Exercise 1

```
fn is_it_luhn(s: String) -> bool {
    if s.len() <= 1 {
        false
    } else {
        let mut sum: u32 = 0;

        for (index, mut digit) in
            s.chars()
                .rev()
                .enumerate()
                .map(|(index, value)| match value.to_digit(10)
{
                    Some(d) => (index, d),
                    None => panic!("NaN"),
                })
        {
            if index % 2 != 0 {
                digit *= 2;
                if digit > 9 {
                    digit -= 9
                }
            }
            sum += digit;
        }

        sum % 10 == 0
    }
}

fn is_it_luhn_optimal(s: String) -> bool {
    if s.len() <= 1 {
        false
    } else {
        s.chars()
            .rev()
            .enumerate()
            .map(|(index, value)| match value.to_digit(10) {
```

```
                Some(mut digit) => {
                    if index % 2 != 0 {
                        digit *= 2;
                        if digit > 9 {
                            digit -= 9
                        }
                        digit
                    } else {
                        digit
                    }
                }
                None => panic!("NaN"),
            })
            .sum::<u32>()
            % 10
            == 0
    }
}
```

Exercise 2

```
enum Fuel {
    Diesel,
    Gasoline,
    LPG,
    Methane,
    Electricity,
}

enum Ip {
    Ipv4([u8; 4]),
    Ipv6([u16; 8]),
}

struct PointNamedFields {
    x: f64,
    y: f64,
    z: f64,
}
```

```
// alternative
struct PointUnnamedFields(f64, f64, f64);
```

Exercise 3

```
use std::collections::HashMap;

#[test]
fn test() {
    let mut hash_map: HashMap<String, String> = HashMap::new();
    hash_map.insert("CX196SP".to_string(),
"James".to_string());
    hash_map.insert("SASSARI".to_string(),
"Silvio".to_string());

    let mut parking = Parking::new(hash_map, 3);
    assert_eq!(
        parking.park_car("ZZ121PS".to_string(),
"Mario".to_string(), 10.),
        Ok(10. * 0.25)
    );

    assert_eq!(
        parking.park_car("RT534LL".to_string(),
"Luca".to_string(), 10.),
        Err("No more spots available")
    );

    assert_eq!(parking.exit_parking("NO".to_string()), Err("Car
not found"));
    assert_eq!(parking.exit_parking("ZZ121PS".to_string()),
Ok(()))
}

struct Parking {
    parked_cars: HashMap<String, String>,
    remaining_spots: u32,
}

impl Parking {
    pub fn new(parked_cars: HashMap<String, String>,
```

```rust
max_capacity: u32) -> Self {
        let remaining_spots = max_capacity - parked_cars.len()
as u32;
        Self {
            parked_cars,
            remaining_spots,
        }
    }

    pub fn park_car(
        &mut self,
        car_plate: String,
        owner: String,
        minutes: f32,
    ) -> Result<f32, &str> {
        if self.remaining_spots > 0 {
            self.parked_cars.insert(car_plate, owner);
            self.remaining_spots -= 1;
            Ok(minutes * 0.25)
        } else {
            Err("No more spots available")
        }
    }

    pub fn exit_parking(&mut self, car_plate: String) ->
Result<(), &str> {
        if let Some(_) = self.parked_cars.remove(&car_plate) {
            self.remaining_spots += 1;
            Ok(())
        } else {
            Err("Car not found")
        }

        // match self.parked_cars.remove(&car_plate) {
        //     Some(_) => {
        //         self.remaining_spots += 1;
        //         Ok(())
        //     }
        //     None => Err("Car not found"),
        // }
    }
```

```
    pub fn recognise_owner(s: String, hash: &mut
HashMap<String, String>) -> Option<&String> {
        hash.get(&s)
    }
}
```

