2628. JSON Deep Equal

Medium

Leetcode Link

Problem Description

The problem requires developing a function to check if two values, `o1` and `o2`, are *deeply equal*. This concept of deep equality extends beyond surface-level comparison (e.g., simply checking if two numbers are the same) and delves into structured data such as arrays and objects. Here, the comparison must be thorough:

- For **primitive values** (like numbers or strings), they are deeply equal if they are strictly equal (`===`).
- For **arrays**, they are deeply equal if they have the same number of elements, each element is in the same order, and each corresponding pair of elements is also deeply equal.
- For **objects** (ignoring those that are instances of classes), they are deeply equal if they have the same set of keys and each corresponding value pair is deeply equal.

An additional constraint is that we must not use any external deep comparison function like lodash's `_.isEqual()`.

Intuition

To determine if two given values are deeply equal, the solution takes a recursive approach:

- A straightforward comparison is made for primitive values and `null`.
- If the values are arrays, it checks if they are the same length and then compares each corresponding pair of elements recursively.
- If the values are objects, it gets a list of keys for each, checks if those are the same length (to quickly rule out objects with different sets of keys), and then recursively checks deeply equality for each value associated with each key in the lists.

This approach works by systematically breaking down complex structures (arrays and objects) into simpler problems that are essentially similar to the base case: comparing primitives. Recursion is used to apply the same logic at every level of nested structures, allowing for a consistent method of comparison throughout the entire depth of the data.

Solution Approach

The solution provided above employs recursion to implement deep equality checking between two values, `o1` and `o2`. The recursion acts as a control flow mechanism that allows the function to traverse deeper into arrays and object properties if necessary. Here are the key parts of the algorithm with explanations:

1. **Base Case for Primitives and `null`:** Primitive values and `null` can be checked with a strict equality `===` comparison. If `o1` is `null` or not an object, the function returns the result of `o1 === o2`.

2. **Type Checking:** The algorithm quickly rules out different types by checking whether `o1` and `o2` are of the same type. This is done using the `typeof` operator.

3. **Array Check and Comparison:** The function needs to identify if the values being compared are arrays, which is achieved using `Array.isArray(o1)` and `Array.isArray(o2)`. If one is an array and the other is not, they can't be deeply equal, so the function returns `false`.

4. **Deep Array Equality:** Once it's determined that both `o1` and `o2` are arrays, the lengths are compared. If the lengths don't match, the arrays can't be deeply equal. If the lengths are the same, the function iteratively compares each element pair using a for-loop while recursively calling `areDeeplyEqual` to check the deep equality of each pair.

5. **Object Property Equality:** If `o1` and `o2` are not arrays, the code treats them as objects. It retrieves the keys from both objects with `Object.keys` and compares the lengths of these keys arrays. If they're unequal, it can immediately return `false`. Otherwise, it iterates over the keys of one object, calling `areDeeplyEqual` recursively for each corresponding value pair.

6. **Recursive Descent:** The recursion is essential to the algorithm, allowing the equality check to be valid no matter how deeply nested the arrays or objects are. Each call to `areDeeplyEqual` within the loops ensures arrays and objects nested within other arrays and objects are also subjected to this thorough checking.

Given the recursive nature of the function, it implicitly utilizes the call stack as its data structure, each frame holding context for a comparison at a particular level of depth. The patterns here are typical for recursive tree/graph traversal, though simplified because the inputs are limited to arrays and objects without cycles (thanks to the JSON restriction).

No additional data structures are utilized since the structure of the original inputs is directly traversed without the need for transformation or storage of intermediate results beyond what the recursive calls do inherently.

This implementation ensures a full deep equality check in line with the requirements outlined in the problem description.

Example Walkthrough

Let's consider an example where we have two values that we want to compare using the solution approach described above. Our first value `o1` is an object, and the second value `o2` is another object. We want to determine if they are deeply equal.

```
let o1 = {
  a: 1,
  b: [1, 2, {c: 3}],
  d: {e: {f: 4}}
};

let o2 = {
  a: 1,
  b: [1, 2, {c: 3}],
  d: {e: {f: 4}}
};
```

Both `o1` and `o2` look the same at first glance, but we need to check each element in detail.

