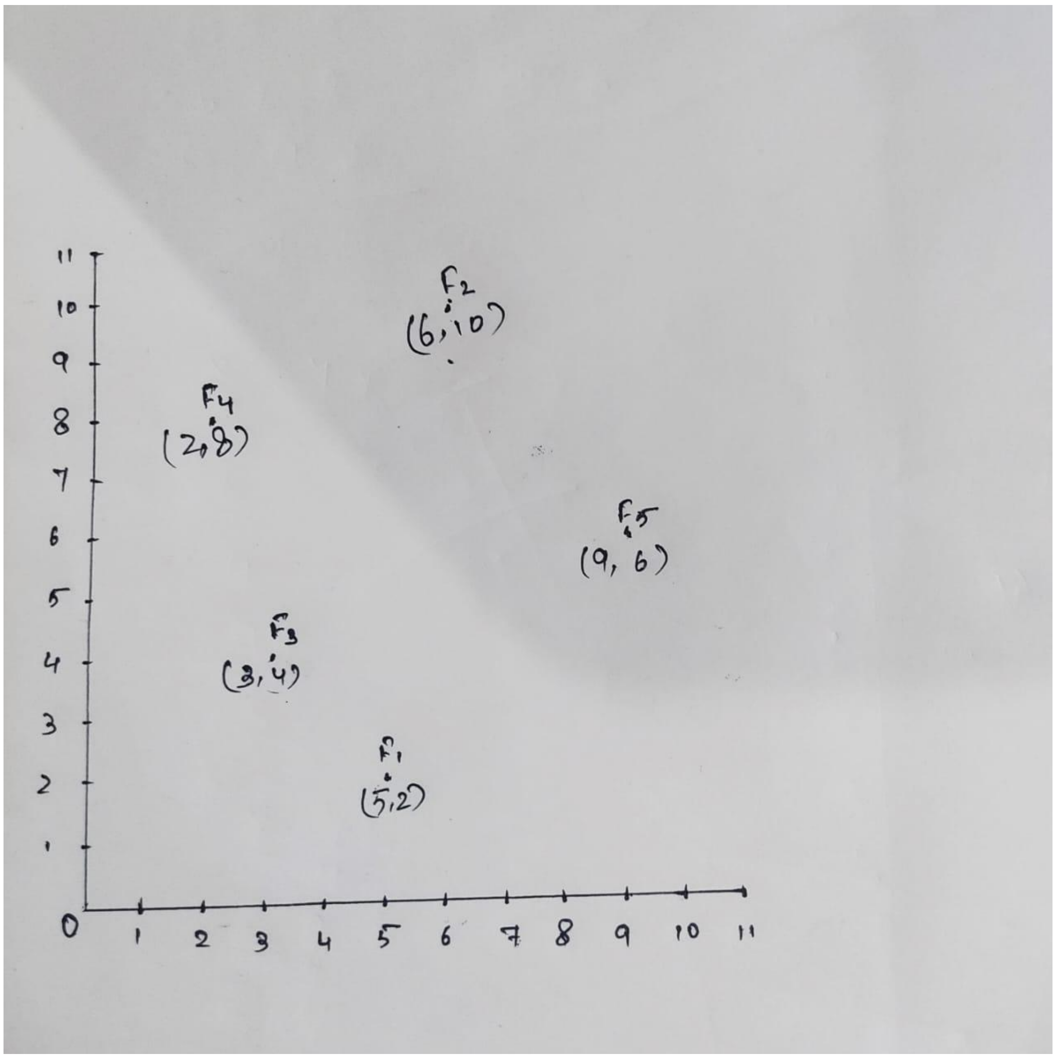
Blogx Assignment

Section A

1. Let's use an example to better understanding and visualise the problem.

   Assume we have 5 factories in a 2D plane, and their Cartesian coordinates are as follows:

   Factory1(5,2)
   Factory2(6,10)
   Factory3(3,4)
   Factory4(2,8)
   Factory5(9,6)



Now, i have to find the strategic location of the warehouse such that the total distance truck needs to travel in order to collect items from all the factories is minimised

Step 1: Extract the x- and y-coordinates from the factory locations:

> X-coordinates are as follows: 5,6,3,2,9
> Y-coordinates are as follows: 2,10,4,8,6

Step 2: Arrange the x and y coordinates in ascending order:

> X-coordinates are as follows: 2, 3, 5, 6, 9
> Y-coordinates are as follows: 2, 4, 6, 8, 10

Step 3: Determine the median of the x- and y-coordinates:

Since there are 5 data points (an odd number), the median is the value in the middle of each sorted list.