Exercise 4

```
pub struct VendingMachine {
    coins: u32,
    items: HashMap<Item, usize>,
}

impl VendingMachine {
    pub fn new(items: HashMap<Item, usize>) -> Self {
        Self { coins: 0, items }
    }

    pub fn add_item(&mut self, item: Item, qty: usize) {
        self.items.insert(item, qty);
    }

    pub fn insert_coin(&mut self, coin: Coin) -> Result<&str,
&str> {
        let result = match coin {
            Coin::Cent10 => Ok("10 Cent inserted"),
            Coin::Cent20 => Ok("20 Cent inserted"),
            Coin::Cent50 => Ok("50 Cent inserted"),
            Coin::Eur1 => Ok("1 Euro inserted"),
            Coin::Eur2 => Ok("2 Euro inserted"),
            _ => Err("Invalid coin inserted"),
        };

        if result.is_ok() {
            self.coins += coin.to_cents();
        }
        result
    }

    pub fn get_item_price(&self, item: &Item) -> u32 {
```

```
        match item {
            Item::Coke => 350,
            Item::Water => 100,
            Item::Tea => 250,
            Item::Sprite => 300,
        }
    }

    pub fn buy(&mut self, item: Item) -> Result<u32, &str> {
        let price = self.get_item_price(&item);

        if self.coins >= price {
            if let Some(available) = self.items.get_mut(&item)
{
                if *available > 0 {
                    let change = self.coins - price;
                    self.coins = 0;
                    *available -= 1;
                    Ok(change)
                } else {
                    Err("Item finished")
                }
            } else {
                Err("Item not available")
            }
        } else {
            Err("Not enough money")
        }
    }
}

pub enum Coin {
    Cent10,
    Cent20,
    Cent50,
    Eur1,
    Eur2,
}

impl Coin {
    pub fn to_cents(&self) -> u32 {
```

```
        match self {
            Coin::Cent10 => 10,
            Coin::Cent20 => 20,
            Coin::Cent50 => 50,
            Coin::Eur1 => 100,
            Coin::Eur2 => 200,
        }
    }
}

#[derive(PartialEq, Eq, Hash, Debug)]
pub enum Item {
    Coke,
    Water,
    Tea,
    Sprite,
}
```

Exercise 5

```
#[derive(Debug)]
struct Date(u8, u8, u16);
#[derive(Debug)]
struct Hour(u8, u8);

#[derive(Debug)]
struct BoxShipping {
    name: String,
    barcode: String,
    shipment_date: Date,
    shipment_hour: Hour,
}

impl fmt::Display for Date {
    fn fmt(&self, f: &mut Formatter<'_>) -> fmt::Result {
        write!(f, "{:02}/{:02}/{:04}", self.0, self.1, self.2)
    }
}

impl fmt::Display for Hour {
    fn fmt(&self, f: &mut Formatter<'_>) -> fmt::Result {
```

```
        write!(f, "{:02}:{:02}", self.0, self.1)
    }
}

impl fmt::Display for BoxShipping {
    fn fmt(&self, f: &mut Formatter<'_>) -> fmt::Result {
        write!(
            f,
            "name: {}, barcode: {}, date of shipment: {}, hour
of shipment: {}",
            self.name, self.barcode, self.shipment_date,
self.shipment_hour
        )
    }
}
```

