```rust
/*ITERATOR SU UN VEC PER MODIFICARE I VALORI ALL'INTERNO */
fn main() {
    let mut v: Vec<f32> = vec![1.0, 2.5, 3.7];
    for i in v.iter_mut() {
        *i = i.powf(2.0);
    }
}

/* AGIRE SU VETTORI E GESTIRE CONFRONTI DI ENUM*/
#[derive(PartialEq, Debug, Clone)]
enum AirplaneCompany {
    Boeing,
    Airbus,
}
struct Airplane {
    company: AirplaneCompany,
    model: String,
}
struct AirFleet {
    fleet: Vec<Airplane>,
}
impl AirFleet {
    fn remove_boeing(&mut self) {
        self.fleet.retain(|a| a.company != Boeing);
        //TOGLIE TUTTI QUELLI CHE ADERISCONO ALLA CONDIZIONE
    }
    fn remove_airplane(&mut self, model: &str) -> Result<(), String> {
        // Ciclo for per trovare l'indice dell'aereo da rimuovere
        for (i, a) in self.fleet.iter().enumerate() {
            if a.model == model {
                // Usa remove per rimuovere l'aereo all'indice trovato
                self.fleet.remove(i);
                return Ok(());
            }
        }
        Err("Airplane not found in fleet".to_string())
    }
    fn add_airplane(&mut self, a: Airplane) {
        self.fleet.push(a);
    }
    fn search_airplane(&mut self, model: &str) -> Result<AirplaneCompany,
        for a in &self.fleet {
```

```
                if a.model == model {
                    return Ok(a.company.clone());
                }
            }
            Err("Not in this fleet".to_string())
        }
    }
    use std::cmp::PartialEq;
    use crate::AirplaneCompany::Boeing;
    pub fn main() {
        let mut fleet = AirFleet {
            fleet: Vec::new(),
        };
        let airplane1 = Airplane {
            company: AirplaneCompany::Airbus,
            model: "A380".to_string(),
        };
        let airplane2 = Airplane {
            company: AirplaneCompany::Boeing,
            model: "747".to_string(),
        };
        let airplane3 = Airplane {
            company: AirplaneCompany::Airbus,
            model: "A320".to_string(),
        };
        fleet.add_airplane(airplane1);
        fleet.add_airplane(airplane2);
        fleet.add_airplane(airplane3);

        println!("{:?}", fleet.search_airplane("A380"));
        println!("{:?}", fleet.search_airplane("747"));
        println!("{:?}", fleet.search_airplane("A320"));
        println!("{:?}", fleet.search_airplane("A330"));
    }

    /*IMPLEMENT CLONE, DEBUG, I DISPLAY e GESTIONI DI LISTE*/
    /*USANDO DEBUG POSSO STAMPARE IN MODO AUTOMATICO SENZA IMPLEMENTARE DISPLA
    #[derive(Clone, Debug)]
    struct Student {
        name: String,
        id: u32,
    }
    impl Student {
        fn new(str: &str, id: u32) -> Self {
            Self { name: str.to_string(), id }
```

```
    }
}
impl fmt::Display for Student {
    fn fmt(&self, f: &mut Formatter<'_>) -> fmt::Result {
        write!(f, "Student {{ name: \"{}\", id: {} }}", self.name, self.id
    }
}
struct University {
    name: String,
    vec: Vec<Student>,
}
impl University {
    fn new(str: &str, vec: &[Student]) -> Self {
        let mut students = Vec::from(vec);
        Self { name: str.to_string(), vec: students }
    }
    fn remove_student(&mut self, id: u32) -> Result<Student, &str> {
        if let Some(index) = self.vec.iter().position(|student| student.id
            //cerco lo studente con id uguale ad id
            Ok(self.vec.remove(index)) //remove ritorna l'elemento
        } else {
            Err("Student not found")
        }
    }
}
impl fmt::Display for University {
    fn fmt(&self, f: &mut Formatter<'_>) -> fmt::Result {
        let mut s = String::new();
        s.push_str(format!("{}\n", self.name).as_str());
        s.push_str(format!("{:?}", self.vec).as_str());
        s.push_str("Students: ");
        write!(f, "{s}")
    }
}

use std::fmt;
use std::fmt::{format, write, Formatter};
use std::ops::Index;

pub fn main() {
    let s1 = Student::new("marco", 1);
    let s2 = Student::new("anto", 2);
    let s3 = Student::new("anna", 3);
    let mut university = University::new("Trento", &vec![s1, s2, s3]);
```

```
    println!("{}", university);

    println!("{}", university.remove_student(1).unwrap().id);
}

/*MODULI*/
use crate::modsum::sum;
mod modsum {
    use crate::{modx, mody};
    pub fn sum(x1: modx::X, x2: mody::X) -> f64 {
        let mut ret = 0.0;
        match x1 {
            modx::X::S(v) => ret += (v as i64) as f64,
            modx::X::C(v) => ret += v.len() as f64,
            _ => {}
        }

        match x2 {
            mody::X::F(a, b) => ret += a * b as f64,
            _ => {}
        }
        ret
    }
}
mod modx {
    pub enum X {
        S(char),
        C(String),
        F(f64, usize),
    }
}
mod mody {
    pub enum X {
        S(char),
        C(String),
        F(f64, usize),
    }
}
