# 2755. Deep Merge of Two Objects

Medium

Leetcode Link

## Problem Description

This problem involves creating a function that can deeply merge two objects or arrays based on specific rules. A deep merge means that any nested structures within the objects or arrays will also be merged according to certain rules. Here are the rules for deep merging provided in the problem:

- If both values are objects, the merged result should include all keys from both objects. If a key is present in both, their associated values should be deep merged. If a key is only present in one object, its value should simply be included.
- If both values are arrays, the merged array should be as long as the longer of the two original arrays. Just like with objects, apply the same deep merge logic to each position in the arrays, treating the indices as if they were keys.
- In any other case (e.g., values are of different types or not objects/arrays), the result should be the second value `obj2`.

The problem statement clarifies that the inputs `obj1` and `obj2` are the results of `JSON.parse()`, meaning they are valid JavaScript objects or arrays that have been parsed from JSON strings.

## Intuition

The solution is based on recursion, a common technique for dealing with nested data structures like objects and arrays.

- We first check if the inputs are both objects or arrays since the merging logic applies only in those cases. If either `obj1` or `obj2` is not an object, or they are not the same type (one is an array and the other is not), we immediately return `obj2`.
- If both are objects or arrays, we iterate through the keys of `obj2` and apply the `deepMerge` function to each key. This means we attempt to merge `obj1[key]` and `obj2[key]`, recursively ensuring that any nested objects or arrays are also merged correctly.
- We then return `obj1` as it now contains merged values from `obj2`.

This recursive strategy neatly handles arbitrarily complex and deep object structures and directly maps onto the merging rules provided in the problem statement. The base case of the recursion is effectively whenever we reach a point where the values to be merged are not both objects or arrays.

## Solution Approach

The solution uses a recursive function `deepMerge` to merge two potentially complex data structures, which could be objects or arrays. Here's how this is carried out:

- First, we define *helper* functions `isObj` and `isArr` to check if a value is an object or array, respectively. These are used to ensure that we only apply merging logic to appropriate data types.

```
const isObj = (obj) => obj && typeof obj === 'object';
const isArr = (obj) => Array.isArray(obj);
```

- We then examine the input values `obj1` and `obj2`. If either is not an object (or if one is an array and the other isn't), we return `obj2`. This serves as the base case for non-object/array values or mismatched type pairs.

```
if (!isObj(obj1) || !isObj(obj2)) {
    return obj2;
}

if (isArr(obj1) !== isArr(obj2)) {
    return obj2;
}
```

- Next, the function iterates through the keys of `obj2` using a `for...in` loop. For each key, we call `deepMerge` on the value at that key from both `obj1` and `obj2`, essentially performing a merge operation on any nested structures:

```
for (const key in obj2) {
    obj1[key] = deepMerge(obj1[key], obj2[key]);
}
```

- The operation `obj1[key] = deepMerge(obj1[key], obj2[key])` ensures that if the same key is present in both `obj1` and `obj2`, their values will be merged. Otherwise, if the key exists only in `obj2`, it gets added to `obj1`.
- Finally, `obj1` is returned, now containing merged data from both `obj1` and `obj2`.

Recursion is the key algorithmic pattern that makes the implementation succinct and able to cope with objects and arrays of any depth. The `deepMerge` function merges `obj2` into `obj1` at every level of the data structure, complying with the rules laid out in the problem statement.

### Example Walkthrough

To illustrate the solution approach using a small example, let's consider two objects that we wish to merge using the `deepMerge` function described in the solution approach.

```
let obj1 = {
  a: 1,
  b: { c: 3, d: 4 },
  e: [5, 6],
};

let obj2 = {
  b: { c: 8, e: 9 },
  e: [7, 10],
  f: 11,
};
```

Here, we want to merge `obj2` into `obj1`. Here's a step-by-step walkthrough of how the deep merge would function:

1. Since both `obj1` and `obj2` are objects, we proceed with merging. We do not immediately return `obj2` because the base case does not apply.
2. We iterate over the keys in `obj2`. The keys are `b`, `e`, and `f`.
3. For key `b`, `obj1[b]` and `obj2[b]` are both objects:

```
{ obj1[b]: { c: 3, d: 4 }
  obj2[b]: { c: 8, e: 9 }
}
```

So, we call `deepMerge` on these two objects:

- Both have a key `c`, so we merge the values. The value from `obj2` (8) takes precedence.
- `obj1[b]` has a key `d` that is not present in `obj2[b]`; it remains as is.
- `obj2[b]` has a key `e` that is not present in `obj1[b]`; it gets added to the result.