Exercise 6

```
struct Book {
    name: String,
    code: String,
    year_publication: u16,
    author: String,
    publishing_company: String,
}

impl Display for Book {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) ->
std::fmt::Result {
        write!(
            f,
            "name: {}, code: {}, year: {}, author: {},
publisher: {}",
            self.name, self.code, self.year_publication,
self.author, self.publishing_company
        )
    }
}

struct Article {
```

```
    name: String,
    code: String,
    year_publication: u16,
    orcid: String,
}

impl Display for Article {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) ->
std::fmt::Result {
        write!(
            f,
            "name: {}, code: {}, year: {}, orchid: {}",
            self.name, self.code, self.year_publication,
self.orcid
        )
    }
}

struct Magazine {
    name: String,
    code: String,
    year_publication: u16,
    number: u8,
    month: String,
}

impl Display for Magazine {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) ->
std::fmt::Result {
        write!(
            f,
            "name: {}, code: {}, year: {}, number: {}, month:
{}",
            self.name, self.code, self.year_publication,
self.number, self.month
        )
    }
}

struct LibrarySystem {
    books: Vec<Book>,
```

```
    articles: Vec<Article>,
    magazines: Vec<Magazine>,
}

impl Display for LibrarySystem {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) ->
std::fmt::Result {
        let mut s = String::from("Books: \n");
        for book in &self.books {
            s.push_str(&format!("\t{}\n", book));
        }
        s.push_str("Articles: \n");
        for article in &self.articles {
            s.push_str(&format!("\t{}\n", article));
        }
        s.push_str("Magazines: \n");
        for magazine in &self.magazines {
            s.push_str(&format!("\t{}\n", magazine));
        }
        write!(f, "{}", s)
    }
}

impl LibrarySystem {
    fn new() -> LibrarySystem {
        LibrarySystem {
            books: Vec::new(),
            articles: Vec::new(),
            magazines: Vec::new(),
        }
    }

    fn add_book(&mut self, b: Book) {
        self.books.push(b);
    }

    fn add_article(&mut self, a: Article) {
        self.articles.push(a)
    }
    fn add_magazine(&mut self, m: Magazine) {
        self.magazines.push(m);
```

```
    }
}
```

Exercise 7

```rust
// on file point.rs
pub struct Point {
    pub x: f32,
    pub y: f32,
}

impl Point {
    pub fn new(x: f32, y: f32) -> Self {
        Point { x, y }
    }

    pub fn distance(&self, other: &Point) -> f32 {
        let x = (self.x - other.x).powi(2);
        let y = (self.y - other.y).powi(2);
        (x + y).sqrt()
    }
}

// on file line.rs
use super::point::Point;

pub struct Line {
    start: Point,
    end: Point,
    m: f32,
    q: f32,
}

impl Line {
    pub fn new(start: Point, end: Point) -> Self {
        let m = (end.y - start.y) / (end.x - start.x);
        let q = end.y - start.y - m * (end.x - start.x);
        Line { start, end, m, q }
    }
    pub fn contains(&self, point: &Point) -> Result<(), &str> {
```

```
        let res = self.m * point.x + self.q;
        if point.y == res {
            Ok(())
        } else {
            Err("Not contained")
        }
    }
}
```

Exercise 8

```
// on sentence.rs
use std::fmt::Display;

pub struct Sentence {
    pub words: Vec<String>,
}

impl Display for Sentence {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) ->
std::fmt::Result {
        write!(f, "{}", self.words.join(" "))
    }
}

impl Sentence {
    pub fn new_default() -> Self {
        Sentence { words: vec![] }
    }

    pub fn new(s: &str) -> Self {
        Sentence {
            words: s.split_whitespace().map(|str|
str.to_string()).collect(),
        }
    }

    pub fn add_word(&mut self, word: String) {
        self.words.push(word);
    }
```

```rust
}


// on mod.rs
use self::sentence::Sentence;
use std::collections::HashMap;

pub mod sentence;

fn magic_sentence(map: &HashMap<i32, Sentence>, i: i32, j: i32)
-> Result<Sentence, &str> {
    let si = match map.get(&i) {
        Some(e) => e,
        None => return Err("i not found"),
    };

    let sj = match map.get(&j) {
        Some(e) => e,
        None => return Err("j not found"),
    };

    let mut sentence = Sentence::new_default();

    for (wordi, wordj) in si.words.iter().zip(sj.words.iter())
{
        if wordi == wordj {
            sentence.add_word(wordi.clone());
        }
    }

    Ok(sentence)
}
```