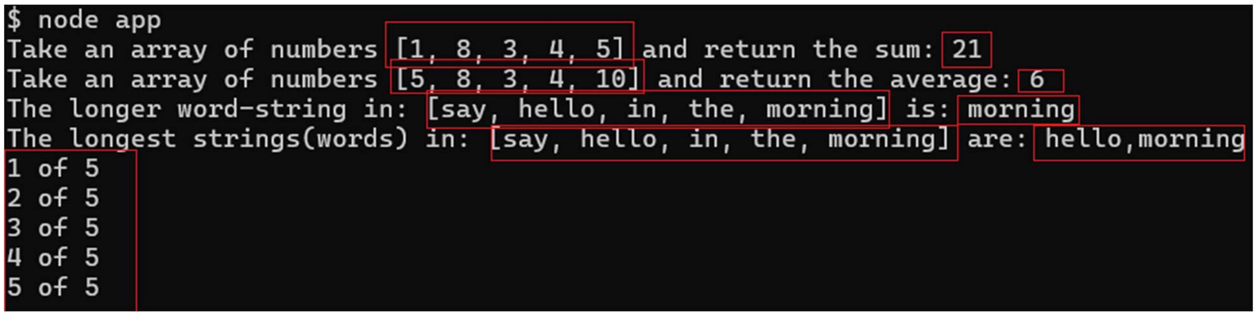
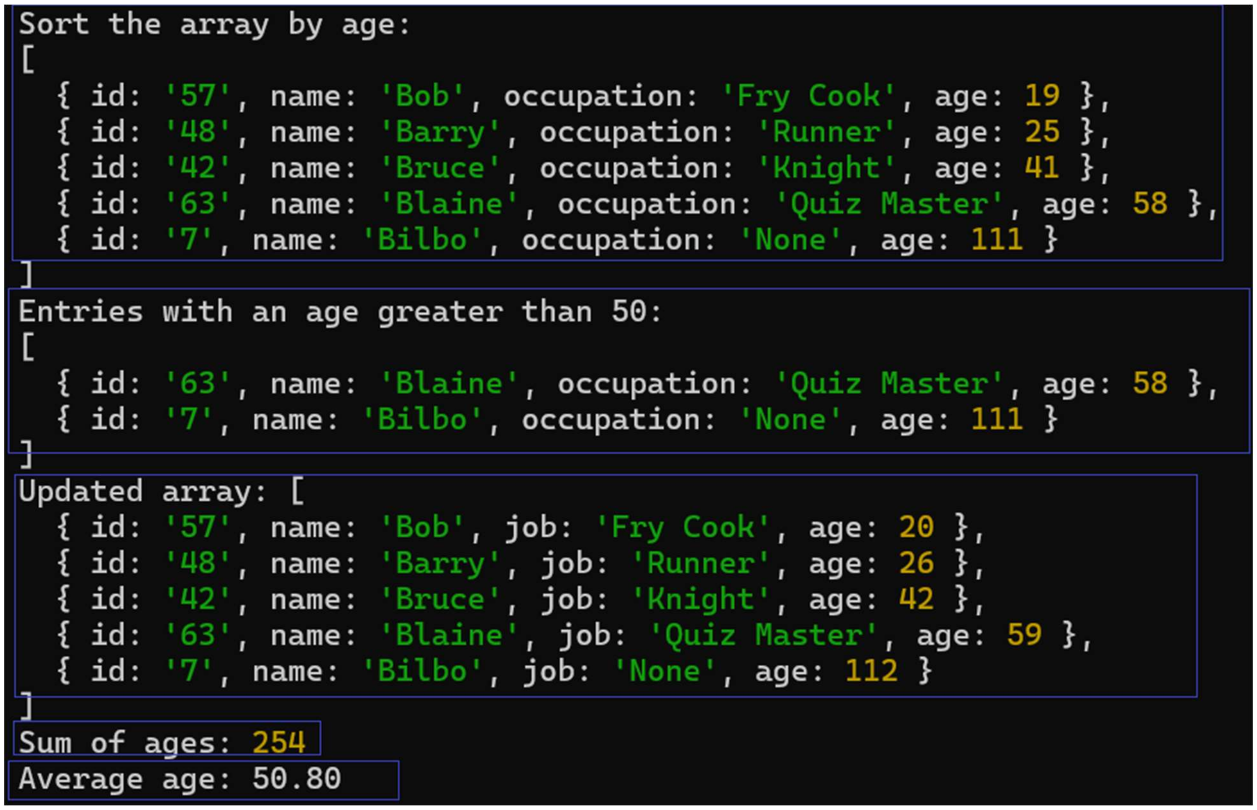
Part 1: Thinking Functionally



```
$ node app
Take an array of numbers [1, 8, 3, 4, 5] and return the sum: 21
Take an array of numbers [5, 8, 3, 4, 10] and return the average: 6
The longer word-string in: [say, hello, in, the, morning] is: morning
The longest strings(words) in: [say, hello, in, the, morning] are: hello,morning
1 of 5
2 of 5
3 of 5
4 of 5
5 of 5
```

