Sheet 2 - solutions

1. Write a function called `modify_odd` that takes a mutable reference to an array slice of integers `slice` and sets all odd numbers to 0. Then write a second function that create a Vec, filled with all numbers from 0 to 100, and pass it to `modify_odd`;

```
fn modify_odd(slice: &mut [u32]) {
    for elem in slice {
        if *elem % 2 == 1 {
            *elem = 0;
        }
    }
}
fn modify_odd_iterators(slice: &mut [u32]) {
    slice.iter_mut().for_each(|x| {
        if *x % 2 == 1 {
            *x = 0
        }
    });
}

fn modify_odd_recursive(slice: &mut [u32]) {
    if slice.len() == 0 {
        return;
    }
    if slice[0] % 2 != 0 {
        slice[0] = 0;
    }

    modify_odd_recursive(&mut slice[1..]);
}
```

2. Write a function `count_character` that takes a string consisting of ASCII characters `string` as input and returns a HashMap. The keys of the HashMap should be the characters in the string, and the values should be an u32 representing how many times each character appears in the string.

```
fn count_character(string: &str) -> HashMap<char, u32> {
    let mut map = HashMap::<char, u32>::new();

    for c in string.chars() {
        if let Some(val) = map.get_mut(&c) {
            *val += 1;
        } else {
            map.insert(c, 1);
        }
    }

    map
}

fn count_character_recursive(string: &str) -> HashMap<char, u32> {
    let mut map = HashMap::<char, u32>::new();

    if let Some(val) = string.chars().nth(0) {
        map.insert(val, 1);
    } else {
        return map;
    }

    let map_2 = count_character_recursive(&string[1..]);

    for (c, n) in map_2 {
        if let Some(val) = map.get_mut(&c) {
            *val += n;
        } else {
            map.insert(c, n);
        }
    }

    map
}
```

3. Write a function named `split_at_value` that takes two arguments: a slice of i32 called `slice` and a single i32 value called `value`. The function should find the first element equal to `value` inside `slice`. It should then split the slice at the corresponding index and return the two resulting slices wrapped in an Option. If `value` is not found in `slice`, the function should return None.

```
fn split_at_value(slice: &[i32], value: i32) -> Option<(&[i32], &[i32])> {
    let mut index = Option::<usize>::None;

    for (i, element) in slice.iter().enumerate() {
        if *element == value {
            index = Option::Some(i);
            break;
        }
    }

    let index = index?;

    // option 1
    let s1 = &slice[..index];
    let s2 = &slice[index..];
    return Option::Some((s1, s2));

    // option 2
    //Option::Some(slice.split_at(index))
}
```

4. Write a function `sub_slice` that takes two `&Vec<i32>` as input. If the second vector is contained inside the first one it print the corresponding slice, otherwise it print `Not found`;

```
pub fn sub_slice(vector: &Vec<i32>, sub_vector: &Vec<i32>) {
    let slices_number = vector.len() - sub_vector.len();
    let slice_size = sub_vector.len();
    let mut found = false;
    for i in 0..=slices_number {
        let slice = vector.split_at(i).1.split_at(slice_size).0;
        println!("{:?}", slice);
        if slice == sub_vector {
            println!("Found");
            //println!("{:?}", slice);
            found = true;
        }
    }
    if !found {
        println!("Not found");
    }
}

pub fn sub_slice_recursive(vector: &[i32], sub_vector: &[i32]){

    // recursive function
    pub fn sub_slice_recursive_inner(vector: &[i32], sub_vector: &[i32]) ->
bool{
        if sub_vector.len() == 0{
            return true;
        }

        if vector.len() < sub_vector.len(){
            return false;
        }

        if vector.split_at(sub_vector.len()).0 == sub_vector{
            return true
        }

        return sub_slice_recursive_inner(vector.split_at(1).1,sub_vector);
    }

    if sub_slice_recursive_inner(vector,sub_vector){
        println!("Found");
    }else{
        println!("Not found");
    }

}
```

5. Write the following functions, for each of the functions think carefully about what is the best way to pass the arguments (&, &mut or passing ownership):

   - Write a function `max` that takes a Vec of i32 and returns the maximum value inside it.
   - Write a function `swap` that swaps the first and last element of a vector of i32.
   - Write a function `is_sorted` that takes a Vec of i32 and returns a boolean indicating whether the vector is sorted in non-decreasing order.
   - Write a function `insert_if_longer` that takes a Vec of String (`vec`) and a String (`string`). This function should insert `string` into `vec` only if the length of `string` is greater than 10.