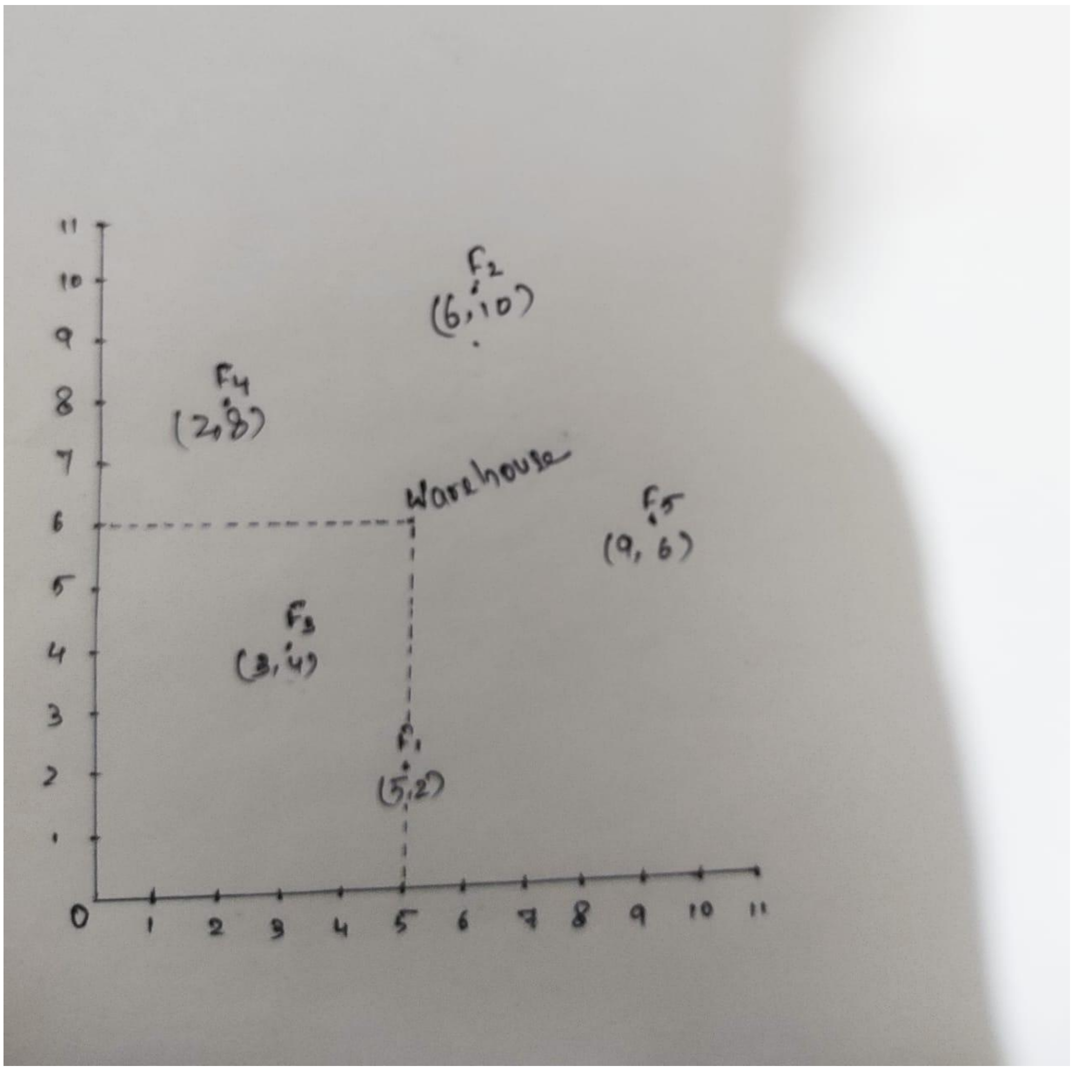
For the x-coordinates, enter:

> Median of x-coordinates = Value of the (5 + 1)/2 -th x-coordinate in the ordered dataset
> = Value of the ordered dataset's 3rd x-coordinate
> = 5

For the y-coordinates, use the following formula:

> Median of y-coordinates = Value of the (5 + 1)/2 -th y-coordinate in the ordered dataset
> = Value of the ordered dataset's 3rd y-coordinate
> = 6

As a result, the x-coordinate median is 5 and the y-coordinate median is 6.



Therefore, the proposed strategic warehouse location that minimizes the total distance that the truck needs to travel to gather things from all factories is (5, 6).

2. To efficiently generate and release unique session IDs with a small memory footprint, we can use a mix of a compact data structure and a free list to keep track of available IDs.

   Use a Bit Vector: A bit vector is a small data structure that represents a bit array. A 32-bit integer can be used as a bit vector to keep track of which IDs are now in use and which are available. Each bit of the integer represents a session ID. When a session ID is in use, the associated bit is set to 1, and when it is not, the bit is set to 0.

   Use a Free List: To keep track of the available IDs, we use a free list. When an ID is released, it is placed on the free list and can be reused for future user sessions.