The merged result for key `b` is: `{ c: 8, d: 4, e: 9 }`.

4. For key `e`, `obj1[e]` is an array and so is `obj2[e]`:

```
{ obj1[e]: [5, 6]
  obj2[e]: [7]
}
```

We extend `obj1[e]` to match the length of the longer array, but since `obj2[e]` is the shorter one, no change is made to the length. The deep merge logic doesn't introduce new items from the longer array unless there are corresponding items in `obj2`. So the merged result for key `e` is `[7, 6]`.

5. For key `f`, which is only present in `obj2`, we simply add it to `obj1`:

```
{ obj1[f]: 10
}
```

The final merged object `obj1` now looks like this:

```
{
  a: 1,
  b: { c: 8, d: 4, e: 9 },
  e: [7, 6],
  f: 10,
}
```

This example makes it clear how each step in the solution approach actively contributes to the desired final result according to the rules of deep merging. Recursion ensures that objects and arrays nested at any level are merged correctly and the rules are followed.

## Python Solution

```python
def deep_merge(first_object, second_object):
    """
    Function to deeply merge two objects, combining their properties and
    sub-properties.

    :param first_object: The first dictionary or object to merge.
    :param second_object: The second dictionary or object to merge.
    :return: A dictionary with properties from both first_object and second_object merged.
    """
    # Helper function to check if a value is a dictionary.
    def is_object(item):
        return isinstance(item, dict)

    # Helper function to check if a value is a list (array).
    def is_array(item):
        return isinstance(item, list)

    # If either argument is not a dictionary (object), return the second_object.
    if not is_object(first_object) or not is_object(second_object):
        return second_object

    # If the types of container for both objects differ (one is a list and one is a dictionary), return second_object.
    if is_array(first_object) != is_array(second_object):
        return second_object

    # Iterate over each property of the second_object.
    for key in second_object:
        # If the property key is in the dictionary's own properties (not inherited),
        if key in second_object:
            # Recursively merge properties from both objects.
            first_object[key] = deep_merge(first_object.get(key), second_object[key])

    # Return the merged first_object, which now contains properties from both objects.
    return first_object

# Example usage:
# obj1 = {"a": 1, "c": 3}
# obj2 = {"a": 2, "b": 2}
# merged_object = deep_merge(obj1, obj2)
# Should print: {"a": 2, "c": 3, "b": 2}
```

## Java Solution

```java
import java.util.HashMap;
import java.util.Map;
import java.util.Set;

public class DeepMergeExample {

    // Method to check if an object is a Map (used to simulate an 'object' in Java).
    private static boolean isMap(Object item) {
        return item instanceof Map;
    }

    // Deeply merge two Maps and return the result.
    @SuppressWarnings("unchecked")
    public static Map<String, Object> deepMerge(Map<String, Object> firstMap, Map<String, Object> secondMap) {
        for (String key : secondMap.keySet()) {
            // If the key exists in the first map,
            if (firstMap.containsKey(key)) {
                Object secondValue = secondMap.get(key);

                // If the current key is present in both maps and both values are also maps, then merge them recursively.
                if (isMap(firstMap.get(key)) && isMap(secondValue)) {
                    Map<String, Object> firstMapValue = (Map<String, Object>) firstValue;
                    Map<String, Object> secondMapValue = (Map<String, Object>) secondValue;
                    firstMap.put(key, deepMerge(firstMapValue, secondMapValue));
                } else {
                    // If the second map has a key that's not in the first or the values are not maps, put it in the first map.
                    firstMap.put(key, secondValue);
                }
            }
        }

        // Return the merged Map object.
        return firstMap;
    }

    // Example usage:
    public static void main(String[] args) {
        Map<String, Object> map1 = new HashMap<>();
        map1.put("a", 1);
        map1.put("c", 3);

        Map<String, Object> map2 = new HashMap<>();
        map2.put("a", 2);
        map2.put("b", 2);

        Map<String, Object> mergedMap = deepMerge(map1, map2);

        // Should display: {a=2, b=2, c=3}
        System.out.println(mergedMap);
    }
}
```

