

```
Lorem ipsum dolor sit Amet, consectetur adipiscing Elit. Nulla hendrerit maximus erat, sit amet consectetur ante hendrerit id. Nulla non auctor diam.
La crc est Une Competition de robotique. Trop cool!
```

Explanation of the first ouput:

The first letter is capitalized because it starts the string. In the remainder of the first sentence, only the words Amet and Elit have to keep their first letter capitalized. Every other letter is changed to lowercase. In the following sentences, we can see that the first word (Nulla) starts with a capital letter, as it is the first word of a new sentence.

TT3: Caesar cycle decoding (20)

After messing with a few *caesar ciphers* and a few secret keys to encode some secret messages, somebody became aware of those methods and started to crack the code every time. To counter this, the terrestrial base of operations came up with a new cipher: Caesar's cycle. It

consists of changing the order of the letters by cycling through them with a predetermined *offset*. Therefore, encoding “orange” with an *offset* of 1 would give “eorang”, for example. You will have to reverse it to decipher given words and find the right words by applying the *offset* the other way around.

Input specification:

The first line contains an integer *n* that corresponds to the number of entries needing to be transformed. Every one of the *n* lines below contains a word that will have to be deciphered followed by the offset that will need to be applied towards the left of the word (the first letter becomes the last letter of the word) to get the right word.

Output specification:

For every word in the input, you will have to give the deciphered word in the output.

Sample input:

```
4
essenobl 4
mpereure 7
mercury 0
erjardini 2
```

Sample output:

```
noblesse
empereur
mercury
jardinier
```

Explanation of the first output:

We have to cycle the letters 4 times to the left. Doing it once gives ssenoble. After that, we get senobles, enobless and finally, noblesse.

DATA SORTING (TRI)

TRI1: A bit of order (25)

To make sure you keep the intel on the planets you have already visited, you want to organize them in a way that you will be able to find them again quickly. To accomplish that, you create an organized set that depends on the first letter of the planet . To do so, you will have to use the ASCII value of the first letter to find its position in the array. If the array had 100 *cases*, the "a" character would be in the 97th *case* (starting from 0) and the "f" character would be in the 2nd *case* (starting from 0) because the value of "f" exceeds the last position of the array, causing us to continue counting from the start once the end is reached. (97 mod 100 = 97, and 102[for f] mod 100 = 2) All planets go in different cases in the final array.

Input specification:

The first line contains an integer *n* that corresponds to the number of entries needing to be transformed. Every one of the *n* lines below contains the length of the array (which is bigger than the number of planets) followed by the names of the planets to sort out.

Output specification:

For every input, you will have to give the array with the planets in the right order, empty cases being 4 spaces and all cases being divided by ";". The array ends by a last ";".

Sample input:

```
2
7 neptune venus pluton terre
4 Mars Jupiter
```

Sample output:

```
pluton;    ;    ;    ;terre;neptune;venus;
    ;Mars;Jupiter;    ;
```

Explanation of the first output:

7 indicates the number of cases in the final array. Starting with "neptune", the first letter is "n" and its ASCII value is 110. We would then put "neptune" at the 110th case, but the array being only 7 cases long, we are required to continue from the beginning every time we reach the end. In this case, after going through the array 15 times, we finally get 5 (110 mod 7 = 5). We then put "neptune" at the 5th case counting from the start of the array. We then repeat this process for every planet in the input. **Reminder: counting the cases of an array that is 7 cases long goes from 0 to 6 in programming.**

TRI2: Dictionary! (30)

The ship's main file cabinet crumbled during an asteroid impact. All files initially contained fell to the floor, in a total disorder. Naturally, it would be much quicker to teach a robot how to sort those files than to do it yourself as there are at least thousands of them! You will have to sort all file names by alphabetical order.

Input specification:

The first line contains an integer *n* that corresponds to the number of words (all different) contained in the dictionary. Every one of the *n* lines below contains a word that will have to be included in the dictionary. Given words will not contain any accents or special symbols.

Output specification:

In the output, you will have to give the words sorted by alphabetical order (so *aaaaaa* would come first and *zzzzzz* would come last). A line contains one word only.

Sample input:

```
6
amenagement
philosophie
tricherie
amour
parapluie
environnement
```

Sample output:

```
amenagement
amour
environnement
parapluie
philosophie
tricherie
```

Explanation of the first output:

The words starting with a are obviously the first ones. Because amenagement and amour start the same way, we have to look further. The first distinction comes at the 3rd letter, where e comes before o, putting amenagement in front of amour. The same thing goes for parapluie and philosophie at the 2nd letter. E finds itself between a and p for environnement, and t comes after p for tricherie.

