Lesson 06 Demo 03

Analyzing and Optimizing Existing Code Using Generative AI

Objective: To analyze and enhance the time complexity of the codebase used in the Expressgo parcel delivery system by leveraging GitHub Copilot

Tools required: Visual Studio, GitHub Copilot, and POSTMAN

Prerequisites: refer expressgo code

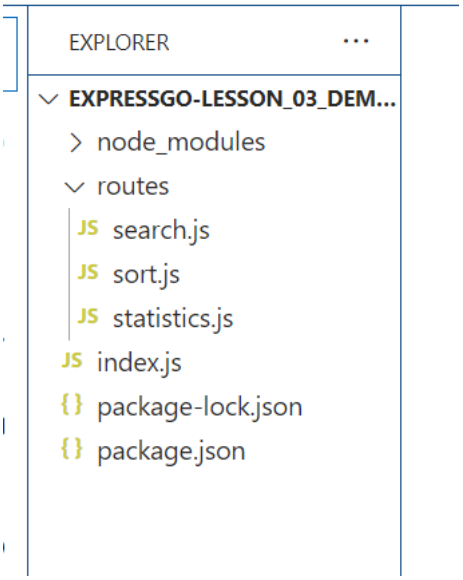
Steps to be followed:

1. Analyze and optimize the existing code using GitHub Copilot
2. Verify the optimized code

Note: Generative AI tool used in this exercise can produce varied outputs even when presented with similar prompts. Thus, you may get different output for the same prompt.

Step 1: Analyze and optimize the existing code using GitHub Copilot

1.1 Download the **expressgo** zip file provided in the LMS and create a folder structure on the local drive, as shown below:



This project is base upon express js module running on port number 3000. Which provide 3 End Point as
http://localhost:3000/sort
This end is use to do sorting by default ascending. If need desending order we need to pass order as desc.
http://localhost:3000/search
This end point is use to search the particular value present in array or not.
http://localhost:3000/statistics
This end point is use to find mean, median and mode (the element occurs more than one time)

1.2 Navigate to the **expressgo-lesson_06_demo_03** folder and execute the command as **npm install**. Which is use to installed all required dependencies.

