1507. Reformat Date

Easy String

Leetcode Link

Problem Description

The given problem requires us to take a string that represents a date in the format of `Day Month Year` and convert it into the standard ISO format `YYYY-MM-DD`. The `Day` part of the date is represented by ordinal numbers (like `"1st"`, `"2nd"`, `"3rd"`, and so on up to `"31st"`). The `Month` is given by its three-letter abbreviation (for example, `"Jan"`, `"Feb"`, `"Mar"`, etc.), and the `Year` is a four-digit number ranging from 1900 to 2100. The task is to reformat the date string so that the `Year` is followed by the `Month` and `Day`, where the month and day are each displayed as two digits, with leading zeroes if necessary.

Intuition

The intuition behind the solution is to break down the original `date` string into its three components (`Day`, `Month`, and `Year`), then reassemble the date in the required `YYYY-MM-DD` format. To accomplish this, follow these steps:

1. Split the original date string into its components by using the space character as a delimiter.
2. Reverse the order of the components so that the `Year` comes first, followed by the `Month`, then the `Day`.
3. Map the `Month` from its abbreviation to its corresponding month number. This is done by creating a string that contains the abbreviations in order and finding the index of each month in this string. Since each abbreviation is three characters long, divide the index by 3 and add 1 to get the month number.
4. Format the `Month` so that it is always displayed as two digits by padding with a leading zero if necessary.
5. Remove the ordinal suffix (e.g., `"st"`, `"nd"`, `"rd"`, `"th"`) from the `Day` part and also ensure it is always two digits, adding a leading zero if needed.
6. Join the three components with hyphens to form the final standardized date string.

By following these logical steps, we convert the date from its given verbose form into a standardized format that is widely accepted and easy to understand programmatically.

Solution Approach

The solution is implemented in Python and follows a direct approach, leveraging Python's list and string manipulation capabilities. Here's a step-by-step explanation of the implementation:

1. The input date string is split into its different components (Day, Month, Year) using the `split` method, which uses whitespace as the default separator.

   ```
   1  s = date.split()
   ```

2. Reverse the list `s` so that `Year` becomes the first element, followed by `Month` and `Day`.

   ```
   1  s.reverse()
   ```

3. Create a string `months` that contains all the month abbreviations concatenated together. This acts like a lookup table to easily map each month abbreviation to its numeric value.

   ```
   1  months = "JanFebMarAprMayJunJulAugSepOctNovDec"
   ```

4. Calculate the numeric representation of the month. We find the index of the month in the `months` string using `index`, divide it by 3 (since each month abbreviation consists of 3 characters), and add 1 because indexing starts at 1 in our `months` string.

   ```
   1  s[1] = str(months.index(s[1]) // 3 + 1).zfill(2)
   ```

   Use `zfill(2)` to make sure the result always has two digits, padding with a zero if necessary.

5. The day component still contains the ordinal suffix, which we strip off by slicing the string excluding the last two characters (the ordinal part), and then use `zfill(2)` to ensure the day is also two digits.

   ```
   1  s[2] = s[2][:-2].zfill(2)
   ```

6. Finally, we join the components of the date together with hyphens to form the required `YYYY-MM-DD` format, and return the result.

   ```
   1  return "-".join(s)
   ```

This implementation does not use any complex algorithms or data structures; it's mainly targeted towards the utilization of basic string operations and list manipulation to format the date string correctly. The approach is simple, efficient, and does not require any additional libraries or resources beyond standard Python capabilities.

Example Walkthrough

Let's take an example date string: `"21st Jan 2023"`. Our goal is to convert this to the ISO format `YYYY-MM-DD`.

Following the steps outlined in the solution approach:

1. Split the string into components:

   ```
   1  s = "21st Jan 2023".split()
   2  # s now contains ["21st", "Jan", "2023"]
   ```

2. Reverse the list so that the `Year` comes first:

   ```
   1  s.reverse()
   2  # s now contains ["2023", "Jan", "21st"]
   ```

3. Create a string `months` to help us map the `Month` abbreviation to a number:

   ```
   1  months = "JanFebMarAprMayJunJulAugSepOctNovDec"
   2  # This helps us find the numeric representation of "Jan"
   ```

4. Convert the `Month` from abbreviation (`Jan`) to number (`01`):

   ```
   1  s[1] = str(months.index(s[1]) // 3).zfill(2)
   2  # Index of "Jan" in the `months` string is 4, so the month number is (4 // 3) + 1 = 02
   ```

5. Remove the ordinal suffix from the `Day` and add a leading zero if necessary:

   ```
   1  s[2] = s[2][:-2].zfill(2)
   2  # "21st" becomes "21", and since it's already two digits, no leading zero is added
   ```

6. Join the components with hyphens to get the final date string in ISO format:

   ```
   1  iso_date = "-".join(s)
   2  # iso_date is "2023-01-21"
   ```

The original date string of `"21st Jan 2023"` has now been successfully converted to `"2023-01-21"` using the described approach. This process can be applied to any date string in the given format to achieve the desired ISO standard date format.