TRI3: Exoplanets classification (30)

Multiple exoplanets have been discovered in the galaxy up until now. These kinds of discoveries often make us ask ourselves if the human race could inhabit those planets. In this case, with Earth being already a bit overpopulated, we will only look at the habitable space.You will have to sort given exoplanets by habitable space from highest to lowest. One can find a planet's surface area with this formula:

$$A_P = 4\pi R_P^2$$

Input specification:

The first line contains an integer *n* that corresponds to the number of exoplanets to consider. Every one of the *n* lines below contains the planet's name, its radius (in km) and the percentage (so from 0 to 100) of its surface area that is habitable.

Output specification:

For every planet in the input, your program will have to compute its habitable space (surface area multiplied by habitable percentage). After that, it will have to sort the planets from highest habitable space to lowest. Every line will have to contain the planet's name and its habitable space (in km^2) rounded to the closest integer.

Sample input:

```
4
Kepler-69c 10894 37.38
Arcadium-14d 8799 68.88
Entractam-37k 3751 0
Smokybaconus-72a 14253 4.13
```

Sample output:

```
Arcadium-14d 670146321
Kepler-69c 557473083
Smokybaconus-72a 105432010
Entractam-37k 0
```

Explanation of the first output:

Multiplying the surface area calculated using the formula above multiplied by the habitable percentages, we find the habitable spaces above. Because 670 millions of km^2 are way bigger than 557 millions and 105 millions, Arcadium-14d is the first output.

MATHEMATICAL COMPUTING (MTH)

MTH1: Prime numbers (15)

The ship's calculator was completely bamboozled following the interaction with the magnetic field of a star in the vicinity. We will have to reteach him the basics, starting with prime numbers. We are now looking to compute if a given number is prime. It is important to remember that a prime number is only divisible by 1 and itself. For example, 12 is not a prime number, as it is also divisible by 2, 3, 4 and 6, while 13 is in fact a prime number.

Input specification:

The first line contains an integer *n* that corresponds to the number of entries needing to be transformed. Every one of the *n* lines below contains an integer between 2 and 10000.

Output specification:

For every given number, your program will have to return "prime" if it is prime, or "non prime" if it is not prime, followed by one of its dividers as a counter-example. Any divider except 1 and the number itself will do.

Sample input:

```
5
8991
578
97
117
1373
```

Sample output:

```
non prime 37
non prime 2
prime
non prime 13
prime
```

Explanation of the first output:

8991 is not prime. In fact, $8991=3^5 \times 37$. We can therefore give as a counter-example 3, 9, 37, 3x37, etc. Any of those dividers works and we chose 37 here.

MTH2: Quadribonacci (20)

The Fibonacci sequence, which consists of adding the *n-1* and *n-2* terms to obtain the *n*-th term, has beautiful properties and a ton of applications. However, the simple Fibonacci sequence did not seem like a hard enough test to draw conclusions about the good functionment of the calculator's recursion module. To do so, we will instead use a modified version of the Fibonacci sequence that adds the 4 previous terms instead of only the usual 2. For example, for the 5th and 8th terms:

$$n_5 = n_4 + n_3 + n_2 + n_1$$

Or:

$$n_8 = n_7 + n_6 + n_5 + n_4$$

Input specification:

The first line contains an integer *n* that corresponds to the number of entries needing to be transformed. Every one of the *n* lines below contains the first 4 terms (in order) required to initialize the modified Fibonacci sequence. (so $n_1\ n_2\ n_3\ n_4$)

Output specification:

For every line of initializing terms, you will have to calculate and print the first ten terms of the modified sequence INCLUDING the first 4 given in the input.

Sample input:

```
3
1 1 1 1
1 2 3 4
12 5 7 31
```

Sample output:

```
1 1 1 1 4 7 13 25 49 94
1 2 3 4 10 19 36 69 134 258
12 5 7 31 55 98 191 375 719 1383
```

Explanation of the first output:

Adding the first four terms together, we get 1+1+1+1=4. After that, we repeat the recipe for the following terms: 1+1+1+4=7, 1+1+4+7=13 and so on. 94 being the 10th term of the sequence, we stop after reaching it.

MTH3: The homie Pythagoras (25)

We will then test the calculator with a theorem elementary to the generalization of distances in more than one dimension: the infamous Pythagoras' theorem. This theorem tells us that, for every right-angled triangle, the sides length obey the following relation:

$$a^2 + b^2 = c^2$$

Any combination of three integers that obey this theorem are called pythagorean triples. We would then like to know if there exists a pythagorean triple for a given side length *a*.

Input specification:

The first line contains an integer *n* that corresponds to the number of entries needing to be transformed. Every one of the *n* lines below contains the number *a* for which we want to find a triple. **The given number is guaranteed to be the smallest in the triple.**

Output specification:

For every number *a* in the input, you will have to give the 3 numbers *a b c* of a pythagorean triple for *a* with the smallest *b* if there exists multiple triples for that *a*. Following the given equation, it becomes clear that *c* will also be the smallest. If there is no triple for *b* such that $a < b < 1000$, where 1000 corresponds to the superior limit (excluded) to test, you will instead have to write "pas de triple pour" *a*, which is the given number in the input.