1.3 Then open this folder in VS code and run the index.js file using below command as

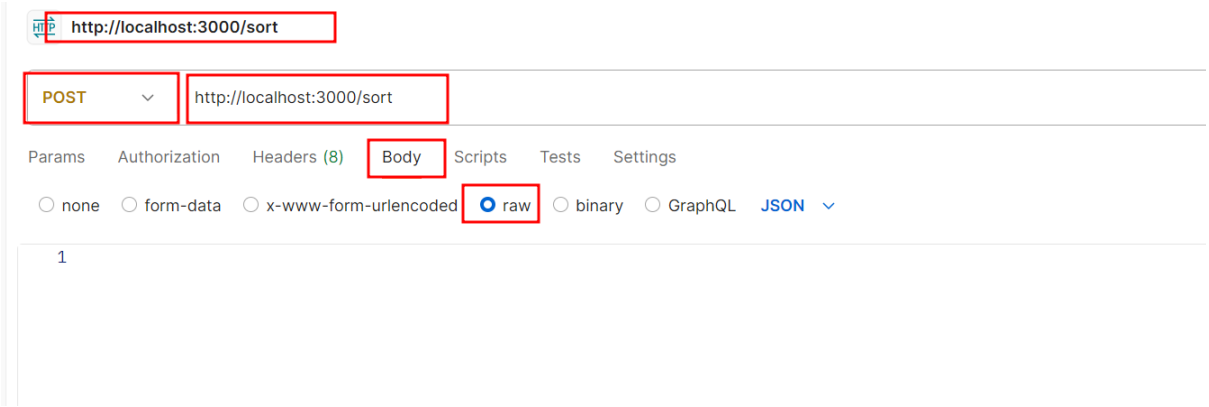
node index.js

```
03>node index.js
```

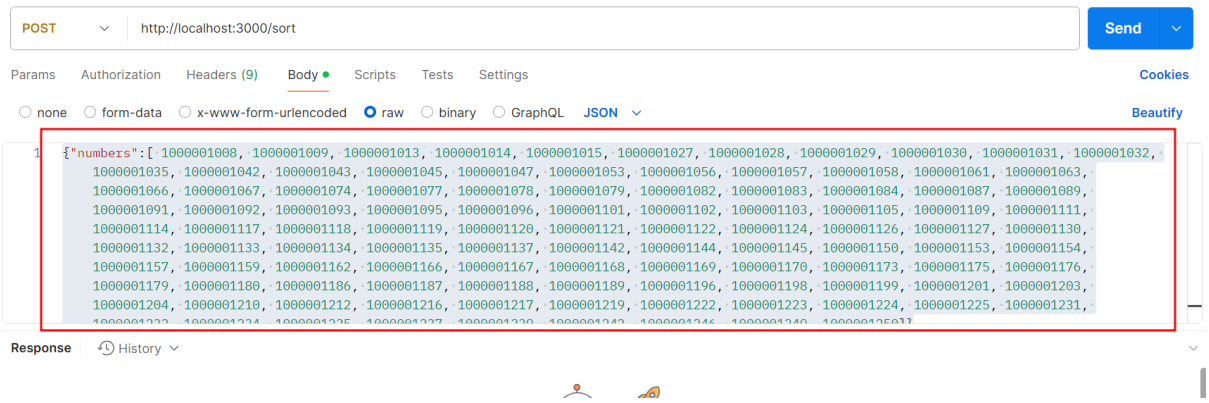
```
Server running on http://localhost:3000
```

Note: Refer to Lesson 05 Demo 02, on how to utilize the Postman application for GET and POST requests to send and receive data. Ensure that the Python code in Visual Studio is running while you perform GET and POST methods using Postman.

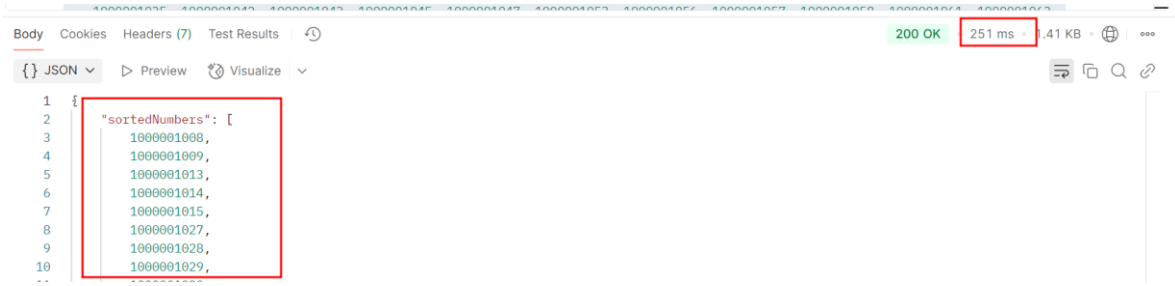
1.4 Open the POSTMAN application, select the **POST method** in drop down option, http://127.0.0.1:5000/sort. Then select body part option and sub option as raw. In Text area you need to pass the data.



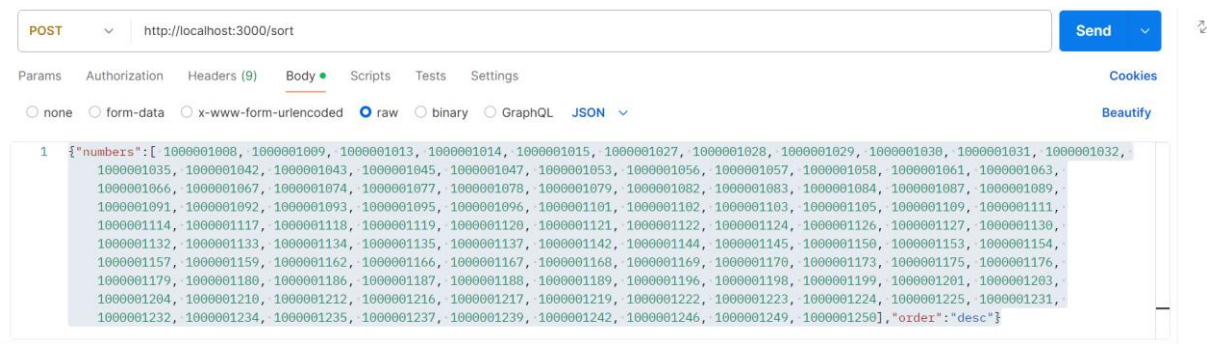
{"numbers":[1000001008, 1000001009, 1000001013, 1000001014, 1000001015, 1000001027, 1000001028, 1000001029, 1000001030, 1000001031, 1000001032, 1000001035, 1000001042, 1000001043, 1000001045, 1000001047, 1000001053, 1000001056, 1000001057, 1000001058, 1000001061, 1000001063, 1000001066, 1000001067, 1000001074, 1000001077, 1000001078, 1000001079, 1000001082, 1000001083, 1000001084, 1000001087, 1000001089, 1000001091, 1000001092, 1000001093, 1000001095, 1000001096, 1000001101, 1000001102, 1000001103, 1000001105, 1000001109, 1000001111, 1000001114, 1000001117, 1000001118, 1000001119, 1000001120, 1000001121, 1000001122, 1000001124, 1000001126, 1000001127, 1000001130, 1000001132, 1000001133, 1000001134, 1000001135, 1000001137, 1000001142, 1000001144, 1000001145, 1000001150, 1000001153, 1000001154, 1000001157, 1000001159, 1000001162, 1000001166, 1000001167, 1000001168, 1000001169, 1000001170, 1000001173, 1000001175, 1000001176, 1000001179, 1000001180, 1000001186, 1000001187, 1000001188, 1000001189, 1000001196, 1000001198, 1000001199, 1000001201, 1000001203, 1000001204, 1000001210, 1000001212, 1000001216, 1000001217, 1000001219, 1000001222, 1000001223, 1000001224, 1000001225, 1000001231, 1000001232, 1000001234, 1000001235, 1000001237, 1000001239, 1000001242, 1000001246, 1000001249, 1000001250]}