1. **Base Case for Primitives and `null`:**
   - The algorithm starts by checking `o1.a` and `o2.a`, which are both primitive values.
   - Using the `===` operator, we get that 1 === 1, which is true.
2. **Type Checking:**
   - Next, `o1.b` and `o2.b` are both arrays, so their types match (`typeof o1.b` returns "object", as it does for `o2.b` because arrays are objects in JavaScript).
3. **Array Check and Comparison:**
   - Since both are arrays, we use `Array.isArray(o1.b)` and `Array.isArray(o2.b)` to confirm.
4. **Deep Array Equality:**
   - We compare the lengths of `o1.b` and `o2.b` which are both 3, so we move forward.
   - Then we check each element at the corresponding indexes:
     - First pair: `o1.b[0] === o2.b[0]` (1 === 1 is true).
     - Second pair: `o1.b[1] === o2.b[1]` (2 === 2 is true).
     - Third pair: They are objects `{c: 3}`, so the algorithm checks them recursively.
     - During this recursive step, the third pair passes the base case for primitives (`o1.b[2].c === o2.b[2].c` which is true).
5. **Object Property Equality:**
   - Lastly, `o1.d` and `o2.d` are objects, not arrays. The keys ("e") and their lengths match, so we proceed to their values.
   - `o1.d.e` and `o2.d.e` are also objects, triggering another recursion. They have the same set of keys "f", and their values are primitive, so `o1.d.e.f === o2.d.e.f` (4 === 4 is true).
6. **Recursive Descent:**
   - Throughout the comparison, the `areDeeplyEqual` function calls itself for each pair of values that aren't strictly equal primitives, which includes objects and arrays regardless of their nesting level.

Since all checks passed without returning `false`, `o1` and `o2` are determined to be deeply equal by the implementation.

Python Solution

```python
def are_deeply_equal(value1, value2):
    """
    Determines if two values are deeply equal.
    Handles comparisons of primitives, arrays, and objects.
    """
    # Check for direct equality or if both are 'None'
    if value1 is None or not isinstance(value1, (list, dict)):
        return value1 == value2

    # If types are different, they are not equal
    if type(value1) != type(value2):
        return False

    # If values are lists (similar to JavaScript arrays), compare their elements
    if isinstance(value1, list):
        # Lists of different lengths are not equal
        if len(value1) != len(value2):
            return False

        # Recursively compare each element of the list
        for index in range(len(value1)):
            if not are_deeply_equal(value1[index], value2[index]):
                return False
        return True

    elif isinstance(value1, dict):
        # Handle comparison of dictionary (object in JavaScript) items

        # Objects with different numbers of keys are not equal
        if len(value1) != len(value2):
            return False

        # Recursively compare each item of the dictionary
        for key in value1:
            if key not in value2 or not are_deeply_equal(value1[key], value2[key]):
                return False

        # All properties match, so the objects are deeply equal
        return True

    # For all other types that are not list or dict, check for direct equality
    return value1 == value2

```

Java Solution

```java
import java.util.Arrays;
import java.util.Objects;

public class DeepEquality {

    // Determines if two values are deeply equal.
    // Handles comparisons of primitives, arrays, and objects.
    public static boolean areDeeplyEqual(Object value1, Object value2) {
        // Check for direct equality or if both values are null
        if (value1 == value2) {
            return true;
        }

        // If one is null and the other is not, they cannot be equal
        if (value1 == null || value2 == null) {
            return false;
        }

        // Check if types are different, then values are not equal
        if (value1.getClass() != value2.getClass()) {
            return false;
        }

        // If values are arrays, compare their elements
        if (value1.getClass().isArray()) {
            // Compare arrays deeply, considering multi-dimensional cases
            return Arrays.deepEquals(new Object[]{value1}, new Object[]{value2});
        } else if (value1 instanceof Iterable) {
            // This if clause could handle the comparison of Iterable objects (like Lists)
            // For simplicity, this feature is not implemented in this example
            // Implement Iterable comparison logic here if required
            return false; // Placeholder, replace with actual comparison code
        } else if (value1 instanceof Objects) {
            // Handle comparison of object properties
            // For simplicity, this feature is not implemented in this example
            // Implement custom object comparison logic here if required
            return Objects.equals(value1, value2); // Placeholder, replace with deep comparison
        } else {
            // For non-array and non-Iterable objects, check equality
            return value1.equals(value2);
        }
    }

    public static void main(String[] args) {
        // Example usage of areDeeplyEqual method
        int[] array1 = {1, 2, 3};
        int[] array2 = {1, 2, 3};
        System.out.println("Are the arrays deeply equal? " + areDeeplyEqual(array1, array2));
    }
}