## C++ Solution

```cpp
#include <unordered_map>
#include <vector>
#include <string>
#include <any>

// Function to check if a std::any value is an object (i.e., std::unordered_map).
bool isObject(const std::any &item) {
    if (item.type() == typeid(std::unordered_map<std::string, std::any>)) {
        return true;
    }
    return false;
}

// Function to check if a std::any value is an array (i.e., std::vector).
bool isArray(const std::any &item) {
    if (item.type() == typeid(std::vector<std::any>)) {
        return true;
    }
    return false;
}

// Function to deeply merge two objects, combining their properties and sub-properties.
std::any deepMerge(std::any firstObject, std::any secondObject) {
    if (!isObject(firstObject) || !isObject(secondObject)) {
        return secondObject;
    }

    if (isArray(firstObject) != isArray(secondObject)) {
        return secondObject;
    }

    std::unordered_map<std::string, std::any> firstMap =
        std::any_cast<std::unordered_map<std::string, std::any>>(firstObject);
    std::unordered_map<std::string, std::any> secondMap =
        std::any_cast<std::unordered_map<std::string, std::any>>(secondObject);

    for (const auto &pair : secondMap) {
        const std::string &key = pair.first;
        if (firstMap.find(key) == firstMap.end()) {
            firstMap[key] = pair.second;
        } else {
            firstMap[key] = deepMerge(firstMap[key], pair.second);
        }
    }

    return firstObject;
}

// Example usage:
// std::unordered_map<std::string, std::any> obj1 = {{"a", 1}, {"c", 3}};
// std::unordered_map<std::string, std::any> obj2 = {{"a", 2}, {"b", 2}};
// auto result = deepMerge(obj1, obj2);
// The resulting 'obj1' will be deep-merged with 'obj2'.
```

## Typescript Solution

```typescript
// Function to deeply merge two objects, combining their properties and sub-properties.
function deepMerge(firstObject: any, secondObject: any): any {
    // Helper function to check if a value is an object.
    const isObject = (item: any): boolean => item && typeof item === 'object' && !Array.isArray(item);

    // Helper function to check if a value is an array.
    const isArray = (item: any): boolean => Array.isArray(item);

    // If either argument is not an object or array, return the secondObject.
    if (!isObject(firstObject) || !isObject(secondObject)) {
        return secondObject;
    }

    // If the types of container for both objects differ (one is array and one is object), return secondObject.
    if (isArray(firstObject) !== isArray(secondObject)) {
        return secondObject;
    }

    // Iterate over each property of the secondObject.
    for (const key in secondObject) {
        // If the property key is from the object's own properties (not inherited).
        if (secondObject.hasOwnProperty(key)) {
            // Recursively merge properties from both objects.
            firstObject[key] = deepMerge(firstObject[key], secondObject[key]);
        }
    }

    // Return the merged firstObject, which now contains properties from both objects.
    return firstObject;
}

// Example usage:
// let obj1 = {"a": 1, "c": 3}, obj2 = {"a": 2, "b": 2};
// deepMerge(obj1, obj2); // Should log: {"a": 2, "c": 3, "b": 2}
```

## Time and Space Complexity

### Time Complexity

The time complexity of the `deepMerge` function is a bit tricky to calculate due to its recursive nature. It depends on the structure and size of the two input objects, `obj1` and `obj2`. For each key in `obj2`, it recursively merges the corresponding values. If $n$ is the total number of keys in both `obj1` and `obj2` at all levels, and assuming the worst case where each value is a nested object requiring a recursive merge, the time complexity would be $O(n)$. It's important to note, though, that each merge itself takes time proportional to the number of keys in the object being merged, which could imply more than just a simple $O(n)$ complexity, but for each merge operation, we consider a constant time assuming that objects/arrays being merged at the same depth are relatively small.

Note: The recursion depth also depends on the depth of the object structure, which might impact the time complexity if we account for the cost of function calls; however, this is often disregarded in favor of analyzing the operation's complexity on the data set rather than the call stack.

### Space Complexity

The space complexity is affected by two factors: the depth of the recursion (call stack) and the creation of new objects/arrays during the merge process.

1. **Recursion Depth (Call Stack):** If we consider $d$ to be the maximum depth of recursion (which is the deepest level of nested objects/arrays), the space complexity in terms of the call stack would be $O(d)$.
2. **Object Creation:** Since the function modifies `obj1` directly without creating new objects for each merge operation, the space complexity due to object creation would be $O(1)$ - constant space.

Taking both points into account, the overall space complexity of `deepMerge` would be $O(d)$, assuming that we only consider the additional space required by the recursion stack and not the space taken by the input objects, which would remain the same throughout the execution of the function.