Now click on send buton.



By default asending order. If you need descending order then.
Sample data
{"numbers":[1000001008, 1000001009, 1000001013, 1000001014, 1000001015, 1000001027, 1000001028, 1000001029, 1000001030, 1000001031, 1000001032, 1000001035, 1000001042, 1000001043, 1000001045, 1000001047, 1000001053, 1000001056, 1000001057, 1000001058, 1000001061, 1000001063, 1000001066, 1000001067, 1000001074, 1000001077, 1000001078, 1000001079, 1000001082, 1000001083, 1000001084, 1000001087, 1000001089, 1000001091, 1000001092, 1000001093, 1000001095, 1000001096, 1000001101, 1000001102, 1000001103, 1000001105, 1000001109, 1000001111, 1000001114, 1000001117, 1000001118, 1000001119, 1000001120, 1000001121, 1000001122, 1000001124, 1000001126, 1000001127, 1000001130, 1000001132, 1000001133, 1000001134, 1000001135, 1000001137, 1000001142, 1000001144, 1000001145, 1000001150, 1000001153, 1000001154, 1000001157, 1000001159, 1000001162, 1000001166, 1000001167, 1000001168, 1000001169, 1000001170, 1000001173, 1000001175, 1000001176, 1000001179, 1000001180, 1000001186, 1000001187, 1000001188, 1000001189, 1000001196, 1000001198, 1000001199, 1000001201, 1000001203, 1000001204, 1000001210, 1000001212, 1000001216, 1000001217, 1000001219, 1000001222, 1000001223, 1000001224, 1000001225, 1000001231, 1000001232, 1000001234, 1000001235, 1000001237, 1000001239, 1000001242, 1000001246, 1000001249, 1000001250],"order":"desc"}



Body Cookies Headers (7) Test Results

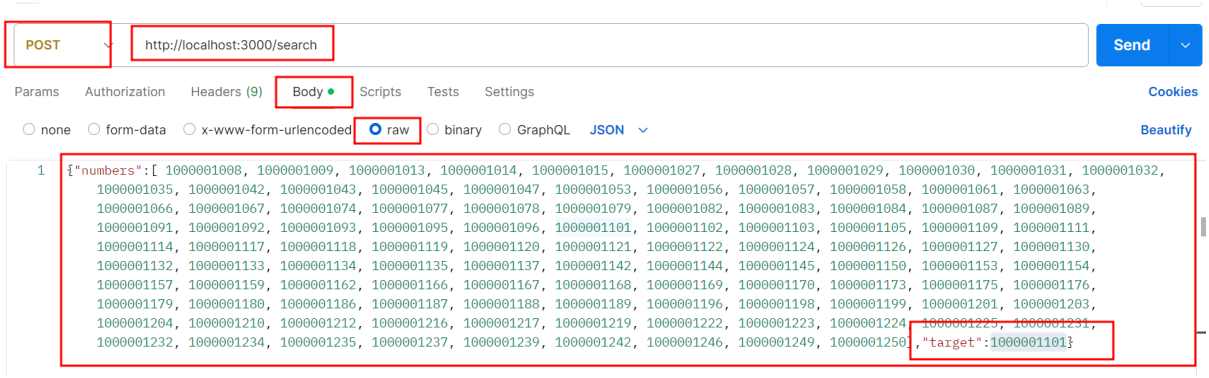
{} JSON Preview Visualize

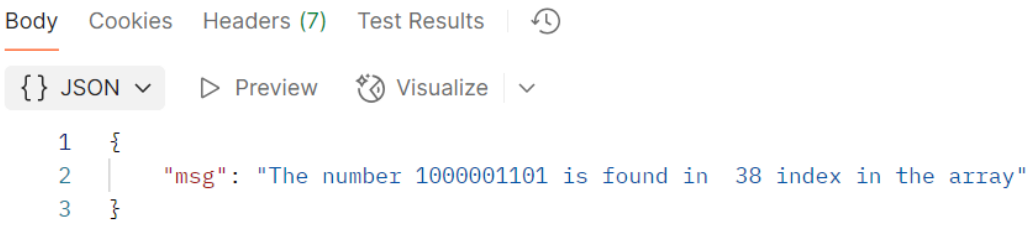

```
{
    "sortedNumbers": [
        1000001250,
        1000001249,
        1000001246,
        1000001242,
        1000001239,
        1000001237,
        1000001235,
        1000001234,
        1000001232,
        1000001231,
        1000001225,
        1000001224
```