```

C++ Solution

```cpp
#include <iostream>
#include <vector>
#include <unordered_map>

// Helper function to compare two vectors deeply.
// Returns true if given vectors are deeply equal; false otherwise.
bool AreVectorsDeeplyEqual(const std::vector<any>& vector1, const std::vector<any>& vector2);

// Helper function to compare two maps deeply.
// Returns true if given maps are deeply equal; false otherwise.
bool AreMapsDeeplyEqual(const std::unordered_map<std::string, any>& map1, const std::unordered_map<std::string, any>& map2);

// Determines if two values are deeply equal.
// Handles comparisons of primitives, vectors, and maps.
bool AreDeeplyEqual(const any& value1, const any& value2) {
    // Check for direct equality or if both are null
    if (value1.IsNull() || !value1.IsObject()) {
        return value1 == value2;
    }

    // If types are different, they are not equal
    if (value1.Type() != value2.Type()) {
        return false;
    }

    // Check if both values are vectors (and subsequently both are tuples)
    if (value1.IsVector() != value2.IsVector()) {
        return false;
    }

    // If values are vectors, compare their elements
    if (value1.IsVector()) {
        return AreVectorsDeeplyEqual(value1.AsVector(), value2.AsVector());
    } else {
        // Handle comparison of map properties
        return AreMapsDeeplyEqual(value1.AsMap(), value2.AsMap());
    }
}

bool AreVectorsDeeplyEqual(const std::vector<any>& vector1, const std::vector<any>& vector2) {
    // Vectors of different lengths are not equal
    if (vector1.size() != vector2.size()) {
        return false;
    }
    // Recursively compare each element of the vector
    for (size_t index = 0; index < vector1.size(); ++index) {
        if (!AreDeeplyEqual(vector1[index], vector2[index])) {
            return false;
        }
    }
    return true;
}

bool AreMapsDeeplyEqual(const std::unordered_map<std::string, any>& map1, const std::unordered_map<std::string, any>& map2) {
    // Maps with different numbers of keys are not equal
    if (map1.size() != map2.size()) {
        return false;
    }

    // Recursively compare each corresponding property of the map
    for (const auto& pair : map1) {
        const auto& key = pair.first;
        if (map2.find(key) == map2.end() || !AreDeeplyEqual(map1.at(key), map2.at(key))) {
            return false;
        }
    }

    // All properties match, so the maps are deeply equal
    return true;
}

// Note: 'any' is a placeholder here and would need to be defined or replaced with an appropriate type system
//       capable of handling different types (e.g., boost::any, std::variant from C++17, or a custom any class).

```

Typescript Solution

```typescript
// Determines if two values are deeply equal.
// Handles comparisons of primitives, arrays, and objects.
function areDeeplyEqual(value1: any, value2: any): boolean {
    // Check for direct equality or if both are null
    if (value1 === null || typeof value1 !== 'object') {
        return value1 === value2;
    }

    // If types are different, they are not equal
    if (typeof value1 !== typeof value2) {
        return false;
    }

    // Check if both values are arrays (and subsequently both are tuples)
    if (Array.isArray(value1) !== Array.isArray(value2)) {
        return false;
    }

    // If values are arrays, compare their elements
    if (Array.isArray(value1)) {
        // Arrays of different lengths are not equal
        if (value1.length !== value2.length) {
            return false;
        }
        // Recursively compare each element of the array
        for (let index = 0; index < value1.length; index++) {
            if (!areDeeplyEqual(value1[index], value2[index])) {
                return false;
            }
        }
        return true;
    } else {
        // Handle comparison of object properties
        const keys1 = Object.keys(value1);
        const keys2 = Object.keys(value2);

        // Objects with different numbers of keys are not equal
        if (keys1.length !== keys2.length) {
            return false;
        }

        // Recursively compare each corresponding property of the object
        for (const key of keys1) {
            if (!areDeeplyEqual(value1[key], value2[key])) {
                return false;
            }
        }

        // Ensure that value2 does not have extra properties
        for (const key of keys2) {
            if (!value1.hasOwnProperty(key)) {
                return false;
            }
        }

        // All properties match, so the objects are deeply equal
        return true;
    }
}

```

Time and Space Complexity

Time Complexity

The time complexity of the `areDeeplyEqual` function is `O(N)`, where `N` represents the total number of elements or properties within the objects (or arrays) being compared. This analysis is based on the fact that each element or property must be visited once to compare them. In the case of arrays, the comparison happens sequentially through iteration, and for objects, it happens by cycling through all the keys.

However, the worst-case time complexity can be more accurately described as `O(N * K)`, where `N` is the number of elements or properties in the objects and `K` is the average depth of nested structures within those objects. Each recursive call to `areDeeplyEqual` for nested objects or arrays adds an additional layer of complexity that is linear to the depth `K` of the objects.

Space Complexity

The space complexity can also be seen as `O(K)`, due to the recursion stack that grows in proportion to the depth of the nested structures (arrays or objects). Each recursive call takes up some space on the call stack, and in the case of deeply nested objects or arrays, the number of recursive calls corresponds to the depth of those structures.