Exercise 1

```rust
const SIZE_SUB_STR: usize = 2;

fn find_equal<'a,'b>(s1: &'a str, s2: &'b str) -> Option<(&'a
str,&'b str)>{

    for s1_start in 0..(s1.len() - SIZE_SUB_STR) {

        let s1_end = s1_start + SIZE_SUB_STR;
        let s1_slice = &s1[s1_start..s1_end];

        for s2_start in 0..(s2.len() - SIZE_SUB_STR){

            let s2_end = s2_start + SIZE_SUB_STR;
            let s2_slice = &s2[s2_start..s2_end];

            if s1_slice == s2_slice{
                return Some((s1_slice,s2_slice))
            }
        }
    }
    None
}

fn random_letter() -> char{
    let mut n = rand::random::<u8>();
    n = n % 26 + 'a' as u8;
    n as char
}

fn random_string(len: usize) -> String{
    let mut s = String::with_capacity(len);
    for _ in 0..len{
        s.push(random_letter());
    }
    s
}
```

```
fn lucky_slice(input_str: &str) -> Option<&str>{
    let second_string = random_string(input_str.len());
    let r = find_equal(input_str, &second_string);
    return Some(r?.0);
}
```

Exercise 2

```
use std::fmt::{Debug, Formatter};

pub struct Person<'a>{
    name: String,
    father: Option<&'a Person<'a>>,
    mother: Option<&'a Person<'a>>
}

impl Debug for Person<'_>{
    fn fmt(&self, f: &mut Formatter<'_>) -> std::fmt::Result {
        write!(f, "{}", self.name)
    }
}


impl<'a> Person<'a>{

    fn new(name: String, father:  Option<&'a Person<'a>>,
mother:  Option<&'a Person<'a>>) -> Self{
        Self{ name, father, mother}
    }

    fn private_find_relatives<'b> (&'b self, generations: u32,
relatives: &mut Vec<&'b Self>){
        relatives.push(self);
        if generations > 0{
            if let Some(mother) = self.mother{
                mother.private_find_relatives(generations-
1,relatives);
            }
            if let Some(father) = self.father{
                father.private_find_relatives(generations-
```

```
1,relatives);
            }
        }
    }

    pub fn find_relatives(&self, generations: u32) ->
Vec<&Self>{
        let mut relatives = Vec::new();
        self.private_find_relatives(generations, &mut
relatives);
        relatives
    }


    fn private_find_roots<'b> (&'b self, roots: &mut Vec<&'b
Self>){
        match (self.father,self.mother) {
            (None,None) => roots.push(self),
            (None,Some(m)) => {
                m.private_find_roots(roots);
                roots.push(self);
            }
            (Some(f),None) => {
                f.private_find_roots(roots);
                roots.push(self)
            }
            (Some(f),Some(m)) => {
                f.private_find_roots(roots);
                m.private_find_roots(roots);
            }
        }
    }

    pub fn find_roots(&self) -> Vec<&Self>{
        let mut roots = Vec::new();
        self.private_find_roots(&mut roots);
        roots
    }
}
```

Exercise 3

```
struct ImportantExcerpt<'a> {
    part: &'a str,
}

impl<'a: 'b, 'b> ImportantExcerpt<'a> { //THIS LINE
    fn announce_and_return_part(&'a self, announcement: &'b
str) -> &'b str {
        println!("Attention please: {}", announcement);
        self.part
    }
}
```

Exercise 4

```
struct DoubleRef<'r,'s: 'r,T> {
    r: &'r T,
    s: &'s T
}
```

Exercise 5

```
use std::collections::LinkedList;

trait Split<'a>{
    type ReturnType;
    fn split(&'a self) -> (Self::ReturnType,Self::ReturnType);
}

impl<'a> Split<'a> for String{
    type ReturnType = &'a str;

    fn split(&'a self) -> (Self::ReturnType,Self::ReturnType) {
        self.as_str().split_at(self.len()/2)
    }
}

impl<'a> Split<'a> for &'a [i32]{
    type ReturnType = &'a [i32];
    fn split(&'a self) -> (Self::ReturnType, Self::ReturnType)
{
```

```
        self.split_at(self.len()/2)
    }
}

impl<'a> Split<'a> for LinkedList<f64>{
    type ReturnType = Self;
    fn split(&'a self) -> (Self::ReturnType, Self::ReturnType)
{
        let mut left = self.clone();
        let right = left.split_off(left.len()/2);
        (left,right)
    }
}
```