Sample input:

```
5
78
43
656
65
2
```

Sample output:

```
78 104 130
43 924 925
pas de triple pour 656
65 72 97
pas de triple pour 2
```

Explanation of the first output:

Iterating *b* from *a* to 1000, we find that there is indeed an integer *c* that confirms this relation and gives a triple. We then print this triple in the output.

MTH4: Completely based (40)

Humans are used to doing computations in base 10, but it can be useful for a computer to have some numbers or variables written in other bases like the notorious base 2 (binary), base 16 (hexadecimal), for example. You will then have to make sure that the calculator can convert a base 10 number to a number written in another base, considering that a base 10 number is written so:

$$107\ (base\ 10) = 1 \times 10^2 + 0 \times 10^1 + 7 \times 10^0$$

The same number in base 7 would look like this:

$$107\ (base\ 10) = 212\ (base\ 7) = 2 \times 7^2 + 1 \times 7^1 + 2 \times 7^0$$

Input specification:

The first line contains an integer *n* that corresponds to the number of entries needing to be transformed. Every one of the *n* lines below contains the number to convert followed by the base in which it will need to be written. The base can range from 2 to 16, considering that we commonly use A to put 10 at a digit, B for 11, C for 12, D for 13, E for 14 and F for 15.

Output specification:

For every number to convert in the input, you will have to write it in the correct base using the appropriate digits. The first digit of every output cannot be zero.

Sample input:

```
4
284 5
2177 13
982 9
4236 3
```

Sample output:

```
2114
CB6
1111
12210220
```

Explanation of the first output:

The first few powers of 5 are 1, 5, 25, 125 and 625. Writing 284 in base 5, we get 2x125 + 1x25 + 1x5 + 4x1, which adds up to 284 in base 10.

TWO-DIMENSIONAL PROBLEMS (2D)

2D1: Trajectory (30)

We now wish to visualize the trajectory of the ship on a simple 10 by 10 grid. The ship starts from the left and goes to the right with constant speed on the x axis. You will have to compute the new position in y for every movement on the x axis. The y acceleration changes the y speed, which then changes the y position for every displacement by one unit in x. The origin (0,0) is the point in the bottom-left corner of the grid.

Input specification:

The first line contains an integer *n* that corresponds to the number of entries needing to be transformed. Every one of the *n* lines below contains the starting height of the ship followed by its initial speed (positive going upwards) and its initial acceleration (positive also upwards), all integers. A height of 10 is outside of the grid as we count from 0.

Output specification:

For every input, you will have to generate the 10 by 10 grid given by the trajectory of the ship. The trajectory is marked by x and spaces with no x in the grid become points. There is a space between each component of the grid in the x axis.

Sample input:

```
3
4 4 -1
-3 -3 1
11 -1 0
```

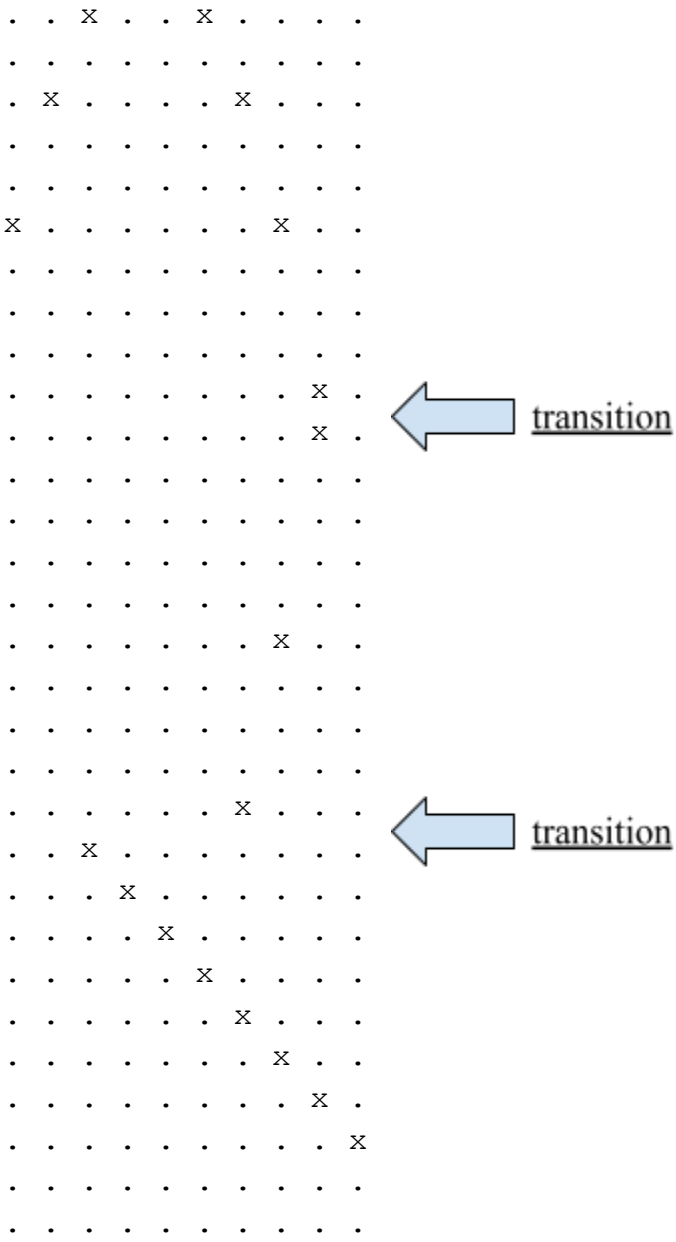
First output (4 4 -1):

```
. . x . . x . . . .
. . . . . . . . . .
. x . . . . x . . .
. . . . . . . . . .
. . . . . . . . . .
x . . . . . . x . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . x .
```