   Also, when possible, implement these functions recursively, not iteratively.

```
fn max(vec: &Vec<i32>) -> Option<i32> {
    if vec.len() == 0 {
        return Option::None;
    }

    let mut max = vec[0];
    for val in vec.iter() {
        if *val > max {
            max = *val;
        }
    }
    Option::Some(max)
}

// note: in general is always better using &[T] instead of &Vec<T> since ti
makes your code more
// flexible
fn max_2(vec: &[i32]) -> Option<i32> {
    if vec.len() == 0 {
        return Option::None;
    }
    let mut max = vec[0];
    for val in vec.iter() {
        if *val > max {
            max = *val;
        }
    }
    Option::Some(max)
}
```

```
fn max_recursive(vec: &[i32]) -> Option<i32> {
    if vec.len() == 0 {
        return Option::None;
    }
    if vec.len() == 1 {
        return Option::Some(vec[0]);
    }

    let (v1, v2) = vec.split_at(vec.len() / 2);

    let max_1 = max_recursive(v1);
    let max_2 = max_recursive(v2);

    match (max_2, max_1) {
        (Option::None, Option::None) => None,
        (Option::Some(e), Option::None) => Some(e),
        (Option::None, Option::Some(e)) => Some(e),
        (Option::Some(e1), Option::Some(e2)) => Some(i32::max(e1, e2)),
    }
}

// the same is true for mutable references, as long as you don't need to change the dimensions
fn swap(vec: &mut Vec<i32>) {
    swap2(vec.as_mut_slice());
}
fn swap2(vec: &mut [i32]) {
    if vec.len() <= 1 {
        return;
    }

    let last_index = vec.len() - 1;

    // option 1
    let x = vec[0];
    vec[0] = vec[last_index];
    vec[last_index] = x;

    // option 2
    //vec.swap(0,last_index);}

fn is_sorted(vec: &Vec<i32>) -> bool {
    if vec.len() == 0 {
        return true;
    }

    let mut prev = vec[0];

    for i in vec {
        if *i < prev {
            return false;
        }
        prev = *i;
    }
    true
}

fn is_sorted_recursive(vec: &[i32]) -> bool {
    if vec.len() < 2 {
        return true;
    }
    if vec.len() == 2 {
        return vec[0] <= vec[1];
    }

    if vec.len() == 3 {
        return vec[0] <= vec[1] && vec[1] <= vec[2];
    }
    let (v1, v2) = vec.split_at(vec.len() / 2);

    if *v1.last().unwrap() > v2[0] {
        return false;
    }

    is_sorted_recursive(v1) && is_sorted_recursive(v2)
}

fn insert_if_longer(vec: &mut Vec<String>, string: String) {
    if string.len() > 10 {
        vec.push(string);
    }
}

// this option works as well, but this allocate some space on the heap when
the function clone is
// called. the other option si better since it move the String.
fn insert_if_longer2(vec: &mut Vec<String>, string: &String) {
    if string.len() > 10 {
        vec.push(string.clone());
    }
}
```

6. Write a function `build_vector` that takes a `Iter<i32>` and returns the `Vec<&i32>` containing all the elements of the iterator;

```
use std::slice::Iter;

fn build_vector(iterator: Iter<i32>) -> Vec<&i32> {
    let mut vector: Vec<&i32> = vec![];
    for el in iterator {
        vector.push(el)
    }
    vector
}

fn build_vector_collect(iterator: Iter<i32>) -> Vec<&i32> {
    let vector: Vec<&i32> = iterator.collect();
    vector
}
```

7. Write a function `pancake_sort` that takes a `&mut Vec<i32>` and sorts it using the pancake sort algorithm;

```
fn flip(vector: &mut Vec<i32>, k: usize) {
    let (left, _) = vector.split_at_mut(k + 1);
    left.reverse();
}

fn find_max(vector: &[i32]) -> usize {
    let mut index = 0;
    for i in 0..vector.len() {
        if &vector[i] > &vector[index] {
            index = i;
        }
    }
    index
}

pub fn pancake_sort(vector: &mut Vec<i32>) {
    let mut index = vector.len();
    while index > 0 {
        let (first_half, _) = vector.split_at(index);
        index -= 1;
        let max_index = find_max(first_half);
        if index != max_index {
            flip(vector, max_index);
            flip(vector, index);
        }
    }
}

pub fn pancake_sort_recursive(vector: &mut Vec<i32>, len: usize) {
    if len == 0 || len == 1{
        return;
    }
    let n = find_max(&vector[0..len]);

    if n < len {
        flip(vector, n);
        flip(vector, len - 1);
    }


    pancake_sort_recursive(vector, len - 1);
}
```

