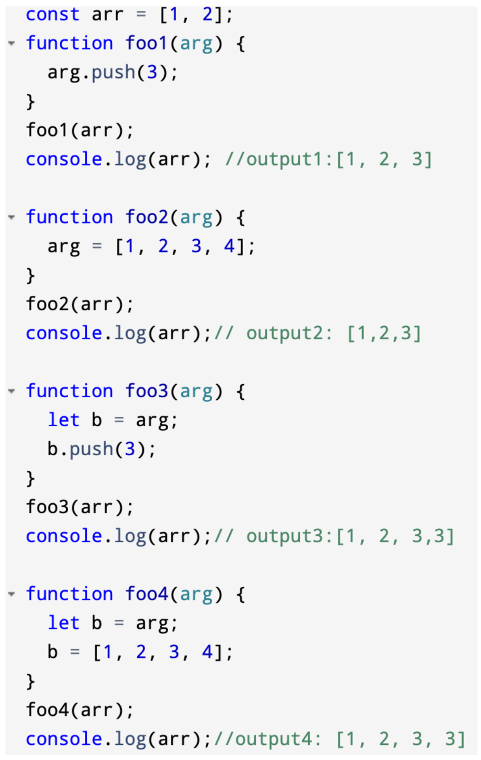
1. Why are closures useful in JavaScript? Give an example use case.
   Closures are powerful and important in JavaScript because they allow a function to remember the variables from the scope where it was created, even after that scope has exited. This means a function can retain access to variables defined in its outer function, allowing you to create private data, preserve state across function calls, and build flexible, reusable logic.
   One of the most common and practical uses of closures is in event handling or delayed logic, where a function is defined in one place but executed later, for example, inside a setTimeout or after a user interaction.
2. When should you choose to use “let” or “const”
   It's best to use const by default for variables that won't be reassigned in JavaScript. It shows clear intent that the value should remain constant. Use let only when the variable’s value needs to change, such as in loops or conditional logic.
3. Give an example of a common mistake related to hoisting and explain how to fix it.
   Not hoisted:
   ```
   console.log(x);
   let x = 5; // ReferenceError
   ```

   Hoisted:
   ```
   Var x;
   console.log(x);
   x = 5; // undefined
   ```

Fix: Always declare the variables at the top of their scope, before we use them. Prefer let or const to avoid unintentional hoisting behavior.

```
let x = 5;
console.log(x);
```

4. What will the outcome of each console.log() be after the function calls? Why?



```
const arr = [1, 2];
function foo1(arg) {
  arg.push(3);
}
foo1(arr);
console.log(arr); //output1:[1, 2, 3]

function foo2(arg) {
  arg = [1, 2, 3, 4];
}
foo2(arr);
console.log(arr);// output2: [1,2,3]

function foo3(arg) {
  let b = arg;
  b.push(3);
}
foo3(arr);
console.log(arr);// output3:[1, 2, 3,3]

function foo4(arg) {
  let b = arg;
  b = [1, 2, 3, 4];
}
foo4(arr);
console.log(arr);//output4: [1, 2, 3, 3]
```

Output 1 is [1, 2, 3] because arrays are passed by reference; arg points to the same array as arr. Using .push(3) mutates the original array

Output 2 is [1, 2, 3] because inside foo2, the line arg = [1, 2, 3, 4], creates a new array, but only changes the local reference arg. It won't affect the original arr.

Output 3 is [1, 2, 3, 3] because b is a reference to the same array as arg and arr. b.push(3) mutates the original array again.

Output 4 is [1, 2, 3, 3] because it's similar logic like foo2, reassigning b to a new array won't affect the local arg. So the original array remains [1,2, 3,3]