Name:	Roll No:	

Programming Language Concepts

Quiz 2, II Semester, 2024–2025

13 February, 2025

1. Consider the following Rust functions.

(i)
```
fn quiz1a(){
    let s1 = "hello";
    let s2 = "world";
    let a = [s1,s2];
    let head = a[0];
    let b = (head == a[0]);
    println!("{}",b)
}
```

(ii)
```
fn quiz1b(){
    let s1 = String::from("hello");
    let s2 = String::from("world");
    let a = [s1,s2];
    let head = a[0];
    let b = (head == a[0]);
    println!("{}",b)
}
```

(iii)
```
fn quiz1c(){
    let s1 = String::from("hello");
    let s2 = String::from("world");
    let a = [s1,s2];
    let head = &a[0];
    let b = (head == a[0]);
    println!("{}",b)
}
```

(iv)
```
fn quiz1d(){
    let s1 = String::from("hello");
    let s2 = String::from("world");
    let a = [s1,s2];
    let head = &a[0];
    let b = (head == &a[0]);
    println!("{}",b)
}
```

Fill in each entry in the following table with **Yes** or **No**.

	Compiles	*Runs*
`quiz1a`	Yes	Yes
`quiz1b`	No	No
`quiz1c`	No	No
`quiz1d`	Yes	Yes

Explanation for Question 1

quiz1a The elements of array `a` are string constants, so the assignment `let head = a[0]` copies the value and there is no problem.

quiz1b The elements of array `a` are strings allocated on the heap, so the assignment `let head = a[0]` borrows the value, which means `a[0]` is no longer valid.

quiz1c The assignment `let head = &a[0]` creates a reference. The comparison `head == a[0]` fails because `head` is a reference and `a[0]` is a string.

quiz1d The assignment `let head = &a[0]` creates a reference. The comparison `head == &a[0]` succeeds because both sides are references.

... Question 2 on reverse

2. Consider the following Rust functions.

(i)
```
fn quiz2a(){
    let mut s = String::from("PLC 2025");
    let a = &s[..4];
    let b = &s[4..];
    s = String::from("Hello world");
}
```

(ii)
```
fn quiz2b(){
    let mut s = String::from("PLC 2025");
    let a = &s[..4];
    let b = &s[4..];
    s = String::from("Hello world");
    println!("s:{}, a:{}, b:{}",s,a,b);
}
```

(iii)
```
fn quiz2c(){
    let mut s = String::from("PLC 2025");
    let a = &mut s[..4];
    let b = &mut s[4..];
    s = String::from("Hello");
}
```

(iv)
```
fn quiz2d(){
    let mut s = String::from("PLC 2025");
    let a = &mut s[..4];
    let b = &mut s[4..];
    s = String::from("Hello");
    println!("s:{}, a:{}, b:{}",s,a,b)
}
```

Fill in each entry in the following table with **Yes** or **No**.

	Compiles	*Runs*
`quiz2a`	Yes	Yes
`quiz2b`	No	No
`quiz2c`	Yes	Yes
`quiz2d`	No	No

Explanation for Question 1

quiz2a Slices `a` and `b` reference the original string `s`. Reassigning `s` would normally create a problem with these references to the original `s`, but the lifetime of Rust references end with their last use. Since `a` and `b` are not used after they are defined, their lifetime has ended when `s` is reeassigned, and the code compiles.

quiz2b Slices `a` and `b` reference the original string `s`. Reassigning `s` creates a problem with these references to the original `s`. The final `println!` extends the lifetime of the references `a` and `b` beyond the reassignment of `s`, so the code does not compile.

quiz2c Slices `a` and `b` are mutable references to the original string `s`. Rust does not allow more than one mutable reference to be live. Since `a` and `b` are not used after they are defined, the lifetime of `a` ends before `b` is assigned, and the code compiles.

quiz2d Slices `a` and `b` are mutable references to the original string `s`. Rust does not allow more than one mutable reference to be live. The final `println!` extends the lifetime of the references `a` and `b` to the end of the block, so there are multiple live references to the same string `s`, and the code does not compile.
