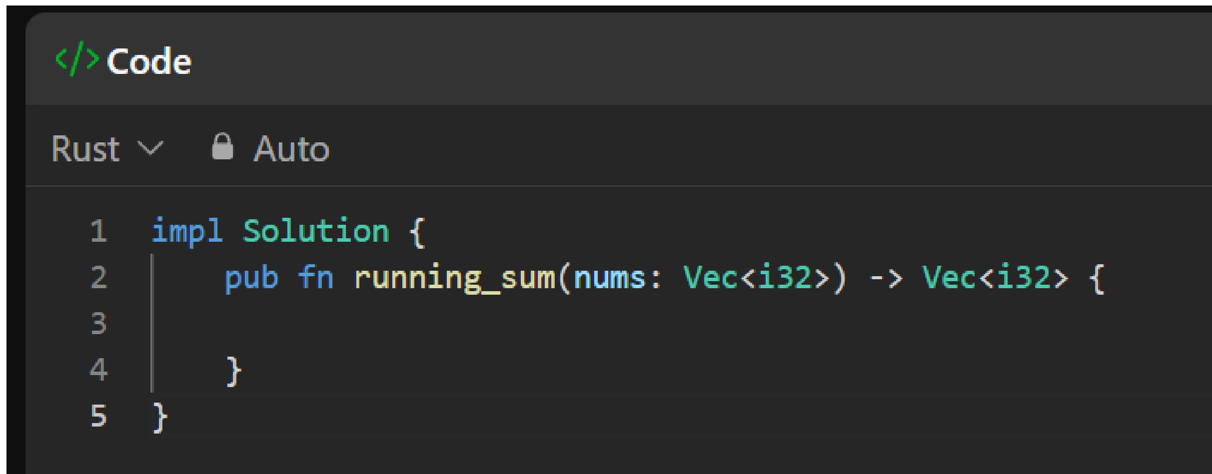
DS 210 Discussion 3

1 LeetCode

On LeetCode, what you need is to finish your codes inside of the given function:



It gives you the input through the parameters (which is `nums: Vec<i32>`, a vector of 32-bit signed integers), and you need to return the results also in a `Vec<i32>`.

- In Rust, you can do that by explicitly calling `return val;`, or leave the val as the last expression of function.

```
fn foo() -> i32 {
  return 3;
}
// Equivalent
fn foo() -> i32 {
  3 // No ;
}
// It can be even like this:
fn foo(x: i32) -> i32 {
  if x > 0 {
    3
  } else {
    5
  }
}
```

To write and run the code locally on your IDE for easier debugging and better linting & hinting, you can have things like:

```
impl Solution {
    pub fn running_sum(nums: Vec<i32>) -> Vec<i32> {
        // Your codes
    }
}
struct Solution; // Make a struct 'Solution'
fn main() {
    println!("{}", Solution::running_sum(vec![3,1,2,10,1]));
}
```

- Explaination: The LeetCode format for rust is a bit strange. the `impl Solution` part is to **implement** some functionality of **struct** *Solution*, which is **running_sum** here. So we define a `struct Solution` and call `Solution::running_sum` in our main function with some test inputs.

2 (1480) Running Sum of 1d Array

2.1 Recall

1. Recall creating an empty *mutable* vector can be done by

```
let mut res: Vec<i32> = Vec::new();

// Or, leave the vector type for rust to auto-inference
let mut res = Vec::new();
res.push(0); // Rust compiler will infer that res is a Vec<i32> by default now.
```

2. A for loop can be done by:

```
// A range 0,1,2,...,N-1 (exclusive N)
for i in 0..N {
  //
}
// Inclusive version. It's up to you which you prefer!
for i in 0..=N-1 {
  //
}
```

You can also iterate through a Vec or Array (and any types with `.iter()` method) by

- *Immutable* borrow "&"

```
for val in vals.iter() {
  // You cannot change vals inside of this loop
  // If that's intended, then use this so you won't change vals by mistake.
}
// Shorter version
for val in &vals { }
```

- *Mutable* borrow "`&mut`"

```
for val in vals.iter_mut() {
  // val is a &mut which can be changed
}
for val in &mut vals { }
```

- Consumes.

```
for val in vals {
  // This is equivalent to vals.into_iter()
}
// You can no longer use 'vals' after this loop!
```

3 (69) Sqrt(x)

3.1 Hints:

1. How will your code handle $x = 0$ and $x = 1$?
2. Is it possible that some of your operations overflows `i32`, in the extreme cases?

[*] Integer types

[*] The unsigned integer types consist of:

Type	Minimum	Maximum
u8	0	2^8-1
u16	0	2^{16}-1
u32	0	2^{32}-1
u64	0	2^{64}-1
u128	0	2^{128}-1

[*] The signed two's complement integer types consist of:

Type	Minimum	Maximum
i8	-(2^7)	2^7-1
i16	-(2^{15})	2^{15}-1
i32	-(2^{31})	2^{31}-1
i64	-(2^{63})	2^{63}-1
i128	-(2^{127})	2^{127}-1

You can use "as" to cast between different types.

```
let x: i32 = 210;
let x_i64: i64 = x as i64;
```