8. Write a function `merge` that takes two `&[i32]` and returns a `Vec<i32>`. The returned vector should contain the elements of the two passed elements sorted, you can assume that the passed slices are sorted;

```
pub fn merge(left: &[i32], right: &[i32]) -> Vec<i32> {
    let mut result = vec![];
    let mut left_index = 0;
    let mut right_index = 0;
    for _ in 0..left.len() + right.len() {
        if left_index < left.len()
            && right_index < right.len()
            && left[left_index] < right[right_index]
        {
            result.push(left[left_index]);
            left_index += 1;
        } else if right_index < right.len() {
            result.push(right[right_index]);
            right_index += 1;
        } else {
            result.push(left[left_index]);
            left_index += 1;
        }
    }
    result
}
```

9. Create a `Vec` that can contain both an `i32` and a `String`;

```
enum DoubleType {
    T1(i32),
    T2(String),
}

pub fn main() {
    let _double_vector = vec![DoubleType::T1(1),
DoubleType::T2(String::from("Hello"))];
}
```

10. Write these enums to represent a mathematical expression:

    - One enum is called `Operation` and can be: `Add`, `Sub`, `Mul`, `Div`.
    - One enum is called `Expression` an can be:
        - `Number` (contain inside an i32)
        - `Operation` (contain inside a left Expression, a right Expression and an Operation)

    Note: the left and right expression must be wrapped around a Box

    ```
    Box<Expression>.
    ```

    You will see Boxes further into the course, from now you just need to know that you can build a box using

    ```
    let my_box = Box::new(my_expression)
    ```

    and you can get the value inside the box by dereferencing it

    ```
    let value_inside = *my_box
    ```

    Write a function `evaluate_expression` that take as input an Expression, and return a Result with a i32 if the result is evaluated correctly, or a string if an error occurs.

```
use std::fmt::Display;

enum Operation {
    Add,
    Sub,
    Mul,
    Div,
}

enum Expression {
    Operation {
        left: Box<Expression>,
        op: Operation,
        right: Box<Expression>,
    },
    Number(i32),
}

impl Display for Operation {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        match self {
            Operation::Add => write!(f, "+"),
            Operation::Sub => write!(f, "-"),
```

```
            Operation::Mul => write!(f, "*"),
            Operation::Div => write!(f, "/"),
        }
    }
}

// just to have a nice output, not required for the exercise
impl Display for Expression {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        match self {
            Expression::Operation { left, op, right } => {
                write!(f, "({} {} {})", left, op, right)
            }
            Expression::Number(n) => write!(f, "{}", n),
        }
    }
}

fn evaluate_expression(expression: &Expression) -> Result<i32, &str> {
    match expression {
        Expression::Operation { left, op, right } => {
            let val_left = evaluate_expression(left)?;
            let val_right = evaluate_expression(right)?;

            match op {
                Operation::Add => {
                    let r = val_left.checked_add(val_right);
                    match r {
                        Option::None => Result::Err("overflow"),
                        Option::Some(v) => Result::Ok(v),
                    }
                }
                Operation::Sub => {
                    let r = val_left.checked_sub(val_right);
                    match r {
                        Option::None => Result::Err("overflow"),
                        Option::Some(v) => Result::Ok(v),
                    }
                }
                Operation::Mul => {
                    let r = val_left.checked_mul(val_right);
                    match r {
                        Option::None => Result::Err("overflow"),
                        Option::Some(v) => Result::Ok(v),
                    }
                }
                Operation::Div => {
                    let r = val_left.checked_div(val_right);
                    match r {
                        Option::None => Result::Err("division by zero"),
                        Option::Some(v) => Result::Ok(v),
                    }
                }
            }
        }
        Expression::Number(n) => Result::Ok(*n),
    }
}
```