1.5 Provide the search API as **target: 1000001101** key value pairs to check element present **target:1234** element is not present.

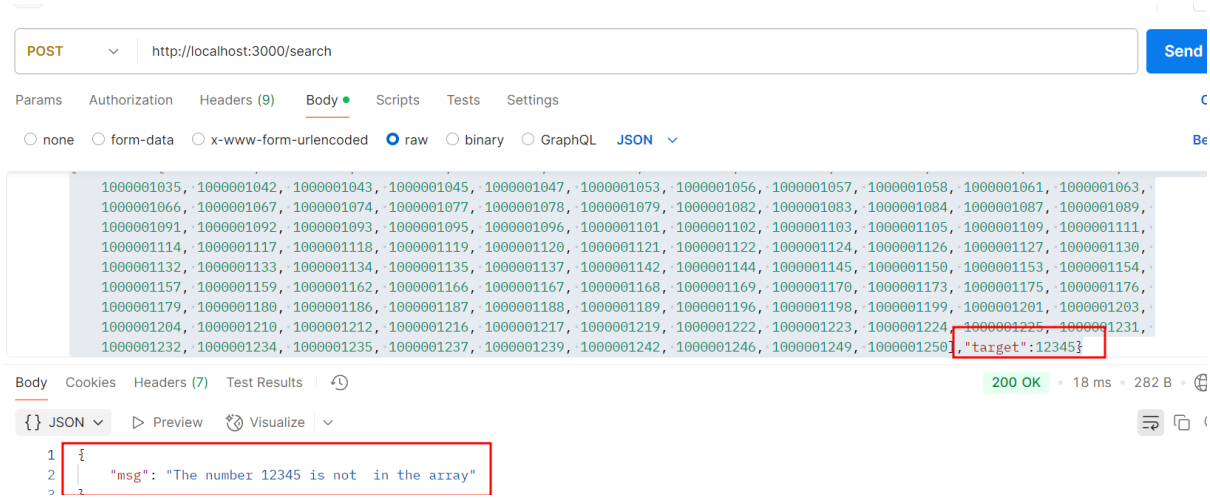
Sample data

{"numbers":[1000001008, 1000001009, 1000001013, 1000001014, 1000001015, 1000001027, 1000001028, 1000001029, 1000001030, 1000001031, 1000001032, 1000001035, 1000001042, 1000001043, 1000001045, 1000001047, 1000001053, 1000001056, 1000001057, 1000001058, 1000001061, 1000001063, 1000001066, 1000001067, 1000001074, 1000001077, 1000001078, 1000001079, 1000001082, 1000001083, 1000001084, 1000001087, 1000001089, 1000001091, 1000001092, 1000001093, 1000001095, 1000001096, 1000001101, 1000001102, 1000001103, 1000001105, 1000001109, 1000001111, 1000001114, 1000001117, 1000001118, 1000001119, 1000001120, 1000001121, 1000001122, 1000001124, 1000001126, 1000001127, 1000001130, 1000001132, 1000001133, 1000001134, 1000001135, 1000001137, 1000001142, 1000001144, 1000001145, 1000001150, 1000001153, 1000001154, 1000001157, 1000001159, 1000001162, 1000001166, 1000001167, 1000001168, 1000001169, 1000001170, 1000001173, 1000001175, 1000001176, 1000001179, 1000001180, 1000001186, 1000001187, 1000001188, 1000001189, 1000001196, 1000001198, 1000001199, 1000001201, 1000001203, 1000001204, 1000001210, 1000001212, 1000001216, 1000001217, 1000001219, 1000001222, 1000001223, 1000001224, 1000001225, 1000001231, 1000001232, 1000001234, 1000001235, 1000001237, 1000001239, 1000001242, 1000001246, 1000001249, 1000001250],"target":1000001101}

