2633. Convert Object to JSON String

Medium

Leetcode Link

Problem Description

The challenge here is to implement a function that mimics the behavior of JSON's `stringify` method, which converts a JavaScript value into a valid JSON string. You need to handle all basic JSON types: strings, numbers, booleans, arrays, objects, and null. You must ensure that when you're converting these values to a JSON string, there are no extra spaces -- the string should be as concise as possible. The order in which object keys are inserted is important, and it should reflect the order provided by the `Object.keys()` method in JavaScript. Crucially, the use of the built-in `JSON.stringify` function is not allowed for this challenge.

Intuition

The solution involves recursively converting JavaScript values to JSON strings according to the JSON format specification. The approach taken follows the principles of serialization, handling each data type according to JSON's representation rules.

- For `null`, we straightforwardly return the string "null".
- For strings, we encase them in double quotes (as JSON strings are double-quoted).
- For numbers and booleans, we simply convert them to their string representations using `toString` method.
- Arrays are trickier since they can contain elements of any type, including nested arrays and objects. We handle this by mapping over the array and applying our `jsonStringify` function to each element, then joining the results with commas and enclosing with square brackets.
- Objects require us to map over their entries (key-value pairs), stringify each key and value using our function, join these pairs with commas, and wrap the result with curly braces.
- Finally, if an object is not serializable (like a function or undefined), we'll return an empty string, although in a more robust implementation we might handle this differently (for example, by throwing an exception).

This recursive design allows the function to properly nest arrays and objects within each other, resulting in a valid JSON string of the input value.

Solution Approach

To implement the `jsonStringify` function, we use recursion and the built-in types and methods of JavaScript. Here's a closer look at how each type of value is handled:

- **Null:** If the object is `null`, we return the string literal `"null"`, as this is its JSON representation.
- **String:** We return the string enclosed in double quotes. Any string passed to the function will be surrounded by `"` characters to comply with the JSON format.
- **Number and Boolean:** Both of these data types are directly converted to their string representation using the `.toString()` method without any additional formatting.
- **Array:** Array handling is recursive. We use the `Array.map()` method which invokes the `jsonStringify` function on each element of the array. This call will handle any level of nested arrays or objects properly. The results are joined together with commas, and square brackets are added to the start and end of the string to form a proper JSON array. Therefore, the algorithm takes care of arrays within arrays, objects within arrays, and other complex structures.
- **Object:** Objects are dealt with by first transforming them into an array of `[key, value]` pairs using `Object.entries()`. Each key and value in these pairs is then passed through the `jsonStringify` function. The keys and values are concatenated with `:` to form a valid JSON object property. The resulting strings are joined with commas, and the entire string is enclosed in curly braces. Because JavaScript objects are unordered collections of properties, we do not need to sort the keys; we rely on the order provided by `Object.keys()` as per the problem description.

The constructed solution checks the type of the input value and applies the appropriate handling mechanism. The use of recursion enables the function to handle nested objects and arrays seamlessly. Each recursive call processes a smaller part of the data until the entire object is serialized into a JSON string. The function seamlessly switches between these cases without using additional data structures or complex patterns, adhering strictly to JavaScript's native representations of these data types and JSON format rules.

This custom implementation of a `jsonStringify` function is a good exercise in understanding serialization and the JSON format rules.

Example Walkthrough

Let's walk through an example to illustrate how the solution approach handles different types of JavaScript values and converts them into a valid JSON string using our custom `jsonStringify` function.

Assume we have the following JavaScript object:

```
{
  "name": "Jane",
  "age": 32,
  "isStudent": false,
  "courses": ["Math", "English"],
  "nullValue": null
}
```

Here's how `jsonStringify(person)` would process this object:

1. **Object:** Since our input is an object, we initiate the serialization process by getting all `[key, value]` pairs using `Object.entries(person)`. The algorithm will then recursively process each key and value.
2. **String (Key):** Each key, such as `"name"`, is transformed into a string by enclosing it in double quotes to become `"\"name\""`.
3. **String (Value):** The `name` property's value, `"Jane"`, is also enclosed in double quotes to become `"\"Jane\""`.
4. **Number:** The `age` property's value `32` is converted to a string by simply doing `32.toString()` to become `"32"`.
5. **Boolean:** The `isStudent` boolean value `false` is serialized directly to its string representation to become `"false"`.
6. **Array:** The `courses` array is where recursion comes into play. The array contains two strings and an object, so each element is serialized separately:
   - The strings `"Math"` and `"English"` are enclosed in quotes to become `"\"Math\""` and `"\"English\""`.
   - The object `{ courseName: "Science" }` is itself serialized recursively to become `"{\"courseName\":\"Science\"}"`.

