Homework 4 Answer

> ### Fizz buzz program
>
> 1. Create a project called bootcamp using Cargo
> 2. The main function should print a welcome message.
> 3. Write a 'fizz buzz' function that will be called from your main function.
> 1. The function should have a loop counting up to 301
> 2. If the count is divisible by 3, print "fizz"
> 3. If the count is divisible by 5 print "buzz"
> 4. If the count is divisible by 3 and 5 print "fizz buzz"
> 5. At the end print the number of times "fizz buzz" occurred.

> ### Answer
>
> See Gist

> ### Two Sum Answer
>
> #### Brute Force Approach

```rust
fn two_sum(nums: Vec<i32>, target: i32) -> Vec<i32> {

    for (i, num1) in nums.iter().enumerate() {
        for (j, num2) in nums.iter().skip(i+1).enumerate() {
            if num1 + num2 == target {
                return vec![i as i32, (j + i + 1) as i32];
            }
        }
    }
    vec![]
}
```

This Brute Force Approach iteratively checks each pair of numbers in the array to see if they sum up to the target. This is done by using two nested loops: the outer loop iterates through each element, and the inner loop checks all subsequent elements to find a pair that adds up to the target.

HashMap Approach

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

```

```
fn two_sum(nums: Vec<i32>, target: i32) ->
Vec<i32> {

    let mut hm = HashMap::new();

    for (i, num) in
nums.iter().enumerate() {
        let compliment = target - num;

        if let Some(&index) =
hm.get(&compliment){
            return vec![index as i32, i as
i32];
        }
        hm.insert(*num, i);
    }
    vec![]
}
```

This HashMap Approach is more efficient. It iterates through the array once, using a HashMap to store elements already traversed. For each element, we calculate its complement (target - current element) and check if this complement is already in the HashMap. If it is, we found a pair; if not, we add the current element to the HashMap. This approach

> significantly reduces the time complexity as it avoids the need for nested iteration.