2628. JSON Deep Equal Medium Leetcode Link

Problem Description

corresponding value pair is deeply equal.

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 (===).

The problem requires developing a function to check if two values, o1 and o2, are deeply equal. This concept of deep equality

- 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
- 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:

 If the values are arrays, it checks if they are the same length and then compares each corresponding pair of elements recursively.

A straightforward comparison is made for primitive values and null.

- 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.
- 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.

This approach works by systematically breaking down complex structures (arrays and objects) into simpler problems that are

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

1. Base Case for Primitives and null: Primitive values and null can be checked with a strict equality === comparison. If o1 is null

the key parts of the algorithm with explanations:

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,

loop while recursively calling areDeeplyEqual to check the deep equality of each pair.

transformation or storage of intermediate results beyond what the recursive calls do inherently.

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

Using the === operator, we get that 1 === 1, which is true.

Then we check each element at the corresponding indexes:

The algorithm starts by checking o1.a and o2.a, which are both primitive values.

We compare the lengths of o1.b and o2.b which are both 3, so we move forward.

arrays and objects are also subjected to this thorough checking.

the arrays can't be deeply equal. If the lengths are the same, the function iteratively compares each element pair using a for-

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

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

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.

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

b: [1, 2, {c: 3}], d: {e: {f: 4}}

1. Base Case for Primitives and null:

2. Type Checking:

4. Deep Array Equality:

5. Object Property Equality:

6. Recursive Descent:

b: [1, 2, {c: 3}],

d: {e: {f: 4}}

1 let o1 = {

let o2 = {

11 };

3. Array Check and Comparison: Since both are arrays, we use Array.isArray(o1.b) and Array.isArray(o2.b) to confirm.

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

During this recursive step, the third pair passes the base case for primitives (o1.b[2].c === o2.b[2].c which is true).

Lastly, o1.d and o2.d are objects, not arrays. The keys ("e") and their lengths match, so we proceed to their values.

First pair: o1.b[0] === o2.b[0] (1 === 1 is true). Second pair: o1.b[1] === o2.b[1] (2 === 2 is true).

def are_deeply_equal(value1, value2):

if type(value1) != type(value2):

if len(value1) != len(value2):

for index in range(len(value1)):

return False

if len(value1) != len(value2):

if isinstance(value1, list):

return False

return False

for key in value1:

} else {

38

39

40

41

42

43

44

45

46

47

48

49

50

C++ Solution

1 #include <iostream>

#include <unordered_map>

2 #include <vector>

51

12

16

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

return False

return True

Determines if two values are deeply equal.

Handles comparisons of primitives, arrays, and objects.

Lists of different lengths are not equal

Recursively compare each element of the list

Recursively compare each item of the dictionary

If values are lists (similar to JavaScript arrays), compare their elements

if not are_deeply_equal(value1[index], value2[index]):

// For non-array and non-Iterable objects, check equality

System.out.println("Are the arrays deeply equal? " + areDeeplyEqual(array1, array2));

bool AreVectorsDeeplyEqual(const std::vector<any>& vector1, const std::vector<any>& vector2);

11 bool AreMapsDeeplyEqual(const std::unordered_map<std::string, any>& map1, const std::unordered_map<std::string, any>& map2);

return value1.equals(value2);

// Example usage of areDeeplyEqual method

public static void main(String[] args) {

// Helper function to compare two vectors deeply.

// Helper function to compare two maps deeply.

14 // Handles comparisons of primitives, vectors, and maps.

if (value1.IsNull() || !value1.IsObject()) {

// If types are different, they are not equal

if (value1.IsVector() != value2.IsVector()) {

// If values are vectors, compare their elements

// Handle comparison of map properties

// Check if both values are vectors (and subsequently both are tuples)

return AreVectorsDeeplyEqual(value1.AsVector(), value2.AsVector());

// Recursively compare each corresponding property of the object

if (!areDeeplyEqual(value1[key], value2[key])) {

// Ensure that value2 does not have extra properties

// All properties match, so the objects are deeply equal

if (!value1.has0wnProperty(key)) {

for (const key of keys1) {

for (const key of keys2) {

return true;

Time and Space Complexity

return false;

return false;

return AreMapsDeeplyEqual(value1.AsMap(), value2.AsMap());

15 bool AreDeeplyEqual(const any& value1, const any& value2) {

// Check for direct equality or if both are null

13 // Determines if two values are deeply equal.

return value1 == value2;

return false;

return false;

if (value1.IsVector())

} else {

if (value1.Type() != value2.Type()) {

6 // Returns true if given vectors are deeply equal; false otherwise.

10 // Returns true if given maps are deeply equal; false otherwise.

int[] array1 = {1, 2, 3};

int[] array2 = {1, 2, 3};

if key not in value2 or not are_deeply_equal(value1[key], value2[key]):

are objects in JavaScript).

olide and olide 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).