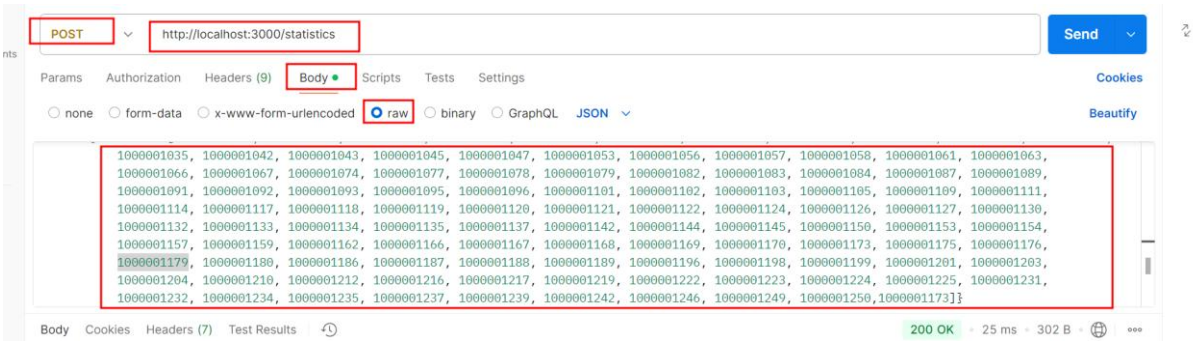




{"numbers":[1000001008, 1000001009, 1000001013, 1000001014, 1000001015, 1000001027, 1000001028, 1000001029, 1000001030, 1000001031, 1000001032, 1000001035, 1000001042, 1000001043, 1000001045, 1000001047, 1000001053, 1000001056, 1000001057, 1000001058, 1000001061, 1000001063, 1000001066, 1000001067, 1000001074, 1000001077, 1000001078, 1000001079, 1000001082, 1000001083, 1000001084, 1000001087, 1000001089, 1000001091, 1000001092, 1000001093, 1000001095, 1000001096, 1000001101, 1000001102, 1000001103, 1000001105, 1000001109, 1000001111, 1000001114, 1000001117, 1000001118, 1000001119, 1000001120, 1000001121, 1000001122, 1000001124, 1000001126, 1000001127, 1000001130, 1000001132, 1000001133, 1000001134, 1000001135, 1000001137, 1000001142, 1000001144, 1000001145, 1000001150, 1000001153, 1000001154, 1000001157, 1000001159, 1000001162, 1000001166, 1000001167, 1000001168, 1000001169, 1000001170, 1000001173, 1000001175, 1000001176, 1000001179, 1000001180, 1000001186, 1000001187, 1000001188, 1000001189, 1000001196, 1000001198, 1000001199, 1000001201, 1000001203, 1000001204, 1000001210, 1000001212, 1000001216, 1000001217, 1000001219, 1000001222, 1000001223, 1000001224, 1000001225, 1000001231, 1000001232, 1000001234, 1000001235, 1000001237, 1000001239, 1000001242, 1000001246, 1000001249, 1000001250],"target":12345}



1.6 Open the **POSTMAN** application, select **POST**, and provide the following link to calculate statistics such as mean, median, and mode of the provided list of numbers: http://localhost:3000/statistics

