2677. Chunk Array

Easy

Leetcode Link

Problem Description

The task is to take an input array `arr` and an integer `size`, and then divide or "chunk" the array into subarrays, where each subarray has a maximum length of `size`. The resulting array of subarrays should include all the elements of the original array in the same order, just segmented into chunks. If there are not enough elements at the end of `arr` to make up a full chunk of length `size`, then the last subarray will contain fewer than `size` elements.

Keep in mind that the array `arr` can be considered as being produced by a process like `JSON.parse`, implying that it is a valid JSON array which could contain any type of elements, not just numbers. Additionally, you should not rely on external libraries such as lodash, particularly avoiding the use of a function like `_.chunk` which essentially solves this problem.

Intuition

To chunk the array without using additional libraries, we can create a new array to hold our chunks, then iterate over the original array, slicing it into smaller arrays of length `size`. Here's the process:

1. Initialize an empty array `ans` to store our chunks.
2. Loop over the original array, using a counter `i` that starts at 0 and increments by `size` each time. This way, each iteration processes a chunk of the array.
3. In each iteration, use the `slice` method to get a subarray of length `size` starting from the current index `i`.
4. Push this subarray into our `ans` array.
5. Continue the process until we've reached the end of the original array.
6. The `slice` method will automatically handle the scenario where there aren't enough elements at the end of the array to form a complete chunk, resulting in the last subarray being the correct, potentially smaller size.
7. Finally, return the `ans` array containing our chunks.

The intuition behind this approach is to systematically break down the original array into sections without needing complex logic or external libraries to assist us. The `slice` method available in TypeScript is quite handy as it easily allows extracting parts of an array based on index positions and does not modify the original array while doing so.

Solution Approach

The solution is simple in nature and relies on basic array manipulation techniques provided by TypeScript/JavaScript.

- **Algorithm:** The solution involves a single pass through the input array, slicing out subarrays and collecting them into a result array.
- **Data Structures:** Only one additional data structure is used, which is the result array that stores the chunks, referred to as `ans` in the code.
- **Patterns:** The pattern used here is a common iteration pattern, where the index is incremented not by 1 but by the chunk size on each loop iteration.

Walking through the code:

1. A result array `ans` is declared to store the chunks that will eventually be returned.

2. A for-loop is used to iterate over the elements of the input array. The loop is controlled by index `i`, which starts at 0 and is incremented by `size` after each iteration to move to the next chunk.

   ```
   for (let i = 0, n = arr.length; i < n; i += size) {
     ...
   }
   ```

3. Inside the loop body, the `slice` method is used to create a subarray consisting of elements from index `i` to `i + size`. The `slice` method is inclusive of the start index and exclusive of the end index.

   ```
   ans.push(arr.slice(i, i + size));
   ```

4. This subarray is then pushed into the `ans` array. The `slice` method will ensure that if the number of elements from index `i` is less than `size`, the subarray will include all elements up to the end of the array, capturing the potentially smaller last chunk if the array size is not perfectly divisible by `size`.

5. The loop continues until `i` is greater than or equal to the length of `arr`, signifying that all elements have been included in the chunks.

6. Once the loop is complete, `ans`, now containing all the chunked subarrays, is returned as the final result.

This approach is efficient because it only requires a single traversal of the original array and uses `slice`, which is an optimized native array method. There are no nested loops or redundant operations, making this approach ideal for chunking an array into subarrays of specified size.

Example Walkthrough

Let's use a small example to illustrate the solution approach. Suppose we have an input array `arr` represented as `[1, 2, 3, 4, 5, 6, 7]`, and we want to chunk this array into subarrays of size `3`.

Following the solution approach:

1. We declare an empty result array `ans` to which we will add our chunks.

2. We start a for-loop with a counter `i` initialized at 0, which will increment by the `size` after each iteration. Our size is `3`, so `i` will take values `0`, `3`, and `6` during the loop.

   ```
   for (let i = 0, n = arr.length; i < n; i += size) {
     ...
   }
   ```

3. On the first iteration (`i = 0`), we use the `slice` method to create a subarray from index 0 to index 3, which yields `[1, 2, 3]`.

   ```
   ans.push(arr.slice(0, 0 + 3)); // equivalent to arr.slice(0, 3)
   ```

4. Then we push `[1, 2, 3]` into the `ans` array.

5. Next iteration (`i = 3`), we use the `slice` method to create a subarray from index 3 to index 6, resulting in `[4, 5, 6]`.