Explanation of the first output:

The height of the first “x” is 4 when we take the bottom-left corner as (0,0). The 2nd x is only 3 units higher, because the -1 acceleration bumps the speed down from 4 to 3. For the 3rd x, the speed goes down to 2 and we go 2 units higher. Then, the ship gets out of the grid at the top, so there is nothing to print mais still the height to compute. We then go back down inside the grid to eventually exit the grid downwards at (8, 0).

Complete sample output:



2D2: Trip planning (45)

Inside the Arcanum belt, you will have to dodge the incoming asteroids. You choose to graphically represent your trip in order to dodge those space rocks. Your starting point will be a "@" in the top-left corner and your movements will result in a "-" horizontally or in a "|" vertically. Letters will be used to simplify the directions' notation: U for upwards, D for downwards, L for the left and R for the right. The planned trip will never go backwards or cross its own path.

Input specification:

The first line contains an integer *n* that corresponds to the number of entries needing to be transformed. Every one of the *n* lines below contains the list of movements planned.

Output specification:

For every line of instructions, you will have to print out the complete planned trip starting with "@".

Sample input:

```
2
DDRRRRRRRDDDDDRRRRRRRRRRRRRRDDDDLLLLUULLLLLLLLLLLLLUUU
RRRRRRRDDDLLLLDDDDRRRRRRRRRRRRUUUUUUL
```

Sample output:

```
@
|
|-------
        |
        |
        |
    |   |
    |   |---------------
    |                   |
    ---------------|    |
                   |    |
                   ----|
@-------
        |       -|
        |        |
    ----|        |
    |            |
    |            |
    |            |
    |------------
```

Explanation of the first output:

The top left corner contains the starting symbol “@”. Following that, we go down 2 units corresponding to “DD”. Then we go right by 7 units, corresponding to “RRRRRRR”. We continue downwards with the last of instructions until the end.

2D3: Labyrinth (65)

The next thing to do is to send your rover on a planet to pick up some samples without it getting destroyed. You are in possession of a map of the most dangerous parcel of terrain to cross, where obstacles are market by "#" and the only exit is the gap going upwards in the otherwise continuous line of "#". Starting from the left side (it is up to you to find the correct height), you will have to make your way around the obstacles while only being allowed to turn at the encounter of an obstacle. You will have to identify the initial height and all the correct direction changes that are needed in order to reach the end (going backwards is not allowed). Once again, we have U pointing upwards, D for downwards, L for the left and R for the right.

Input specification:

The first line contains an integer *n* that corresponds to the number of entries needing to be transformed. Every one of the *n* entries below contains the width of the labyrinth followed by its height on the first line. On the following lines is the complete labyrinth.

Output specification:

For every entry, you will have to give the starting height **(**the first line of "#" being 0 and counting downwards) into the labyrinth followed by the direction changes required to reach the end. Direction changes are separated by spaces in this case.

Sample input:

```
2
8 9
#### ###

   #   #
       #
  #
#    #
       #
   # #
      #
19 8
############### ##
         ##   #
       ##          #
        #        #
  #           #
#
   #       #
   # #          #
       # ##     #
```

Sample output:

```
7 U R U

2 D R U R U
```

Explanation of the first output:

Starting from the left side at a position of 7 counting downwards, we go initially to the right and then turn upwards as we encounter our first obstacle. On our second encounter we choose to go right. On our last encounter, the end of the labyrinth is straight upwards of where we are and we go for it. “$” are used to describe the path taken in this example:

```
####$###
    $
   #$  #
    $  #
  # $
# $$$#
  $    #
$$$# #
      #
```