THE CRC ROBOTICS PROGRAMMING COMPETITION

PRELIMINARY PROBLEM #2

TAK TIK 2025

A program of

AEST EAST

Version 1.0

A FEW NOTES

- The complete rules are in section 4 of the rulebook.
- You have until **Sunday, December 1st, 11:59 pm** to submit your code.
- Feel free to use the programming forum on the CRC discord to ask questions and discuss the problem with other teams. It is there for that purpose!
- **We are giving you quick and easy-to-use template files for your code and the tests. You are required to use them.**

USING THE TEMPLATE FILE

- The template tests call the function to test, take the output and allow you to quickly check if your code works as intended. **All your code, except additional functions you create, should be written in the function of the part of the problem you are solving.**
- Points given in the document are indications of how difficult the section is and how many points you will get if you complete it. This preliminary problem is going to be 2% of the main challenge towards the global score of the programming competition and for more points related information consult the rulebook.

STRUCTURE

Every problem contains a small introduction like this about the basics of the problem and what is required to solve it.

Input and output specification:

Contains the inputs and their format, and which outputs the code is required to produce and in what format they shall be.

Sample input and output:

Contains a sample input, sometimes containing sub examples in the sample input, and what your program should return as an output.

Explanation of the first output:

Explains briefly the logic that was used to reach the first output, given the first input.

Purr-ogramming Challenges: Cat-tastic Coding for Clever Devs

Welcome to the feline kingdom of coding, where every line of code purrs with creativity! Your mission, should you choose to accept it, is to tackle a series of claw-some programming challenges, each sprinkled with a healthy dose of cat-tastic wordplay.

First up, fix the server's **cat-astrophe**: a buggy script that's generating results as random as a kitten's zoomies. Then, unravel the secrets of the "ClawSort" algorithm, designed to organize data with cat-like agility—just watch out for infinite loops; they're as tricky as chasing laser pointers! Finally, build a **purr-fect app** that translates meows into human speech.

Each challenge will test your wits, but remember: for every bug, there's a purr-sistent solution waiting to be uncovered. So sharpen your claws, flex those coding skills, and prepare to pounce on these feline-inspired problems. May your logic be as sleek as a panther's stride! 🐾

As you might have guessed, the intro was generated with CHAT GPT (chat = cat in french). Since CHAT GPT has been available to us, a lot has changed, so here's a series of challenges about cats!

Part 1: ConCATenation (10 points)

After the popular boy math and girl math, here's the CAT math. It is an alternate math system where the operations are different. We are only gonna show you the addition and the multiplication. You will need to make a program that will solve simple CAT math equations. In CAT math, the addition of 2 numbers is to put them one after the other.

$$x + y = xy$$

And here's a concrete example using values:

$$4 + 3 = 43$$

For the multiplication in CAT math, you multiply 2 numbers by repeating the second number the number of times indicated by the first number.

$$3 * 4 = 444$$
$$2 * 14 = 1414$$

Input and output specification:

As an input you will receive a string variable containing 2 numbers separated by the addition or multiplication operator. The numbers will always be positive.
As an output, you will need to give the result of the equation as an int,

Sample input:

```
4+11
6*30
178+82
12*3
```

Sample output:

```
411
303030303030
17882
333333333333
```

Explanation of the first output:

In the first equation, we are adding 4 to 11. In that case, we only have to conCATenate the 2 values together to obtain 411. For the first multiplication we have 6 * 30. This means that we need to repeat the number 30 6 times which gives the result 303030303030.

Part 2: CHATpeaux (hats) (10 points)

You are opening a top hat business. For your new business, you will have to determine the price of your hats. You are doing custom hats and to simplify the work, you want to make a program that will take a few inputs to give you the final price. You will need to compute the amount of materials needed, consider the cost of the materials and finally include a profit margin!

