A  
fn squarelist(v: &Vec<i32>) -> Vec<i32> {

v.iter().map(|&x| x \* x).collect()

}

fn sqsum(v: &Vec<i32>) -> i32 {

v.iter().map(|&x| x \* x).sum()

}

fn compose\_n<'a, T>(funs: &'a [fn(T) -> T]) -> impl Fn(T) -> T + 'a {

move |x: T| funs.iter().rev().fold(x, |acc, &f| f(acc))

}

B  
struct Sqrt {

number: f64,

guess: f64,

}

impl Sqrt {

fn new(number: f64, initial\_guess: f64) -> Self {

Sqrt {

number,

guess: initial\_guess,

}

}

}

impl Iterator for Sqrt {

type Item = f64;

fn next(&mut self) -> Option<Self::Item> {

if self.guess == 0.0 {

return None;

}

let new\_guess = (self.guess + self.number / self.guess) / 2.0;

let result = Some(self.guess);

self.guess = new\_guess;

result

}

}

fn iterate(iter: &mut impl Iterator<Item = f64>) -> f64 {

let mut prev\_guess = if let Some(initial\_guess) = iter.next() {

initial\_guess

} else {

return 0.0; // Return 0.0 if iterator is initially empty

};

for guess in iter {

if (guess - prev\_guess).abs() < std::f64::EPSILON {

return guess;

}

prev\_guess = guess;

}

prev\_guess

}  
  
C  
fn slens(stuff: &Vec<String>) -> HashMap<String, usize> {

stuff.iter().map(|s| (s.clone(), s.len())).collect()

}  
  
D  
impl<K, V> Semimap<K, V>

where

K: Eq + Hash,

{

fn new() -> Self {

Semimap {

pairs: HashMap::new(),

}

}

fn insert\_1(&mut self, key: K) {

self.pairs.insert(key, None);

}

fn insert\_2(&mut self, key: K, value: V) {

self.pairs.insert(key, Some(value));

}

fn pair\_count(&self) -> usize {

self.pairs.iter().filter(|(\_, v)| v.is\_some()).count()

}

fn sing\_count(&self) -> usize {

self.pairs.iter().filter(|(\_, v)| v.is\_none()).count()

}

fn get(&self, key: &K) -> Option<&V> {

self.pairs.get(key)?.as\_ref()

}

fn keys(&self) -> Vec<&K> {

self.pairs.keys().collect()

}

fn values(&self) -> Vec<&V> {

self.pairs.values().filter\_map(|v| v.as\_ref()).collect()

}

fn remove(&mut self, key: &K) {

self.pairs.remove(key);

}

fn contains\_key(&self, key: &K) -> bool {

self.pairs.contains\_key(key)

}

}