Part 2: Thinking Methodically



```
Sort the array by age:
[
  { id: '57', name: 'Bob', occupation: 'Fry Cook', age: 19 },
  { id: '48', name: 'Barry', occupation: 'Runner', age: 25 },
  { id: '42', name: 'Bruce', occupation: 'Knight', age: 41 },
  { id: '63', name: 'Blaine', occupation: 'Quiz Master', age: 58 },
  { id: '7', name: 'Bilbo', occupation: 'None', age: 111 }
]
Entries with an age greater than 50:
[
  { id: '63', name: 'Blaine', occupation: 'Quiz Master', age: 58 },
  { id: '7', name: 'Bilbo', occupation: 'None', age: 111 }
]
Updated array: [
  { id: '57', name: 'Bob', job: 'Fry Cook', age: 20 },
  { id: '48', name: 'Barry', job: 'Runner', age: 26 },
  { id: '42', name: 'Bruce', job: 'Knight', age: 42 },
  { id: '63', name: 'Blaine', job: 'Quiz Master', age: 59 },
  { id: '7', name: 'Bilbo', job: 'None', age: 112 }
]
Sum of ages: 254
Average age: 50.80
```

Part 3: Thinking Critically

1. Increment the age field of an object

Here's a function that takes an object and increments its age field:

```
function incrementAge(obj) {
  if (!obj.hasOwnProperty('age')) {
    obj.age = 0;
  }
  obj.age += 1;
  obj.updated_at = new Date();
}

// Example usage:
let person = { name: 'Alice' };
incrementAge(person);
console.log(person); // { name: 'Alice', age: 1, updated_at: ... }
```

2. Make a copy of an object, increment the age field of the copy, and return the copy

Here's a function that creates a shallow copy of the object, increments the age field in the copy, and updates the updated_at field:

```
function incrementAgeInCopy(obj) {
  // Create a shallow copy of the object
  let copy = { ...obj };

  if (!copy.hasOwnProperty('age')) {
    copy.age = 0;
  }
  copy.age += 1;
  copy.updated_at = new Date();

  return copy;
}

// Example usage:
let person2 = { name: 'Bob' };
let updatedPerson2 = incrementAgeInCopy(person2);
console.log(person2); // { name: 'Bob' }
console.log(updatedPerson2); // { name: 'Bob', age: 1. }
```

Thought Experiment: Modifying the Date object
When working with objects in JavaScript, modifications to nested objects can lead to undesired side effects because the nested objects are still references to the original objects. For example, modifying the Date object in the copy will affect the original object's Date object since they share the same reference.

```
let obj = {
   updated_at: new Date()
};

let copy = { ...obj };

// Modify the Date object in the copy
copy.updated_at.setTime(new Date().getTime() + 10000);

console.log(obj.updated_at); // This will reflect the change made in the copy
```

Avoiding the issue:
To avoid this, we need to make a deep copy of the nested Date object when creating the copy of the original object:

```
function incrementAgeInDeepCopy(obj) {
   // Create a shallow copy of the object
   let copy = { ...obj };

   // Make sure to deep copy the Date object
   if (obj.updated_at instanceof Date) {
      copy.updated_at = new Date(obj.updated_at.getTime());
   }

   if (!copy.hasOwnProperty('age')) {
      copy.age = 0;
   }
   copy.age += 1;
   copy.updated_at = new Date(); // Update with the current time

   return copy;
}

// Example usage:
let person3 = { name: 'Charlie', updated_at: new Date() };
let updatedPerson3 = incrementAgeInDeepCopy(person3);
console.log(person3); // { name: 'Charlie', updated_at: ... }
console.log(updatedPerson3); // { name: 'Charlie', age: 1, updated_at: ... }
```

In this implementation, we ensure that the `Date` object is deep copied, so any modifications to the `updated_at` field in the copied object will not affect the original object.