   ```
   class SessionIdGenerator {
     constructor() {
       this.usedIds = 0; // Bit vector representing used IDs
       this.freeList = []; // Array of available IDs
     }

     getUniqueSessionId() {
       if (this.freeList.length > 0) {
         // If there are available IDs in the free list, reuse one of them
         const sessionId = this.freeList.pop();
         this.usedIds |= (1 << sessionId); // Mark the ID as used in the bit vector
         return sessionId;
       } else {
         // If the free list is empty, find the first available ID in the bit vector
         let sessionId = 0;
         while (this.usedIds & (1 << sessionId)) {
           sessionId++;
         }
         this.usedIds |= (1 << sessionId); // Mark the ID as used in the bit vector
         return sessionId;
       }
     }

     releaseSessionId(sessionId) {
   ```

```
    if (sessionId < 0 || sessionId >= 32) {
      throw new Error("Invalid session ID");
    }

    // Mark the ID as available in the bit vector
    this.usedIds &= ~(1 << sessionId);

    // Add the released ID to the free list
    if (!this.freeList.includes(sessionId)) {
      this.freeList.push(sessionId);
    }
  }
}

// Example usage:
const generator = new SessionIdGenerator();
const user1Id = generator.getUniqueSessionId();
const user2Id = generator.getUniqueSessionId();
console.log(user1Id, user2Id); // Output: 0 1

generator.releaseSessionId(user1Id);
const newUser1Id = generator.getUniqueSessionId();
console.log(newUser1Id); // Output: 0 (The released ID is reused)
```

To control ***session IDs*** and their availability, we define the ***SessionIdGenerator*** class.
The constructor creates a 32-bit integer (initialised to 0) for ***usedIds*** and an empty array for ***freeList***.

The ***getUniqueSessionId()*** method generates a new user's unique session ID. If there are available IDs in the ***freeList***, one of them is reused. Otherwise, it searches the ***usedIds*** bit vector for the first accessible ID and marks it as used by setting the associated bit to 1.

The ***releaseSessionId(sessionId)*** method deactivates an active ID, allowing it to be reused. To signify that the ID is now available, it sets the matching bit in the ***usedIds*** to 0. To maintain track of available IDs, the released ID is added to the ***freeList***.

Overall, this JavaScript implementation, utilizing a bit vector to minimize memory usage and a free list to manage available IDs for reuse.

Section - B

1. we can implement a custom parser function in JavaScript to handle arbitrary-precision integers and floating-point numbers using the built-in `BigInt` and `Number` types.

```
function parseJSON(jsonString) {
  try {
    const parsedData = JSON.parse(jsonString);
    return parseValue(parsedData);
  } catch (error) {
    throw new Error('Invalid JSON string');
  }

  function parseValue(value) {
    if (typeof value === 'string') {
      // Check if the value is a valid integer or floating-point number
      if (/^-?\d+$/.test(value)) {
        return BigInt(value);
      } else if (/^-?\d+\.\d+$/.test(value)) {
        return Number.parseFloat(value);
      } else {
        return value;
      }
    } else if (Array.isArray(value)) {
      return value.map(parseValue);
    } else if (typeof value === 'object' && value !== null) {
      const newObj = {};
      for (const key in value) {
        newObj[key] = parseValue(value[key]);
      }
      return newObj;
    } else {
      return value;
    }
  }

}
```

```
// Test the function with an example JSON string
const jsonString = '{"name": "John", "age": "30", "balance": "100.50",
"is_vip": true}';
const parsedObject = parseJSON(jsonString);
console.log(parsedObject);
```

2. To ensure you don't exceed the limit of 15 calls per minute and avoid penalties, you can implement a rate-limiting mechanism. Here's a algorithm for achieving this:
    1) Keep track of how many API requests have been performed in the last minute.
    2) Make the API request and increment the call count for the current minute if the number of calls is less than 15.
    3) To avoid penalties, if the number of calls hits 15, wait one minute before making another call.
    4) Reset the call count and perform the next API call after the one-minute pause.
    5) Steps 1–4 should be repeated as needed to ensure that the rate restriction is met.

    Below is an implementation of the api- rate-limiting using JavaScript:

```
class APICallManager {
  constructor() {
    this.callsMade = 0;
    this.lastCallTime = null;
  }

  call_me(inputData) {
    // Check if a call was made within the last minute
    const currentTime = Date.now();
    if (this.lastCallTime && currentTime - this.lastCallTime < 60000) {
      // Check if we have reached the limit of 15 calls
      if (this.callsMade >= 15) {
        // Calculate the remaining time in this minute before making the call
        const waitTime = 60000 - (currentTime - this.lastCallTime);

        return new Promise((resolve) => {
          // Wait for the remaining time before resolving the promise
          setTimeout(() => {
            this.callsMade = 0;
            this.lastCallTime = null;
            resolve(this.makeAPICall(inputData));
          }, waitTime);
        });
      }
    }

    // Make the API call and update call count and last call time
    const response = this.makeAPICall(inputData);
    this.callsMade++;
    this.lastCallTime = currentTime;

    return response;
  }

  makeAPICall(inputData) {
    // Here you would implement the actual API call using the provided
function call_me()
    // and return the response from the API.
    // Replace this with the actual API call implementation using the
provided function.
    return call_me(inputData);
  }
}
```