   The elements are joined with commas to form the string representation of the array: `"[\"Math\",\"English\",{\"courseName\":\"Science\"}]"`.
7. **Null:** The `nullValue` key leads to a simple translation where `null` is directly translated to `"null"`.
8. **Combining:** Finally, the serialized key-value pairs are concatenated with a colon between them and joined with commas to form the full object, all enclosed in curly braces:

```
{"name":"Jane","age":32,"isStudent":false,"courses":["Math","English",{"courseName":"Science"}],"nullValue":null}
```

This JSON string is the output of our `jsonStringify` function when given the `person` object. It has been created without using the built-in `JSON.stringify` and follows JSON's format specifications closely for each data type.

Python Solution

```python
def json_stringify(obj):
    """
    This function takes any value and attempts to convert it to a JSON string.
    """
    # Handle the None case explicitly.
    if obj is None:
        return "null"

    # Strings need to be wrapped in quotes.
    if isinstance(obj, str):
        return f"{obj}"

    # Numbers and Booleans can be converted to string directly.
    if isinstance(obj, (int, float, bool)):
        return str(obj)

    # Lists are processed recursively, with each element being converted and joined by commas.
    if isinstance(obj, list):
        list_elements = [json_stringify(element) for element in obj]
        return '[' + ','.join(list_elements) + ']'

    # Dictionaries are processed by converting each key-value pair and joining them by commas.
    if isinstance(obj, dict):
        dict_entries = [f"{json_stringify(key)}:{json_stringify(value)}" for key, value in obj.items()]
        return '{' + ','.join(dict_entries) + '}'

    # Fallback for unsupported types; return an empty string.
    return ""
```

Java Solution

```java
import java.util.Map;

public class JsonStringify {

    /**
     * This method converts an Object to a JSON string.
     *
     * @param object The object to be converted to JSON string.
     * @return A JSON string representation of the object.
     */
    public static String jsonStringify(Object object) {
        // Explicit handling for null case
        if (object == null) {
            return "null";
        }

        // Handle String objects, wrapping them in quotes
        if (object instanceof String) {
            return "\"" + object + "\"";
        }

        // Handle Number and Boolean objects by using toString method
        if (object instanceof Number || object instanceof Boolean) {
            return object.toString();
        }

        // Handle arrays recursively. Assuming it's an array of Objects for simplicity
        if (object.getClass().isArray()) {
            Object[] array = (Object[]) object;
            StringBuilder arrayElementsStringBuilder = new StringBuilder();
            arrayElementsStringBuilder.append("[");
            for(int i = 0; i < array.length; i++) {
                arrayElementsStringBuilder.append(jsonStringify(array[i]));
                if (i < array.length - 1) arrayElementsStringBuilder.append(",");
            }
            arrayElementsStringBuilder.append("]");
            return arrayElementsStringBuilder.toString();
        }

        // Handle Map objects (used to represent objects in Java)
        if (object instanceof Map) {
            Map<?, ?> map = (Map<?, ?>) object;
            StringBuilder objectEntriesStringBuilder = new StringBuilder();
            objectEntriesStringBuilder.append("{");
            int i = 0;
            for (Map.Entry<?, ?> entry : map.entrySet()) {
                // Convert each key-value pair to JSON string
                objectEntriesStringBuilder.append(jsonStringify(entry.getKey()));
                objectEntriesStringBuilder.append(":");
                objectEntriesStringBuilder.append(jsonStringify(entry.getValue()));

                if (i < map.size() - 1) {
                    objectEntriesStringBuilder.append(",");
                }
                i++;
            }
            objectEntriesStringBuilder.append("}");
            return objectEntriesStringBuilder.toString();
        }

        // Fallback for unsupported types, returning an empty string
        return "";
    }

    // Additional main method for demonstration purposes (optional)
    public static void main(String[] args) {
        // You can test the method here by passing various types to the jsonStringify method
        // Examples:
        String jsonString = jsonStringify("Hello World!");
        System.out.println(jsonString); // Outputs: "Hello World!"
    }
}
```