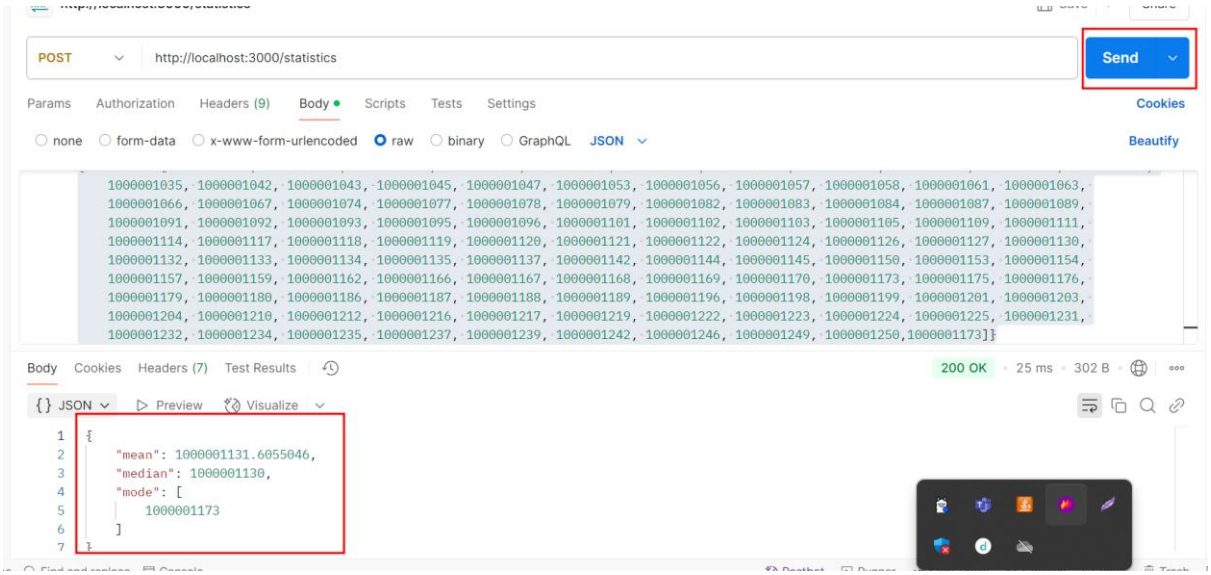


1.7 Provide the following data under **raw** with JSON option

{"numbers":[1000001008, 1000001009, 1000001013, 1000001014, 1000001015, 1000001027, 1000001028, 1000001029, 1000001030, 1000001031, 1000001032, 1000001035, 1000001042, 1000001043, 1000001045, 1000001047, 1000001053, 1000001056, 1000001057, 1000001058, 1000001061, 1000001063, 1000001066, 1000001067, 1000001074, 1000001077, 1000001078, 1000001079, 1000001082, 1000001083, 1000001084, 1000001087, 1000001089, 1000001091, 1000001092, 1000001093, 1000001095, 1000001096, 1000001101, 1000001102, 1000001103, 1000001105, 1000001109, 1000001111, 1000001114, 1000001117, 1000001118, 1000001119, 1000001120, 1000001121, 1000001122, 1000001124, 1000001126, 1000001127, 1000001130, 1000001132, 1000001133, 1000001134, 1000001135, 1000001137, 1000001142, 1000001144, 1000001145, 1000001150, 1000001153, 1000001154, 1000001157, 1000001159, 1000001162, 1000001166, 1000001167, 1000001168, 1000001169, 1000001170, 1000001173, 1000001175, 1000001176,

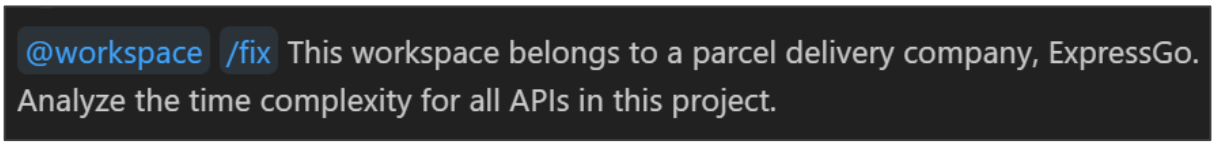
1000001179, 1000001180, 1000001186, 1000001187, 1000001188, 1000001189, 1000001196, 1000001198, 1000001199, 1000001201, 1000001203, 1000001204, 1000001210, 1000001212, 1000001216, 1000001217, 1000001219, 1000001222, 1000001223, 1000001224, 1000001225, 1000001231, 1000001232, 1000001234, 1000001235, 1000001237, 1000001239, 1000001242, 1000001246, 1000001249, 1000001250,1000001173]}

This step provides the following output:



1.8 Enter the following prompt in GitHub Copilot to analyze the time complexity of this codebase:

This workspace belongs to a parcel delivery company, ExpressGo. Analyze the time complexity for all APIs in this project.



Kaleakash

@workspace /fix This workspace belongs to a parcel delivery company, ExpressGo. Analyze the time complexity for all APIs in this project

index.js:1-20

Workspace

Used 1 reference

To analyze the time complexity for all APIs in the project, I need to review the code for the routes defined in sortRoute, searchRoute, and statsRoute. Please provide the code for the following files:

1. sort.js
2. search.js
3. statistics.js

This will allow me to analyze the time complexity of the operations performed by each API.

The time complexity for the sort.js file code.

Used 1 reference

Let's analyze the time complexity for the sort.js file.

Code Analysis

The main operation in this file is sorting an array of numbers. Here's the relevant part of the code:

```
router.post('/', (req, res) => {
    const { numbers, order = 'asc' } = req.body;

    if (!Array.isArray(numbers) || numbers.some(num => typeof num !== 'number')) {
        return res.status(400).json({ error: 'Invalid input, expected an array of numbers' });
    }

    let sortedNumbers = [...numbers].sort((a, b) => a - b);

    if (order === 'desc') {
        sortedNumbers.reverse(); // Reverse for descending order
    } else if (order !== 'asc') {
        return res.status(400).json({ error: 'Invalid order, use "asc" or "desc"' });
    }

    res.json({ sortedNumbers });
});
```

The time complexity for the search.js file code.

```
router.post('/', (req, res) => {
    const { numbers, target } = req.body;
    if (!Array.isArray(numbers) || typeof target !== 'number') {
        return res.status(400).json({ error: 'Invalid input' });
    }
    const index = numbers.indexOf(target);
    res.json({"msg":`The number ${target} is ${index !== -1 ? `found in  ${index} index` : 'not '} in th
});

module.exports = router;
```

Time Complexity

1. **Input Validation**:
   - Checking if `numbers` is an array: `Array.isArray(numbers)` - O(1)
   - Checking if `target` is a number: `typeof target !== 'number'` - O(1)
2. **Searching**:
   - `const index = numbers.indexOf(target);`
     - The `indexOf` method iterates through the array to find the target element. In the worst case, it will check each element once, resulting in a time complexity of O(n), where `n` is the length of the array.