Exercise 6

```
use std::fmt::{Display, Formatter};
use std::ops::{Add, Sub};

#[derive(Copy, Clone)]
struct Point{
    pub x: f32,
    pub y: f32
}

#[derive(Copy, Clone)]
struct Circle{
    pub center: Point,
    pub radius: f32
}

#[derive(Copy, Clone)]
struct Rectangle{
    pub top_left: Point,
    pub bottom_right: Point
}

impl Default for Point{
    fn default() -> Self {
        Self{
            x: 0.,
```

```
            y: 0.
        }
    }
}

impl Default for Circle{
    fn default() -> Self {
        Self{
            center: Point::default(),
            radius: 1.
        }
    }
}

impl Default for Rectangle {
    fn default() -> Self {
        Self{
            top_left: Point{x: -1.,y :1.},
            bottom_right: Point{x: 1.,y: -1.}
        }
    }
}

impl Add for Point{
    type Output = Self;
    fn add(self, rhs: Self) -> Self::Output {
        Self{
            x: self.x + rhs.x,
            y: self.y + rhs.y
        }
    }
}

impl Sub for Point{
    type Output = Self;
    fn sub(self, rhs: Self) -> Self::Output {
        Self{
            x: self.x - rhs.x,
            y: self.y - rhs.y
        }
    }
```

```rust
}

#[derive(Copy, Clone)]
pub struct Area{
    pub area: f32
}

impl Display for Area{
    fn fmt(&self, f: &mut Formatter<'_>) -> std::fmt::Result {
        write!(f,"Area is: {} cm²", self.area)
    }
}


impl Default for Area{
    fn default() -> Self {
        Area{area: 0.}
    }
}

trait GetArea{
    fn get_area(&self) -> Area;
}
impl GetArea for Point{
    fn get_area(&self) -> Area {
        Area::default()
    }
}
impl GetArea for Circle{
    fn get_area(&self) -> Area {
        let area =  self.radius.powi(2)*std::f32::consts::PI;
        Area{area}
    }
}

impl GetArea for Rectangle{
    fn get_area(&self) -> Area {
        let p = self.bottom_right-self.top_left;
        let area = (p.y * p.x).abs();
        return Area{
            area
```

```
        };
    }
}

impl Add for Area{
    type Output = Self;
    fn add(self, rhs: Self) -> Self::Output {
        Area{
            area: self.area + rhs.area
        }
    }
}

fn sum_area(items: &[&dyn GetArea]) -> Area{
    let mut sum = Area::default();
    items.iter().for_each(|x| sum = sum + x.get_area());
    sum
}
```

Exercise 7

```
fn skip_prefix<'a, 'b>(telephone_number: &'a str, prefix: &'b
str) -> &'a str {
    if telephone_number.starts_with(prefix) {
        &telephone_number[prefix.len()..]
    } else {
        telephone_number
    }
}
```