   ```
   ans.push(arr.slice(3, 3 + 3)); // equivalent to arr.slice(3, 6)
   ```

6. We push `[4, 5, 6]` into the `ans` array.

7. On the final iteration (`i = 6`), we `slice` from index 6 to the end of the array, as `slice` naturally handles cases where the end index is beyond the array's length. This yields the subarray `[7]`.

   ```
   ans.push(arr.slice(6, 6 + 3)); // equivalent to arr.slice(6, 9)
   ```

8. We push the final chunk `[7]` into the `ans` array.

9. Once the loop finishes, we have `ans` containing all the chunks: `[[1, 2, 3], [4, 5, 6], [7]]`.

10. This resulting array `ans` is then returned.

By following this process, we efficiently divide the original array into chunks of a given size, demonstrating the algorithm's effective use of TypeScript/JavaScript's native array methods.

Python Solution

```python
def chunk(array, size):
    """Function to split an array into chunks of a specified size.

    Args:
    array: A list of elements that needs to be split.
    size: The size of each chunk.

    Returns:
    A list of lists where each sublist is a chunk of the input array.
    """
    chunks = []  # Initialize an empty list to hold the chunks

    # Loop through the array, incrementing by 'size' on each iteration
    for i in range(0, len(array), size):
        # Slice the array from the current index 'i' up to 'i + size'
        # and append this new chunk to the 'chunks' list
        chunks.append(array[i:i + size])

    return chunks  # Return the list of chunks
```

Java Solution

```java
import java.util.ArrayList;
import java.util.List;

// Function to split a list into chunks of a specified size
public static List<List<Object>> chunk(List<Object> list, int size) {
    List<List<Object>> chunks = new ArrayList<>(); // Initialize an empty list to hold the chunks

    // Loop through the list, incrementing by 'size' on each iteration
    for (int i = 0; i < list.size(); i += size) {
        // Calculate the end index for the current chunk, making sure it does not exceed the list size
        int end = Math.min(i + size, list.size());

        // Create a sublist from the current index 'i' up to 'end' and add this new chunk to 'chunks'
        chunks.add(new ArrayList<>(list.subList(i, end)));
    }

    return chunks; // Return the list of chunks
}
```

C++ Solution

```cpp
#include <vector>

// Function to split a vector into chunks of a specified size
std::vector<std::vector<int>> chunk(const std::vector<int>& array, int size) {
    std::vector<std::vector<int>> chunks; // Initialize an empty vector to hold the chunks

    // Loop through the vector, incrementing by 'size' on each iteration
    for (size_t i = 0; i < array.size(); i += size) {
        // 'std::begin(array)' is the starting iterator of 'array'
        // Move the start iterator 'i' positions forward
        auto start_itr = std::next(std::begin(array), i);

        // Determine 'end_itr' ensuring we don't pass the end of the array
        auto end_itr = std::next(start_itr, std::min(size, static_cast<int>(array.size()) - i));

        // Construct a vector from the range [start_itr, end_itr) and add it to 'chunks'
        chunks.emplace_back(start_itr, end_itr);
    }

    return chunks; // Return the vector of chunks
}
```

Typescript Solution

```typescript
// Function to split an array into chunks of a specified size
function chunk(array: any[], size: number): any[][] {
    const chunks: any[][] = []; // Initialize an empty array to hold the chunks

    // Loop through the array, incrementing by 'size' on each iteration
    for (let i = 0, arrayLength = array.length; i < arrayLength; i += size) {
        // Slice the array from the current index 'i' up to 'i + size'
        // and push this new chunk into the 'chunks' array
        chunks.push(array.slice(i, i + size));
    }

    return chunks; // Return the array of chunks
}
```

Time and Space Complexity

Time Complexity

The time complexity of the code is `O(n)`, where `n` is the total number of elements in the input array `arr`. The for loop iterates over the array in steps of `size`, and within each iteration, the `.slice()` method is called, which runs in `O(k)`, where `k` is the size of the chunk being created. However, since `k <= size` and the steps of the loop are proportional to `size`, the overall number of operations depends linearly on `n`, thus resulting in linear time complexity.

Space Complexity

The space complexity of the code is also `O(n)`. This is because the function creates a new array `ans` to store the chunks. In the worst case, when `size` is 1, the `ans` array will contain the same number of elements as the input array, spread across many sub-arrays, effectively duplicating all elements of the input. Therefore, the maximum space this function can take up is proportional to the number of items in the input array, hence `O(n)`.