Talent Desk

Detect Sum Algorithm

Time and memory complexity comparison

Time Complexity

1. **Version A** (Using `sumIndexOfAll` function)
    - For each pair of elements in the array, the `sumIndexOfAll` function is called.
    - Inside `sumIndexOfAll`, it iterates over the entire array (`arr`) again to find indices where the sum equals the given value.
    - The time complexity of `sumIndexOfAll` is O(n), where n is the length of the input array `arr`.
    - Therefore, for each pair of elements, the time complexity is O(n).
    - Since there are O(n^2) pairs of elements in total, the overall time complexity of this version is O(n^3).
2. **Version B (**Using the optimized version with direct index lookup)
    - We create an index map to store the indices of elements in the array. This has a time complexity of O(n).
    - Then, we iterate over each pair of elements in the array.
    - For each pair, we directly look up the indices of their sum in the index map, which has O(1) time complexity.
    - The overall time complexity of this version is dominated by the nested loops, resulting in O(n^2) time complexity.

Comparing the two versions:

- The optimized version B with direct index lookup has a better time complexity of O(n^2) compared to O(n^3) in the version A (using `sumIndexOfAll)`.
- Therefore, the optimized version is more efficient in terms of time complexity, especially for larger input arrays.

Memory Complexity

1. **Version A** (Using `sumIndexOfAll` function):
    - In this version, there is no additional memory overhead apart from the input array `arr`.
    - The `sumIndexOfAll` function creates a new array to store indices where the sum equals the given value. This array could potentially grow to include almost all indices of the input array, depending on the elements and their sums.
    - Therefore, the memory complexity of this version is directly proportional to the size of the input array `arr`, which is O(n), where n is the length of the input array.
2. **Version B** (Using the optimized version with direct index lookup):
    - In this version, we create an index map to store the indices of elements in the input array.
    - The size of the index map depends on the unique elements in the input array. If there are m unique elements in the array, the index map would have at most m entries.
    - Each entry in the index map stores an array of indices, which could potentially contain all indices of the input array if all elements are identical.
    - Therefore, the memory complexity of this version is also directly proportional to the size of the input array `arr`, but it's also influenced by the number of unique elements in the array.

Comparing the memory complexities of both versions:

- Both versions have a memory complexity proportional to the size of the input array `arr`, which is O(n), where n is the length of the input array.
- However, the optimized version with direct index lookup might have a slightly higher memory overhead due to the additional index map storing arrays of indices.

- Overall, the memory complexities of both versions are similar, with the optimized version potentially having a slightly higher memory overhead due to the index map. Nevertheless, the difference is negligible compared to the size of the input array.

Version 1 (O(n)^3)

```
const sumIndexOfAll = (arr, val) => arr.reduce((acc, el, i) => (el === val ? [...acc, i] : acc),
[]);
function detectSums(arr) {
    const result = [];
    let sumIndexes;
    for (let i = 0; i < arr.length; i++) {
        for (let j = i + 1; j < arr.length; j++) {
            const sum = arr[i] + arr[j];
            sumIndexes = sumIndexOfAll(arr, sum).filter(item => ![i, j].includes(item));
            for (sumIndex of sumIndexes) {
                result.push({ pA: i, pB: j, sum: sumIndex });
            }
        }
    }

    return result;
}


```

Version 2 (O(n)^2)

```
function createIndexMap(arr) {
    const indexMap = {};
    arr.forEach((element, index) => {
        if (indexMap[element] === undefined) {
            indexMap[element] = [index];
        } else {
            indexMap[element].push(index);
        }
    });
    return indexMap;
}


function detectSums(arr) {
    const result = [];
    //  Creates an object that we can use for querying the indice of an element in O(1) time.
    const indexMap = createIndexMap(arr);
    let sumIndexes;
    for (let i = 0; i < arr.length; i++) {
        for (let j = i + 1; j < arr.length; j++) {
            const sum = arr[i] + arr[j];
            // Get the indice of sum,
            sumIndexes = indexMap[sum];
            if (sumIndexes === undefined) {
                sumIndexes = []
            }
            // Exclude indices of sum that are either i or j.
            sumIndexes = sumIndexes.filter(item => ![i, j].includes(item));
            for (sumIndex of sumIndexes) {
                result.push({ pA: i, pB: j, sum: sumIndex });
            }
        }
    }

    return result;
}
```