Sheet 1 - solutions

1. Write a function `string_reverse` that takes a `&str` as input and returns a reversed `String`;

```
fn string_reverse_with_rev(s: &str) -> String {
    let mut out = String::new();
    for c in s.chars().rev() {
        out.push(c);
    }
    out
}

fn string_reverse_with_pop_match(s: &str) -> String {
    let mut out = String::new();
    let mut s = s.to_string();
    loop {
        match s.pop() {
            None => break,
            Some(c) => out.push(c),
        }
    }
    out
}

fn string_reverse_with_pop_while_let(s: &str) -> String {
    let mut out = String::new();
    let mut s = s.to_string();
    while let Some(c) = s.pop() {
        out.push(c);
    }
    out
}

fn string_reverse_with_collect(s: &str) -> String {
    s.chars().rev().collect()
}
```

2. Write a function `bigger` that takes two `i32` and return the bigger number (`i32`) without using another function call and additional variables;

```
fn bigger(int1: i32, int2: i32) -> i32 {
    if int1 >= int2 {
        return int1;
    } else {
        return int2;
    }
}
```

3. Write a function `multiply` that takes an `i32`, a `f32` and a `f64` and returns the multiplication of the three of them as a `f64` value;

```
fn multiply(x: i32, y: f32, z: f64) -> f64 {
    x as f64 * y as f64 * z
}
```

4. Write a function `e_equals_mc_squared` that takes as input a `f32` named `m` representing the mass, and that uses a **globally-defined** `C` constant (`const C: f32 = ...`) containing the value of the speed of light in a vacuum (expressed in m/s). The function outputs the energy equivalent of the mass input;

```
const C: f32 = 299_792_458.0;

fn e_equals_mc_squared(m: f32) -> f32 {
    m * C.powi(2)
}
```

5. Given a vector of `i32`, create a function that returns the minimum and the maximum value inside that vector;

```
pub fn max_min(v: Vec<i32>) -> (i32, i32) {
    let mut max = 0;
    let mut min = 0;
    for num in v {
        if num >= max {
            max = num;
        }

        if num <= min {
            min = num;
        }
    }

    (max, min)
}

pub fn max_min_recursive(v: Vec<i32>, i: usize, j: usize, mut max: &mut i32,
mut min: &mut i32) {
    if i == j {
        return;
    }

    if v[i] >= *max {
        *max = v[i];
    }

    if v[i] <= *min {
        *min = v[i];
    }

    max_min_recursive(v, i + 1, j, &mut max, &mut min);


}
```

6. Write a function `lord_farquaad` that takes a `String` and outputs another `String` in which every character 'e' is substituted by the character '💥';

```
fn lord_farquaad_better(ee: String) -> String {
    ee.replace("e", "💥")
}

fn lord_farquaad(ee: String) -> String {
    let mut new_ee = String::new();
    for c in ee.chars() {
        if c == 'e' {
            new_ee.push_str("💥");
        } else {
            new_ee.push(c);
        }
    }
    new_ee
}
```

7. In the main function initialize a `HashMap<String, f32>` called `furniture` that stores the pair `String` as key and `f32` as value, where the `String` is the name of the furniture and the `f32` is its price. Then write a function that borrows the `HashMap`, takes a `furniture: String` as input and returns the corresponding `f32`. If there is no such furniture in the `HashMap`, return `-1.0`;

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

// Either return a reference or clone the value
// applies to all the solutions
fn get_furniture(furniture: &HashMap<String, f32>, name: String) -> &f32 {
    furniture.get(name.as_str()).unwrap_or(&-1.0)
}

fn get_furniture_2(furniture: &HashMap<String, f32>, name: String) -> &f32 {
    match furniture.get(name.as_str()) {
        Some(x) => x,
        None => &-1.0,
```

```rust
    }
}

fn get_furniture_3(furniture: &HashMap<String, f32>, name: String) -> f32 {
    if let Some(x) = furniture.get(name.as_str()) {
        x.clone()
    } else {
        -1.0
    }
}

fn main() {
    let mut furniture: HashMap<String, f32> = HashMap::new();
    furniture.insert("Sofa".to_string(), 1200.);
    furniture.insert("Lamp".to_string(), 149.99);
    furniture.insert("Television".to_string(), 700.50);
    furniture.insert("Table".to_string(), 1499.99);
}
```

8. We want to:
   - Write a function `append` that takes a `String`, appends the word "foobar" to it and returns it;
   - Write a `main` function in which we:
     - Declare a `String` initialized with some text.;
     - Pass the `String` to the function `append`;
     - Print the original `String` and the one returned by `append`;
       (do it in this order!)

```rust
fn append(mut s: String) -> String {
    s.push_str("foobar");
    s
}

fn main() {
    let s1 = "test ".to_string();
    let s2 = append(s1.clone());
    println!("{}, {}", s1, s2);
}
```

9. An Armstrong number is a number that is the sum of its own digits each raised to the power of the number of digits.
   For example:
   - `9` is an Armstrong number, because $9 = 9^1 = 9$
   - `10` is not an Armstrong number, because $10 \neq 1^2 + 0^2 = 1$
   - `153` is an Armstrong number, because: $153 = 1^3 + 5^3 + 3^3 = 1 + 125 + 27 = 153$

- `154` is not an Armstrong number, because: $154 \neq 1^3 + 5^3 + 4^3 = 1 + 125 + 64 = 190$
  Write the function `is_armstrong` that determines whether a number is an Armstrong number;

```rust
pub fn is_armstrong(mut n: i32) -> bool {
    let original = n.clone();
    let mut digits = Vec::new();

    while n != 0 {
        digits.push(n % 10);
        n /= 10;
    }

    let num_digits = digits.len() as u32;
    let mut sum = 0;
    for d in digits {
        sum += d.pow(num_digits);
    }
    original == sum
}
```

10. Write a function that takes a "matrix" (2x2, i32 tuple) as input, transposes and returns it.

```rust
type Matrix = ((i32, i32), (i32, i32));

fn transpose(matrix: Matrix) -> Matrix {
    let mut trans = matrix;
    let tmp = trans.0 .1;
    trans.0 .1 = trans.1 .0;
    trans.1 .0 = tmp;

    trans
}
```