Every top hat is composed of 3 parts. The first one is the top and is shaped like a disk. The top of the hat will be proportional to client head diameter. Here's the formula to get the amount of material relative to the head diameter.

$$top = \pi * (\frac{d}{2})^2$$

The second section of your top hats is the side. The amount of material will be determined by the head diameter, but this time also by the height of the top hat.

$$side = 2 * \pi * (\frac{d}{2}) * h$$

The third and final part of the hat is the brim. It's width will be decided by the client. The width of the brim will be represented by the letter b. Here's the formula to find the amount of material needed for the brim.

$$brim = \pi * (b + (\frac{d}{2}))^2 - \pi * (\frac{d}{2})^2$$

With all these sections, you now have the total amount of materials that will be needed. The only thing left to do is to multiply the amount of material by the cost and then by your profit margin using the following equation:

$$price = materials * material\,cost * profit\,margin$$

Input and output specification:

For input you will receive:

- d: the width of the opening of the hat
- h: the height of the hat
- b: the width of the brim
- m_cost: the material cost
- margin: the profit margin that you want to make in percentage

Sample input:

d = 16, h = 14, b = 4, m_cost = 0.42, margin = 0.1
d = 18, h = 16, b = 3, m_cost = 0.38, margin = 0.15
d = 17, h = 16, b = 5, m_cost = 0.40, margin = 0.18

Sample output:

48.56
77.36

102.75

Explanation of the first output:

Using the values given to us, we start by finding the amount of material required for the different parts. Here's the amount of material for the top of the hat:

$$top = \pi * (\frac{d}{2})^2$$

$$top = \pi * (\frac{16}{2})^2$$

$$top = 201.06192982974676$$

We can now compute the amount of materials required for the side:

$$side = 2 * \pi * (\frac{d}{2}) * h$$

$$side = 2 * \pi * (\frac{16}{2}) * 14$$

$$side = 703.7167544041137$$

Here's the equations for computing the amount of materials for the brim.

$$brim = \pi * (b + (\frac{d}{2}))^2 - \pi * (\frac{d}{2})^2$$

$$brim = \pi * (4 + (\frac{16}{2}))^2 - \pi * (\frac{16}{2})^2$$

$$brim = 251.32741228718345$$

Once we have all the different sections, we can compute the final cost using the material cost and the profit margin.

$$price = materials * material\,cost * profit\,margin$$

$$price = 1156.1060965210438 * 0.42 * 0.1$$

$$price = 48.55645605388384$$

$$price = 48.56$$

Part 3: CaptCHAT (CatCAPTCHA) (60 points)

The CAPTCHA is a security measure designed to prevent cyberattacks by checking that an Internet user is human and not a bot. A CAPTCHA test could consiste of a random sequence of letters or numbers to identify, or even a math problem to solve. Unfortunately, many researchers have developed algorithms that can pass simple CAPTCHAs, so you decide to create an improved CAPTCHA test called CaptCHAT!

The CaptCHAT involves an ASCII image of a cat that represents two text sequences. You will write a program that transforms two text sequences into such an ASCII image. Internet users will need to decipher the original text from these image representations!

Here are the four default CaptCHAT images, each of which has a **height of 3 rows** and a **width of 5 columns**, separated by hashes:

```
 ^-^  #  ^-^  #  ^-^  #  ^-^
|'_'| # |-_-| # |*_*| # |@_@|
 > <  #  > <  #  > <  #  > <
```

You will modify these images according to the **difference between two text sequences** measured by **Levenshtein distance**.

Levenshtein distance measures the minimum number of operations needed to transform one string (*string1*) into another (*string2*). The **3 allowed operations** are insertion, deletion, and substitution.

- Insertion: insert a character in *string1*.
    - chat → chats: insert the letter ‘s’ in the string ‘chat’.
- Deletion: delete a character in *string1*.
    - chats → cats: delete the letter ‘h’ in “chats”.
- Substitution: replace a character with another in *string1*.
    - cats → hats: replace the letter ‘c’ with ‘h’ in “cats”.