pub fn main() {
    println!("{}", sum(modx::X::S(' '), mody::X::F(1.2, 4)));
    println!("{}", sum(modx::X::C("hello".to_owned()), mody::X::F(2.4, 10)
}

/*STRINGHE E COME USARLE*/
fn order(vec: Vec<String>) -> Vec<String> {
```

```rust
    let mut v: Vec<String> = Vec::new();
    let mut cont = 0;
    while cont < vec.len() {
        v.push(cont.to_string() + " - " + vec.get(cont).unwrap());
        cont += 1;
    }
    v
}
pub fn main() {
    let a: Vec<String> = vec!["Ciao".to_string(), "Come".to_string(), "Va"
    let b = order(a);
    println!("{}", b.get(1).unwrap().to_owned());
    println!("{}", b.get(2).unwrap().to_owned());
}

/*BHO FUNZIONI CONCATENATE*/
fn res1(x: i32) -> Result<i32, String> {
    if x % 10 == 0 {
        Ok(10)
    } else {
        Err("error".to_string())
    }
}
fn res2(r: Result<i32, String>) -> Result<i32, String> {
    if r.is_ok() {
        Ok(5)
    } else {
        Err("error".to_string())
    }
}
fn wrapper(x: i32) -> Result<i32, String> {
    let r1 = res1(x);
    let r2 = res2(r1);
    if r2.is_ok() {
        Ok(x)
    } else {
        Err("error".to_string())
    }
}
pub fn main() {
    println!("{:?}", wrapper(10));
    println!("{:?}", wrapper(5));
    println!("{:?}", wrapper(11));
}
```

```rust
/*BHO CALCOLO COSE CON MATEMATICA*/
struct MaybePoint {
    x: Option<i32>,
    y: Option<i32>,
}

impl MaybePoint {
    fn new(x: Option<i32>, y: Option<i32>) -> Self {
        Self { x, y }
    }

    fn is_some(&self) -> bool {
        self.x.is_some() && self.y.is_some()
    }

    fn maybe_len(&self) -> Option<f32> {
        if !self.is_some() {
            None
        } else {
            Some(((self.x?.pow(2) + self.y?.pow(2)) as f32).sqrt())
        }
    }
}
pub fn main() {
    let x = MaybePoint::new(Some(10), Some(20));
    let y = MaybePoint { x: Some(10), y: None };

    println!("{:?}", x.is_some());
    println!("{:?}", y.is_some());
    println!("{:?}", x.maybe_len());
    println!("{:?}", y.maybe_len());
}


struct Size {
    height: f32,
    width: f32,
}

impl Size {
    fn new(height: f32, width: f32) -> Self {
        Self { height, width }
    }
```

```
    fn area(&self) -> f32 {
        self.height * self.width
    }

    fn compare(&self, size: &Size) -> Option<bool> {
        let a1 = self.area();
        let a2 = size.area();

        if a1 == a2 {
            None
        } else if a1 > a2 {
            Some(true)
        } else {
            Some(false)
        }
    }
}

pub fn main() {
    use std::fmt::{Debug, Formatter};

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

    let s = Size::new(5.7, 1.2);

    println!("{:?}", s.area());
    println!("{:?}", s.compare(&Size::new(8.9, 10.)));
    println!("{:?}", s.compare(&Size::new(1.8, 0.1)));
    println!("{:?}", s.compare(&Size::new(5.7, 1.2)));
}

/*HASHMAP*/

use std::collections::HashMap;
use crate::hashmaps::Maps;
use crate::other::string_to_tuple;

mod unumber {
    pub type Unumber = usize;
}
```

```rust
mod hashmaps {
    use std::collections::HashMap;
    use crate::unumber::Unumber;

    pub struct Maps {
        pub map: HashMap<Unumber, String>,
    }
}

mod other {
    use std::collections::HashMap;
    use crate::hashmaps::Maps;
    use crate::unumber::Unumber;

    pub fn string_to_tuple(maps: Maps) -> HashMap<Unumber, (Unumber, Strin
        let mut hm: HashMap<Unumber, (Unumber, String)> = HashMap::new();
        for m in maps.map {
            hm.insert(m.0, (m.1.len(), m.1));
        }
        hm
    }
}


pub fn main() {
    let mut hashmap = HashMap::new();
    hashmap.insert(1, "ciao".to_string());
    hashmap.insert(2, "ciao".to_string());
    hashmap.insert(3, "ciao".to_string());

    let hashmap = Maps {
        map: hashmap,
    };

    let hashmap = string_to_tuple(hashmap);

    println!("{:?}", (hashmap.get(&1).unwrap().0, hashmap.get(&1).unwrap()
}


/*STRINGHE BHO FA QUALCOSA DI FACILE*/
struct NameSurname {
    name: String,
    surname: String,
}
```

```rust
fn replace_surname(mut nm: NameSurname, str: String) -> String {
    let rit = String::from(nm.surname);
    nm.surname = str.to_string();
    rit
}

/*GESTIONE CARATTERI*/
struct X {
    s: Option<String>,
    i: i32,
}

impl X {
    fn new(s: &str, i: i32) -> Self {
        Self { s: Some(String::from(s)), i }
    }

    fn take_str(&mut self) -> Option<String> {
        let ret = self.s.clone();
        self.s = None;
        ret
    }
}

fn prev_char(c: char) -> char {
    (c as u8 - 1) as char
}

fn prev_str(str: &str) -> String {
    let mut string = String::new();
    for c in str.chars() {
        if c.is_alphabetic() && c != 'a' && c != 'A' {
            string.push(prev_char(c))
        } else {
            string.push(c)
        }
    }
    string
}
```