C++ Solution

```cpp
#include <iostream>
#include <string>
#include <vector>
#include <unordered_map>
#include <variant>
#include <sstream>

// Define a variant type that can hold any of the acceptable JSON types.
using JsonValue = std::variant<std::monostate, std::nullptr_t, std::string, double, bool,
                               std::vector<JsonValue>, std::unordered_map<std::string, JsonValue>>;

// Forward declaration necessary for recursive calls.
std::string jsonStringify(const JsonValue& value);

// Helper function to convert a std::string to a JSON string (adds quotes around the string).
std::string jsonStringify(const std::string& value) {
    return "\"" + value + "\"";
}

// Function to convert JsonValue to a JSON string.
std::string jsonStringify(const JsonValue& value) {

    // Handle different types using std::visit and a lambda function.
    return std::visit([](auto&& arg) -> std::string {
        using T = std::decay_t<decltype(arg)>;
        if constexpr (std::is_same_v<T, std::monostate>) { // Handle null explicitly.
            return "null";
        } else if constexpr (std::is_same_v<T, std::string>) { // Handle strings explicitly.
            return jsonStringify(arg);
        } else if constexpr (std::is_same_v<T, double> || std::is_same_v<T, bool>) { // Handle numbers and booleans.
            std::ostringstream ss;
            ss << arg;
            return ss.str();
        } else if constexpr (std::is_same_v<T, std::vector<JsonValue>>) { // Handle arrays recursively.
            std::string elements;
            for (const auto& elem : arg) {
                if (!elements.empty()) elements += ",";
                elements += jsonStringify(elem);
            }
            return "[" + elements + "]";
        } else if constexpr (std::is_same_v<T, std::unordered_map<std::string, JsonValue>>) { // Handle objects.
            std::string entries;
            for (const auto& [key, val] : arg) {
                if (!entries.empty()) entries += ",";
                entries += jsonStringify(key) + ":" + jsonStringify(val);
            }
            return "{" + entries + "}";
        } else { // Fallback for nonstate (the uninitialized state of std::variant).
            return "";
        }
    }, value);
}
```

Typescript Solution

```typescript
// This function takes any value and attempts to convert it to a JSON string.
function jsonStringify(object: any): string {
  // Handle the null case explicitly.
  if (object === null) {
    return "null";
  }

  // Strings need to be wrapped in quotes.
  if (typeof object === 'string') {
    return `"${object}"`;
  }

  // Numbers and Booleans can be converted to string directly.
  if (typeof object === 'number' || typeof object === 'boolean') {
    return object.toString();
  }

  // Arrays are processed recursively, with each element being converted and joined by commas.
  if (Array.isArray(object)) {
    const arrayElementsString = object.map(jsonStringify).join(',');
    return `[${arrayElementsString}]`;
  }

  // Objects are processed by converting each key-value pair and joining them by commas.
  if (typeof object === 'object') {
    const objectEntriesString = Object.entries(object)
      .map(([key, value]) => `${jsonStringify(key)}:${jsonStringify(value)}`)
      .join(',');
    return `{${objectEntriesString}}`;
  }

  // Fallback for unsupported types: return an empty string.
  return '';
}
```

Time and Space Complexity

The given `jsonStringify` function takes an input `object` and recursively converts it into a JSON string. Analyzing the complexity depends upon the structure of the input object.

Time Complexity

The time complexity of this function is difficult to state definitively without considering the specific structure and size of the input object. However, we can describe the time complexity in terms of the input size.

1. **Primitive Types** (`null`, `string`, `number`, `boolean`): The complexity is $O(1)$ for `null`, numbers, and boolean values, and $O(n)$ for strings, where n is the length of the string.
2. **Arrays**: Time complexity would be $O(n * m)$, where n is the number of elements in the array and n is the size of the largest element (in terms of the time complexity to stringify it). This accounts for mapping over the array (n elements) and the recursive calls for each element which can vary in complexity.
3. **Objects**: The complexity would be $O(k * m)$, where k is the number of keys in the object and m is the complexity of the largest value to stringify. This involves the `Object.entries()` call and mapping over k key-value pairs, with recursion happening depending on the structure and size of value n.

Given these aspects, the overall time complexity has a recursive nature and in the worst case (deeply nested objects or large arrays), it could approach $O(n^2)$ or worse, depending on the complexity of recursion at each level.

Space Complexity

The space complexity will also depend on the input object:

1. **Primitive Types**: Space complexity is $O(1)$ for `null`, numbers, and boolean values, and $O(n)$ for strings, where n is the length of the string.
2. **Arrays**: Space complexity is $O(n * s)$, where n is the number of elements in the array and s is the space complexity to stringify the largest element, since each element's JSON string is accumulated into a new array string.
3. **Objects**: Space complexity is similar to arrays, $O(k * s)$, where k is the number of keys and s is the space needed for the largest value.

The recursive calls add to the call stack, which also increases the space complexity. For deeply nested structures, the space complexity could be significant due to the recursive stack. Therefore, we can consider the space complexity to be $O(n)$ in less complex cases, scaling to $O(n * s)$ in worse scenarios, with n accounting for depth/number of elements and s accounting for the size of elements.

Considering that JSON stringification typically involves creating new strings (immutable in JavaScript), these strings are likely to consume space linearly with respect to the size and depth of the input.