However, note that the Levenshtein distance between “chat” and “hats” **is not** 3. This is because we only need 2 operations to transform “chat” into “hats”: we can, for example, first delete the letter ‘c’ (“chat” → “hat”), then insert the letter ‘s’ at the end (“hat” → “hats”). The **minimum** number of operations needed, i.e. the Levenshtein distance between “chat” and “hats”, is thus 2.

Here is the formal definition of Levenshtein distance $lev(a,\ b)$ between two strings *a* (of length $|a|$) et *b* (of length $|b|$):

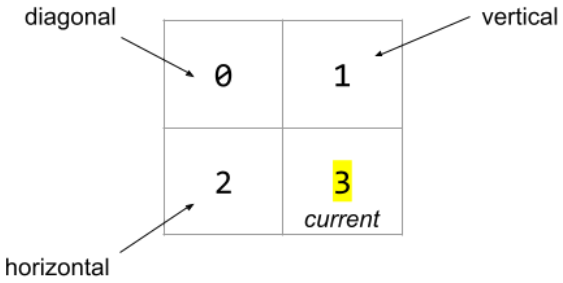
$$\operatorname{lev}(a,b) = \begin{cases} |a| & \text{if } |b| = 0, \\ |b| & \text{if } |a| = 0, \\ \operatorname{lev}\left(\operatorname{tail}(a), \operatorname{tail}(b)\right) & \text{if } \operatorname{head}(a) = \operatorname{head}(b), \\ 1 + \min \begin{cases} \operatorname{lev}\left(\operatorname{tail}(a), b\right) \\ \operatorname{lev}\left(a, \operatorname{tail}(b)\right) \\ \operatorname{lev}\left(\operatorname{tail}(a), \operatorname{tail}(b)\right) \end{cases} & \text{otherwise} \end{cases}$$

We can construct a matrix with the distances between all substrings of *string1* and *string2*. The value in the **bottom-right corner** is the Levenshtein distance between *string1* and *string2*. Here is the matrix for “chat” and “hats”:

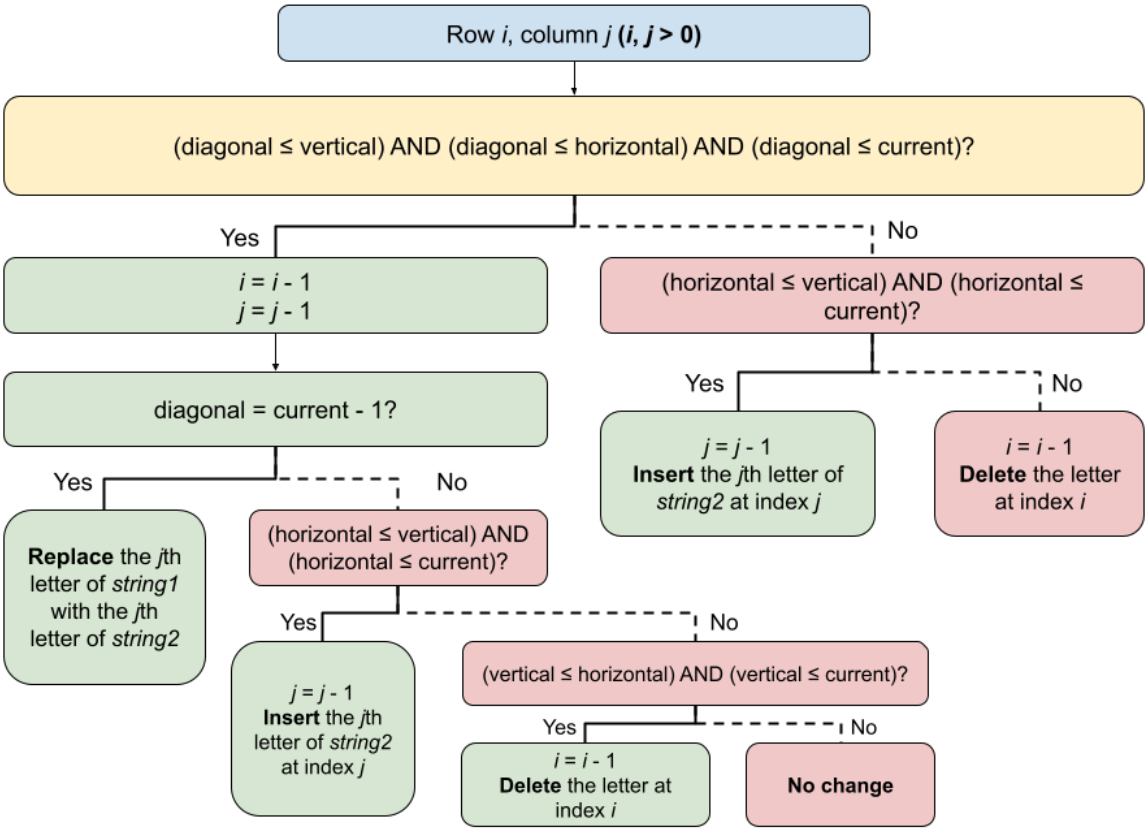
		h	a	t	s
	0	1	2	3	4
c	1	1	2	3	4
h	2	1	2	3	4
a	3	2	1	2	3
t	4	3	2	1	2