Exercise 8

```
use std::fmt;
use std::fmt::{Debug, Display, Formatter};

#[derive(Debug)]
struct Chair<'a>{
    color: &'a str,
    quantity: &'a usize,
}
```

```rust
#[derive(Debug)]
struct Wardrobe<'a>{
    color: &'a str,
    quantity: &'a usize,

}

trait Object{
    fn build(&self) -> &str;
    fn get_quantity(&self) -> String;
}

impl<'a> Object for Chair<'a>{
    fn build(&self) -> &str{
        "Chair has been built"
    }

    fn get_quantity(&self) -> String {
        format!("We have {} chairs",
self.quantity).as_str().to_owned()
    }
}

impl<'a> Object for Wardrobe<'a> {
    fn build(&self) -> &str {
        "Wardrobe has been built"
    }

    fn get_quantity(&self) -> String {
        format!("We have {} wardrobes",
self.quantity).as_str().to_owned()
    }
}

impl<'a> Display for Chair<'a>{
    fn fmt(&self, f: &mut Formatter<'_>) -> fmt::Result {
        if self.quantity <= &0 {
            write!(f, "You have no chair")
        } else {
            if self.quantity == &1 {
```

```
                write!(f, "You have 1 {} chair", self.color)
            } else {
                write!(f, "You have {} {} chairs",
self.quantity, self.color)
            }
        }
    }
}
impl<'a> Display for Wardrobe<'a>{
    fn fmt(&self, f: &mut Formatter<'_>) -> fmt::Result {
        if self.quantity <= &0 {
            write!(f, "You have no wardrobe")
        } else {
            if self.quantity == &1 {
                write!(f, "You have 1 {} wardrobe", self.color)
            } else {
                write!(f, "You have {} {} wardrobes",
self.quantity, self.color)
            }
        }
    }
}
```

Exercise 9

```
use std::collections::HashMap;
use crate::sheet4::es9::Permission::*;
#[derive(PartialEq)]
pub enum Role{
    GUEST,
    USER,
    ADMIN
}

#[derive(PartialEq, Eq, Hash, Copy, Clone)]
enum Permission{
    READ,
    WRITE,
    EXECUTE
}
```

```
struct Actions{
    action: String,
    permission: HashMap<Permission, bool>
}

struct User{
    name: String,
    role: Role,
    actions: Vec<Actions>
}

trait Auth{
    fn check_permission(&self, action: &str, permission_type:
&Permission) -> bool;
    fn can_write(&self, action: &str) -> bool;
    fn can_read(&self, action: &str) -> bool;
    fn can_execute(&self, action: &str) -> bool;
}

impl Default for Actions{
    fn default() -> Self {
        let mut permission = HashMap::new();
        permission.insert(READ, false);
        permission.insert(WRITE, false);
        permission.insert(EXECUTE, false);
        Actions {
            action: "".to_string(),
            permission
        }
    }
}

impl Actions{
    fn new(action: String, read: bool, write: bool, execute:
bool) -> Self{
        let mut s = Self::default();
        s.action = action;
        s.permission.insert(READ, read);
        s.permission.insert(WRITE, write);
        s.permission.insert(EXECUTE, execute);
        s
```

```
    }
}

impl Default for User {
    fn default() -> Self {
        User {
            name: "Guest".to_string(),
            role: Role::GUEST,
            actions: vec![],
        }
    }
}

impl Auth for User{
    fn check_permission(&self, string: &str, permission_type:
&Permission) -> bool {
        self.actions.iter().any(|x| {
            x.action == string &&
x.permission.get(permission_type).cloned().unwrap_or(false)
        })
    }

    fn can_write(&self, string: &str) -> bool {
        self.check_permission(string, &WRITE)
    }

    fn can_read(&self, string: &str) -> bool {
        self.check_permission(string, &READ)
    }

    fn can_execute(&self, string: &str) -> bool {
        self.check_permission(string, &EXECUTE)
    }


}

impl User{
    fn change_role(&mut self, role: Role) -> Result<(), String>
{
        if self.role == Role::ADMIN {
```

```
            self.role = role;
            Ok(())
        } else {
            if self.role == Role::USER{
                match role {
                    Role::ADMIN => {
                        Err("User cannot change to
admin".to_string())
                    },
                    Role::GUEST => {
                        self.role = role;
                        Ok(())
                    },
                    Role::USER => {
                        self.role = role;
                        Ok(())
                    },
                }
            }else{
                match role {
                    Role::GUEST => {
                        self.role = role;
                        Ok(())
                    },
                    _ => {
                        Err("Cannot change role".to_string())
                    }
                }
            }
        }
    }

}



fn sudo_change_permission(user: &mut User, string: String,
permission: Permission, value: bool){
    user.actions.iter_mut().for_each(|x| {
        if x.action == string {
            x.permission.insert(permission, value);
```

```
        }
    });
}
```