Third pair: They are objects {c: 3}, so the algorithm checks them recursively.

- 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
 - # Check for direct equality or if both are 'None' 6 if value1 is None or not isinstance(value1, (list, dict)): return value1 == value2 8 9 10 # If types are different, they are not equal

25 26 elif isinstance(value1, dict): 27 # Handle comparison of dictionary (object in JavaScript) items 28 29 # Objects with different numbers of keys are not equal

11

12

13

14

15

16

17

18

19

20

21

22

23

24

30

31

32

33

34

35

```
36
                     return False
 37
 38
             # All properties match, so the objects are deeply equal
 39
             return True
 40
         # For all other types that are not list or dict, check for direct equality
 41
 42
         return value1 == value2
 43
Java Solution
   import java.util.Arrays;
  2 import java.util.Objects;
     public class DeepEquality {
         // Determines if two values are deeply equal.
  6
         // 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
  9
 10
             if (value1 == value2) {
 11
                 return true;
 12
 13
 14
             // If one is null and the other is not, they cannot be equal
             if (value1 == null || value2 == null) {
 15
                 return false;
 16
 17
 18
 19
             // Check if types are different, then values are not equal
 20
             if (value1.getClass() != value2.getClass()) {
 21
                 return false;
 22
 23
 24
             // If values are arrays, compare their elements
 25
             if (value1.getClass().isArray()) {
 26
                 // Compare arrays deeply, considering multi-dimensional cases
 27
                 return Arrays.deepEquals(new Object[]{value1}, new Object[]{value2});
             } else if (value1 instanceof Iterable) {
 28
 29
                 // This if clause could handle the comparison of Iterable objects (like Lists)
 30
                 // For simplicity, this feature is not implemented in this example
                 // Implement Iterable comparison logic here if required
 31
                 return false; // Placeholder, replace with actual comparison code
 32
             } else if (value1 instanceof Objects) {
 33
                 // Handle comparison of object properties
 34
                 // For simplicity, this feature is not implemented in this example
 35
 36
                 // Implement custom object comparison logic here if required
 37
                 return Objects.equals(value1, value2); // Placeholder, replace with deep comparison
```

44 45 46 47

```
bool AreVectorsDeeplyEqual(const std::vector<any>& vector1, const std::vector<any>& vector2) {
         // Vectors of different lengths are not equal
 41
         if (vector1.size() != vector2.size()) {
 42
 43
             return false;
         // Recursively compare each element of the vector
         for (size t index = 0; index < vector1.size(); ++index) {</pre>
             if (!AreDeeplyEqual(vector1[index], vector2[index])) {
 48
                 return false;
 49
 50
 51
         return true;
 52 }
 53
    bool AreMapsDeeplyEqual(const std::unordered_map<std::string, any>& map1, const std::unordered_map<std::string, any>& map2) {
 55
         // Maps with different numbers of keys are not equal
 56
         if (map1.size() != map2.size()) {
 57
             return false;
 58
 59
 60
         // Recursively compare each corresponding property of the map
 61
         for (const auto& pair : map1) {
 62
             const auto& key = pair.first;
 63
             if (map2.find(key) == map2.end() || !AreDeeplyEqual(map1.at(key), map2.at(key))) {
 64
                 return false;
 65
 66
 67
 68
         // All properties match, so the maps are deeply equal
 69
         return true;
 70
 71
 72 // Note: 'any' is a placeholder here and would need to be defined or replaced with an appropriate type system
 73 //
              capable of handling different types (e.g., boost::any, std::variant from C++17, or a custom any class).
 74
Typescript Solution
  1 // Determines if two values are deeply equal.
  2 // 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;
  6
  8
  9
         // If types are different, they are not equal
         if (typeof value1 !== typeof value2) {
 10
 11
             return false;
 12
 13
 14
         // Check if both values are arrays (and subsequently both are tuples)
         if (Array.isArray(value1) !== Array.isArray(value2)) {
 15
 16
             return false;
 17
 18
 19
         // If values are arrays, compare their elements
 20
         if (Array.isArray(value1))
 21
             // Arrays of different lengths are not equal
 22
             if (value1.length !== value2.length) {
 23
                 return false;
 24
 25
             // Recursively compare each element of the array
 26
             for (let index = 0; index < value1.length; index++) {</pre>
 27
                 if (!areDeeplyEqual(value1[index], value2[index])) {
 28
                     return false;
 29
 30
 31
             return true;
 32
         } else {
 33
             // Handle comparison of object properties
 34
             const keys1 = Object.keys(value1);
 35
             const keys2 = Object.keys(value2);
 36
 37
             // Objects with different numbers of keys are not equal
 38
             if (keys1.length !== keys2.length) {
 39
                 return false;
 40
```

Time Complexity

41

42

43

44

45

46

47

48

49

50

51

52

53

54

55

56

57

58

59

60

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

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

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.