Often, there are several optimal ways to transform *string1* into *string2*. To ensure that we always obtain consistent operations, we will use a **backtracing** algorithm.

We start in the bottom-right corner of the distance matrix, which has *m* rows (0, 1, 2, …, $m-1$) and *n* columns (0, 1, 2, …, $n-1$). We’ll define *diagonal, vertical,* and *horizontal* elements with respect to the *current* value:



Here is how we determine the operation that is applied to *string1* for the value at the *i*th row and *j*th column (the *current* value) of the distance matrix:



The algorithm stops once we reach either row 0 or column 0.

After finding the operations used to transform *string1* into *string2*, you need to modify the CaptCHAT image in the same way. You will apply insertions on line 1, deletions on line 2, and substitutions on line 3. Since the default CaptCHAT images have a width of 5 columns, if the operation was applied on an index $ind \geq 5$, use the remainder of $ind \div 5$ instead.

Finally, the Levenshtein distance $dist$ between *string1* and *string2* will determine the eye shape of the CaptCHAT. Depending on the remainder of $dist \div 4$, the eyes will be:

- `' '` if the remainder = 0,
- `- -` if the remainder = 1,
- `* *` if the remainder = 2,
- `@ @` if the remainder = 3.

Input and output specification:

As input, you will receive an *array* with two variables of type *string* that you need to convert into an ASCII image. The provided strings will always be lowercase.

As output, you will provide an *array* that represents the ASCII cat image corresponding to the input text.

Sample input:

```
["kitten", "sitting"]

["chatons", "adorables!"]

["pun'ctu/ation", "pon~ctuat'ion"]
```

Sample output:

```
[" g^-^ ",
 "|@_@|",
 "s> <i"]

[" ^-r!^ ",
 "|'_'| ",
 "ble<  "]

[" ^-^' ",
 "|''| ",
 " o ~ "]
```

Explanation of the first output:

We want to transform “kitten” into “sitting”. Here is the distance matrix:

		s	i	t	t	i	n	g
	0	1	2	3	4	5	6	7
k	1	1	2	3	4	5	6	7
i	2	2	1	2	3	4	5	6
t	3	3	2	1	2	3	4	5
t	4	4	3	2	1	2	3	4
e	5	5	4	3	2	2	3	4
n	6	6	5	4	3	3	2	3

Let’s use backtracing. We start at the value in the bottom-right corner, in the 6th row and 7th column (*i* = 6, *j* = 7):

		s	**i**	**t**	**t**	**i**	**n**	**g**	
	0	1	2	3	4	5	6	7	
k	1	1	2	3	4	5	6	7	
i	2	2	1	2	3	4	5	6	
t	3	3	2	1	2	3	4	5	
t	4	4	3	2	1	2	3	4	
e	5	5	4	3	2	2	3	4	
							<--		Insert 'g' at index 6
n	6	6	5	4	3	3	2	3	

We need to compare the *diagonal, vertical*, and *horizontal* elements using the backtracing algorithm. Here, we have *diagonal* = 3, *vertical* = 4, and *horizontal* = 2. We see that (*diagonal* ≤ *vertical*), but (*diagonal* > *horizontal*). However, (*horizontal* ≤ *vertical*) and (*horizontal* ≤ *current*), so we subtract 1 from *j* (*j* now has a value of 6): the applied operation is therefore the insertion of 'g' at index 6.