Overall Time Complexity

The overall time complexity is dominated by the search operation, which is O(n). Therefore, the time complexity for the `search.js` file is:

O(n)

This is the time complexity for the worst-case scenario, where `n` is the number of elements in the `numbers` array.

The time complexity for the statistics.js file code

Time Complexity

1. **Input Validation**:
   - Checking if `numbers` is an array and non-empty: `Array.isArray(numbers) || numbers.length === 0` - O(1)
2. **Calculating Mean**:
   - Summing the array: `numbers.reduce((acc, num) => acc + num, 0)` - O(n), where `n` is the length of the array.
   - Calculating the mean: `sum / numbers.length` - O(1)
3. **Calculating Median**:
   - Sorting the array: `sortedNumbers = [...numbers].sort((a, b) => a - b)` - O(n log n)
   - Finding the median: `sortedNumbers.length % 2 === 0 ? (sortedNumbers[mid - 1] + sortedNumbers[mid]) / 2 : sortedNumbers[mid]` - O(1)
4. **Calculating Mode**:
   - Creating frequency map: `numbers.forEach(num => { frequency[num] = (frequency[num] || 0) + 1; })` - O(n)
   - Finding the maximum frequency: `for (let num in frequency) { if (frequency[num] === maxFreq) { modes.push(Number(num)); } }` - O(n)

Overall Time Complexity

The overall time complexity is dominated by the sorting operation, which is O(n log n). Therefore, the time complexity for the `statistic` file is:

O(n log n)

This is the time complexity for the worst-case scenario, where `n` is the number of elements in the `numbers` array.

Similar code found with 1 license type - View matches

1.9 Enter the following prompt to optimize the codebase:

Help me improve the time complexity for these APIs

```
    // Calculate Mean
    const sum = numbers.reduce((acc, num) => acc + num, 0);
    const mean = sum / numbers.length;

    // Calculate Median
    const mid = Math.floor(numbers.length / 2);
    const median = numbers.length % 2 === 0
        ? (quickSelect(numbers, 0, numbers.length - 1, mid - 1) + quickSelect(numbers, 0, numbers.length
        : quickSelect(numbers, 0, numbers.length - 1, mid);

    // Calculate Mode
    const mode = calculateMode(numbers);

    res.json({ mean, median, mode });
});

module.exports = router;
```

Time Complexity Analysis

1. **Mean Calculation**: O(n)
2. **Median Calculation**: O(n) using Quickselect
3. **Mode Calculation**: O(n)

Overall Time Complexity

The overall time complexity for the optimized JS statistics.js file is **O(n)**, which is an improvement from the previous **O(n log n)** due to the optimized median calculation.

Similar code found with 2 license types - View matches

1.10 you can ask any suggestion from github copilot to improve the time complexity.

@workspace /fix help me to imrove the time complexity for the search.js code

JS search.js:1-15

Workspace

> Used 1 reference

The current implementation of the search API uses the indexOf method, which has a time complexity of O(n) for an array of length n. To improve the time complexity, you can use a more efficient search algorithm. One option is to use a hash map (object) to achieve an average time complexity of O(1) for search operations.

Here is the improved code:

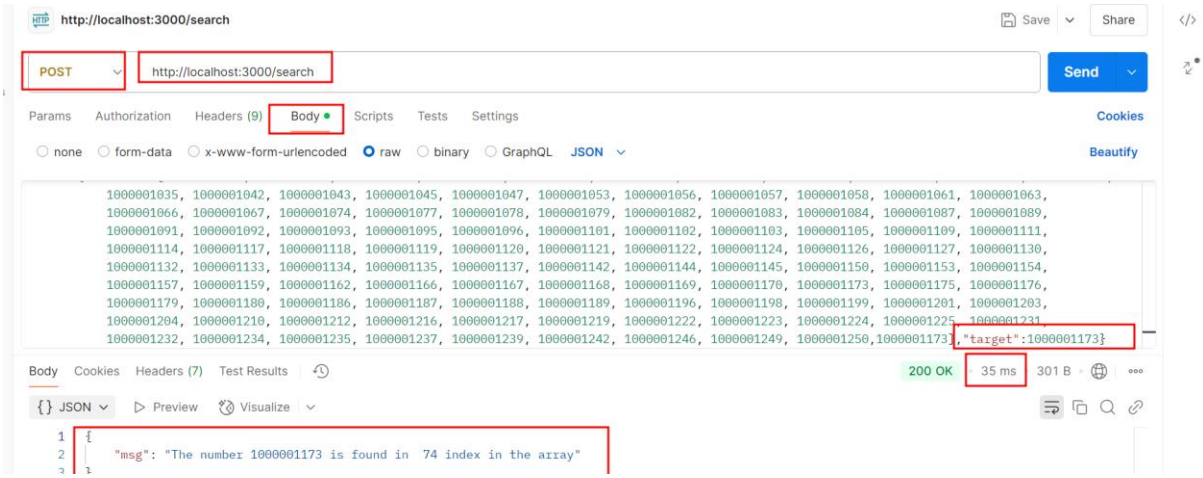
1.11 Check the search.js initial code with time taken to search the elements.

search.js

```
const express = require('express');
const router = express.Router();

router.post('/', (req, res) => {
   const { numbers, target } = req.body;
```

```
    if (!Array.isArray(numbers) || typeof target !== 'number') {
      return res.status(400).json({ error: 'Invalid input' });
    }
    const index = numbers.indexOf(target);
    res.json({"msg":`The number ${target} is ${index !== -1 ? `found in  ${index} index`
  : 'not '} in the array`});
    //res.json({ found: index !== -1, index });
  });

  module.exports = router;
```

1.12 send the request to search the particular element



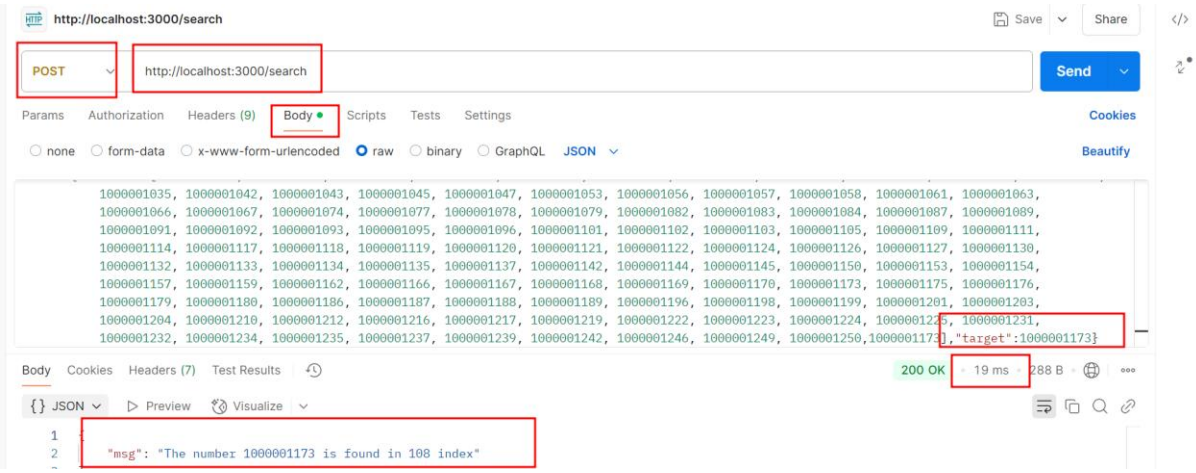
1.13 Now replace the new code provided by git hub copilot and stop and re-run the application and check the search time.

Search.js

```
const express = require('express');
const router = express.Router();

router.post('/', (req, res) => {
  const { numbers, target } = req.body;
  if (!Array.isArray(numbers) || typeof target !== 'number') {
    return res.status(400).json({ error: 'Invalid input' });
  }

  // Create a hash map to store the indices of the numbers
  const numMap = {};
  for (let i = 0; i < numbers.length; i++) {
    numMap[numbers[i]] = i;
  }
```

```
    // Check if the target number exists in the hash map
    const index = numMap[target];
    if (index !== undefined) {
        res.json({"msg":`The number ${target} is found in ${index} index`});
    } else {
        res.json({"msg":`The number ${target} is not in the array`});
    }
});

module.exports = router;
```

1.14 send the request to search the particular element



Here you can find the difference.

You can see the reduced time in the above output.

By following the outlined steps, you have successfully utilized generative AI To analyze and enhance the time complexity of the codebase used in the Expressgo parcel delivery system by leveraging GitHub Copilot to achieve significant improvements in efficiency and effectiveness.