Sheet 3

1. Given a number determine whether it is valid per the **Luhn formula**, creating the function `is_it_luhn`

   The Luhn algorithm is a simple checksum formula used to validate a variety of identification numbers, such as credit card numbers. Check if a given string is valid

   Strings of length 1 or less are not valid. Spaces are allowed in the input, but they should be stripped before checking. All other non-digit characters are disallowed.

   Ex 1: valid card number

   4539 3195 0343 6467

   The first step is to double every second digit, starting from the right. We will be doubling

   43 39 04 66

   If doubling the number results in a number greater than 9 then subtract 9 from the product. The results of our doubling:

   8569 6195 0383 3437

   Then sum all the digits is 80. If the sum is divisible by 10, then the number is valid.
2. For the following examples, decide which of the composite data structures is better (enum or structs). Then implement them
   - you are Rick, a car shop owner, and you have to choose the fuel of your car between Diesel, Gasoline, LPG, Methane and Electric
   - you have to program the recognition of the IP version of a router. Remember that IPv4 is formatted with 4 group of 3 integer values (from 0 to 255), IPv6 is instead formatted with 8 groups of 4 **hexadecimal** (so no strings!) values.
   - you have to track points in a 3-dimensional space, with the f64 values for each dimension
3. In Trento there is an automated car park with a camera that recognises the number plate of the car. Your task is to associate the number plate with the owner of the car in order to track the price for each car owner. Create a main with an appropriate data structure already initialised with some data. Create a function `recognise_owner` that, given the data structures mentioned above and the number of car plate, returns an `Option`al value of the owner of the car
4. Create a vending machine.
   1. Define an `enum Item` that lists the available items inside the machine (e.g. Coke, Coffee, ecc).
   2. Define the `enum Coin` that contains the coin type accepted by the machine (e.g. `Coin::FiftyCents`, `Coin::Euro2`). And implement the method `to_cents` that convert a `Coin` variant into a `u32` representing the number of cents (e.g. `1€ => 100`, `20¢ => 20`).
   3. Define the `struct VendingMachine` that has the following fields:

      - `coins: u32`
        This field represents the number of cents currently held inside the machine (e.g. if the user inserted 1€ and 10¢, then the `coins` should be at `110`).
      - `items: HashMap<Item, usize>`
        This field should associate an `Item` type to the number of available items to buy.

      Now implement the following methods for `VendingMachine`:

      - `new` method that takes an `HashMap` of `Items` contained in the `VendingMachine` initially and returns a new instance of the `struct`, set `coins` to 0.
      - `add_item` takes an `Item` variant and a `usize`; increments the number of the specified type of items contained by the machine
      - `insert_coin` takes a `Coin` variant and increment the field `coins` by the right value. Returns a `Result<&str, &str>` with the confirmation/error message.
      - `get_item_price` takes an `&Item` variant and returns a `u32` item price.
      - `buy` takes a `Item` variant and returns a `Result<u32, &str>` if you have enough money it returns the change, if you don't, it returns the error as a `&str`.

      Note:
      For using `Item` as an `HashMap` key it needs to implement `PartialEq`, `Eq` and `Hash`. Keep in mind that you can derive them.
5. Implement two `tuple structs` named `Date` and `Hour`. The former takes `u8`, `u8` and `u16` and the latter two `u8`
   Implement a `BoxShipping` struct, with the fields `name: String`, `barcode: String`, shipment_date: Date `and` shipment_hour: Hour`
   Make BoxShipping displayable both with {:?} as well as with {} argument in the println! macro.
   Note:
   *Date and Hour structs should be formatted correctly, ex. 12/01/2001 and 09:00
6. How was that book called? Programming crust? Nevermind.
   Create BUP's library system. It should be able to store books, articles and magazines.

Each book, article and magazine should have a name, a code and a year of publication.

- Books should also have an author and a publishing company.
- Articles should have an orchid.
- Magazines should have a number and a month.
  Then implement the methods to add a book, an article and a magazine to the library system.
  Finally, implement a method to print the library system via the `{}` argument in the println! macro.

7. Create a module called Point that inside has a struct `Point` with the fields `x: f32`, `y: f32`.
   Create the following methods
   - `new` that initializes the Point
   - `distance` that borrow a Point and returns the distance between the two points

```
Create then another module called line that has a struct `Line` with the fields `start: Point`, `end: Point`, `m: f32` and `q: f32`
```

   - you have to implement the `new` method that takes two points and calculates the slope and the intercept of the line `m` and `q`
   - `contains` that borrow a `p: Point` and returns a `Result<_, String>`. The function should check if the `Line` contains the borrowed point

```
Create a third module called test that has a function `test` that creates a line and a point and tests the `contains` method.
```

8. Create a module called `sentence` that has a struct `Sentence` with a field `words: Vec<String>`.
   Create the following methods for the struct `Sentence`:

   - `new_default` that initializes the field `words` with nothing in it.
   - `new` that takes a `&str`, splits it by whitespaces and inserts every word inside the `words` field.
     Create another module `test` with the function `magic_sentence` that mutually borrows a `HashMap<i32, Sentence>`, a `i: i32` and a `j: i32` and returns a `Result<Sentence, &str>`
     The function checks if the sentences at the two indexes exist and if so, creates a `Sentence` with all the equal words in the same position (same index) present in the `Sentence`s.
     If no words are found or if the indexes are not present in the HashMap, reutrn an `Err(&str)`.

   Ex. the sentence "Hello my name was cool yesterday" and the sentence "Hi my name is cool" should result in the sentence "my name cool".