The algorithm continues until we reach index 0 or either *string1* or *string2* (i.e. the 0th row or 0th column). Here are all the operations found through backtracing:

```
   0  1  2  3  4  5  6  7
    <--                    Replace with 's' at index 0
k  1  1  2  3  4  5  6  7
       \                   No change at index 1
i  2  2  1  2  3  4  5  6
          \                No change at index 2
t  3  3  2  1  2  3  4  5
             \             No change at index 3
t  4  4  3  2  1  2  3  4
                <--        Replace with 'i' at index 4
e  5  5  4  3  2  2  3  4  No change at index 5
                   \  <--  Insert 'g' at index 6
n  6  6  5  4  3  3  2  3
```

The applied operations are: substitution by 's' at index 0, substitution by 'i' at index 4, and insertion of 'g' at index 6. Let's generate the CaptCHAT image!

The Levenshtein distance is 3, and the remainder of $dist \div 4 = 3 \div 4$ is 3, so the eyes of the CaptCHAT are `@ @`. Our CaptCHAT image without any modifications is the following:

```
[" ^-^ ",
 "|@_@|",
 " > < "]
```

We apply insertions to line 1. We need to insert the letter 'g' at index 6, but 6 ≥ 5, so we insert it at index 1 instead since the remainder of $6 \div 5$ gives us 1. Line 1 then becomes: `" g^-^ "`

There are no deletions, so line 2 stays unchanged. For line 3, we replace the space at index 0 with 's' and the space at index 4 with 'i'. This gives us our final image:

```
[" g^-^ ",
 "|@_@|",
 "s> <i"]
```

Part 4: On CHATvire (We're going overboard!) (30 points)

You go on a boat ride in a group. The issue is that you hate the water and want to make sure to not go overboard! You then want to make sure that it is as stable as possible. To make sure the weight is balanced, you will need to find where to position yourself in the boat.

The boat is a meter large so the positions from left to right will be from -0.5 and 0.5. To choose where to place yourself, you will receive the position and the weight of all the passengers of the boat. Taking into account the effect that all the passengers have on the balance of the boat you will need to find where to position yourself. **Your weight will always be 100 in that problem!**

You will need to give the position from -0.5 and 0-5 with a precision of 2 decimals. However, if the balancing point is out of the boat, you will instead need to return: "On CHATvire!" which means we're going overboard!. To help you balance the boat, here's an equation that will need to be true in order to have the boat balance.

$$left\ side\ =\ right\ side$$

$$\sum_{p\ left} effect\ of\ the\ passengers\ of\ the\ left\ side\ = \sum_{p\ à\ droite} effect\ of\ the\ passengers\ of\ the\ right\ side$$

$$\sum_{p\ left} weight(p_l)\ *\ distance\ from\ the\ center(p_l)\ = \sum_{p\ right} weight(p_r)\ *\ distance\ from\ the\ center(p_r)$$

In this equation we express that the effect of the passengers from the left side must be equal to the effect of the passengers on the right side to make sure it is balanced. So you'll need to balance this equation by choosing where to place yourself. If your position to balance the boat is outside the boat, you need to return "On CHATvire!"

Input and output specification:

For input, you will receive:

- weights: an array of the weight of the passengers
- positions: an array of the position of the passengers relative to the center line of the boat

For output, you will need to provide the position you need to be to balance the boat. This number needs to be of type float and have a 2 digits precision. If that number isn't in the boat, you instead need to return "On CHATvire!".

