1. Define a trait called Hashable which has a single function which calculates an i32 hash

trait Hashable {

fn hash(&self) -> i32;

}

1. Implement Hashable for Pair<i32> (your hash function can do anything so long as it depends on the value of f32)

struct Pair<T> {

first: T,

second: T,

}

impl Hashable for Pair<i32> {

fn hash(&self) -> i32 {

first + second

}

}

1. Implement Hashable for Pair<T> for any type parameter T of which is itself Hashable

impl<T: Hashable> Hashable for Pair<T> {

fn hash(&self) -> i32 {

first.hash() + second.hash()

}

}

1. Implement a function which counts the number of times an item appears in a vector, where the type of item is a generic (hint: use the Eq trait)

fn count<T: Eq>(list: Vec<T>, search: T) -> i32 {

let mut counter = 0;

for item in list {

if item == search {

counter += 1;

}

}

return counter;

}

1. Create the function prototype for a function called generics\_rock which takes three parameters: a, b, c, and d. A must conform to traits Eq and Hashable, b must conform to the trait Large, c must conform to the trait Small, and d must be the same type as A.

fn generics\_rock<A, B, C>(a: A, b: B, c: C, d: A) where

A: Eq + Hashable,

B: Large,

C: Small;