Sample input:

weights = [0.5, -0.2, 0.5, -0.1]
positions = [40, 70, 70, 40]

weights = [0.2, -0.2, 0.2, -0.2]
positions = [40, 70, 70, 40]

weights = [0.5, -0.2, 0.5, -0.3, 0.4]
positions = [40, 70, 70, 40, 110]

Sample output:

0.37
0.0
"On CHATvire!"

Explanation of the first output:

We want to balance the effect of the passengers on both side so we're gonna start by computing the effect on both sides

$$effect\ of\ the\ passengers\ on\ the\ left\ = \sum_{p\ left} effect\ passenger\ on\ the\ left$$

$$effect\ of\ the\ passengers\ on\ the\ left\ = \sum_{p\ left} weight(p_l)\ *\ distance\ from\ the\ center(p_l)$$

$$effect\ of\ the\ passengers\ on\ the\ left\ = 0.2\ *\ 70\ +\ 0.1\ *\ 40$$

$$effect\ of\ the\ passengers\ on\ the\ left\ = 18$$

$$effect\ of\ the\ passengers\ on\ the\ right\ = \sum_{p\ right} effect\ passenger\ on\ the\ right$$

$$effect\ of\ the\ passengers\ on\ the\ right\ = \sum_{p\ right} weight(p_r)\ *\ distance\ from\ the\ center(p_r)$$

$$effect\ of\ the\ passengers\ on\ the\ right\ = 0.5\ *\ 40\ +\ 0.5\ *\ 70$$

$$effect\ of\ the\ passengers\ on\ the\ right\ = 55$$

Now, to balance the effect from the left and the right we need to choose a distance on the left to produce an effect of 18 - 55 so an effect of -37. Since our weight is 100 we get the following equation:

$$100\ *\ distance\ =\ -\ 37$$

$$distance\ =\ -\frac{37}{100}$$

$$distance\ =\ -\ 0.37$$

Part 5: CATapults (30 points)

During an assault operation on a medieval castle, you are in charge of defending your army with a CATapult. We will give you the height of the wall on which the catapult will be placed as well as the horizontal distance to your target. For a given projectile angle(θ) find the speed that you need to yeet (throw) it to hit your target. **The angle will be given in degrees, be careful with the conversion to radians!**

To do so, you have to use dynamics. **To make things simple, air resistance is considered null.** To find the initial speed, you remember your physics class where you learned that this problem can be broken down into two independent axes.

The first axis is from left to right and so it is the distance between the fortress and the target. The second axis is from top to bottom and is the one that considers gravity. **For this problem, the gravity will be considered to be 9.81.**

The initial speed can be found by using those 2 formulas for the different axis.

$$0 = h + v_i * sin(\theta) * t - g * t^2$$

$$d = v_i * cos(\theta) * t$$

Using those formulas, we can merge them and rearrange them by isolating the initial speed to obtain the following equation.

$$v_i = \sqrt{\frac{gd^2}{2cos^2(\theta)*(h+d*tan(\theta))}}$$

Clarification: $cos^2(\theta) = (cos(\theta))^2$

If you implement this formula, you'll be able to get the initial speed of the projectile. You will then have to return the speed with a precision of 2 decimals.

Input and output specification:

For input, you will receive:

- h: the height of the fortress
- d: the distance of the target
- angle: the angle in degrees taken from the horizon

As an output, you'll need to output the initial speed with a precision of 2 decimals.

Exemple d'entrée:

h = 160, d = 140, angle = 35
h = 18, d = 100, angle = 40
h = 28, d = 85, angle = 45

Exemple de sortie:

23.56
28.64
25.04

Explication de la première sortie:

To get the initial speed, we need to take the equation and replace the variables to obtain the initial speed of the projectile.

$$v_i = \sqrt{\frac{gd^2}{2cos^2(\theta)*(h+d*tan(\theta))}}$$

$$v_i = \sqrt{\frac{9.81*140^2}{2cos^2(35^o)*(160+140*tan(35^o))}}$$

$$v_i = 23.56398382$$

$$v_i = 23.56$$