Python Solution

```
class Solution:
    def reformatDate(self, date: str) -> str:
        # Split the date string into a list (e.g., '20th Oct 2052' -> ['20th', 'Oct', '2052'])
        date_components = date.split()

        # Reverse the list to start with the year (e.g., ['2052', 'Oct', '20th'])
        date_components.reverse()

        # Define a string with all months abbreviated and prefixed with a space for indexing purposes
        months_string = " JanFebMarAprMayJunJulAugSepOctNovDec"

        # Find the index of the month in the months string and convert it to a string with leading zero if necessary
        # Using `// 3` because each month is represented by three characters and we want to start at 1 for January
        date_components[1] = str(months_string.index(date_components[1]) // 3).zfill(2)

        # Remove the 'st', 'nd', 'rd', 'th' from the day part and add a leading zero if necessary
        date_components[2] = date_components[2][:-2].zfill(2)

        # Join the components with hyphens to form the reformatted date string (e.g., '2052-10-20')
        return "-".join(date_components)

# Example usage:
# sol = Solution()
# formatted_date = sol.reformatDate("20th Oct 2052")
# print(formatted_date)  # Output: "2052-10-20"
```

Java Solution

```
class Solution {
    public String reformatDate(String date) {
        // Split the input date string into an array of strings
        String[] parts = date.split(" ");

        // String containing abbreviations of months for easy lookup
        String months = " JanFebMarAprMayJunJulAugSepOctNovDec";

        // Extract the day and remove the ordinal suffix (st, nd, rd, th)
        int day = Integer.parseInt(parts[0].substring(0, parts[0].length() - 2));

        // Calculate the month by finding the index of the month abbreviation in the months string
        // Divide by 3 because each month abbreviation consists of three characters
        // And add 1 because month index should start from 1 instead of 0
        int month = months.indexOf(parts[1]) / 3;

        // Reassemble the date in the format "YYYY-MM-DD"
        // Use String.format to ensure leading zeros where necessary
        return String.format("%s-%02d-%02d", parts[2], month, day);
    }
}
```

C++ Solution

```
#include <iostream>
#include <sstream>
#include <string>
using namespace std;

class Solution {
public:
    // Function to reformat a date string from "20th Month YYYY" format to "YYYY-MM-DD" format
    string reformatDate(string date) {
        // A string containing abbreviations of all months in order for easy indexing
        string monthsStr = " JanFebMarAprMayJunJulAugSepOctNovDec";

        // Stringstream to parse the input date string
        stringstream ss(date);

        string year;     // To hold the year as a string
        string temp;     // Temporary string to hold the "th", "nd", "st" suffixes in date
        string month;    // To hold the month as a string
        int day;         // To store the day as an integer

        // Read and parse the date string
        ss >> day >> temp >> month >> year;

        // Find the starting position of the month in the monthsStr
        // Divide the index by 3 as there are 3 characters per month and then add 1 to get the numerical month
        month = to_string(monthsStr.find(month) / 3);

        // Add leading zero if needed to the month
        string formattedMonth = (month.size() == 1 ? "0" + month : month);

        // Add leading zero to the day if it is less than 10
        string formattedDay = (day > 9 ? "" : "0") + to_string(day);

        // Return the reformatted date string in "YYYY-MM-DD" format
        return year + "-" + formattedMonth + "-" + formattedDay;
    }
};
```

Typescript Solution

```
function reformatDate(date: string): string {
    // Split the input date string into an array
    const dateParts = date.split(' ');

    // Define a string of month abbreviations for index lookup
    const monthAbbreviations = ' JanFebMarAprMayJunJulAugSepOctNovDec';

    // Extract the day from the `dateParts` array and parse it as an integer
    // We remove the last two characters ('th', 'nd', 'st', 'rd') before parsing
    const day = parseInt(dateParts[0].substring(0, dateParts[0].length - 2));

    // Find the position of the month abbreviation in the `monthAbbreviations` string.
    // Divide by 3 because each abbreviation is 3 characters long, and add 1 (since index starts at " Jan")
    const month = Math.floor(monthAbbreviations.indexOf(dateParts[1]) / 3);

    // Reform the date in the YYYY-MM-DD format
    // Use `padStart` to ensure day and month are two digits
    return `${dateParts[2]}-${month.toString().padStart(2, '0')}-${day.toString().padStart(2, '0')}`;
}
```

Time and Space Complexity

Time Complexity

The time complexity of the code primarily involves splitting the input string, reversing the split parts, indexing into a string, and joining the parts back into a formatted string.

1. `split()`: The split operation is performed once on the input string. If `n` is the size (length) of the input string, split() would generally have a time complexity of `O(n)` as it goes through the entire string once to split based on the spaces.

2. `reverse()`: Reversing the list of split parts happens in `O(k)` time, where `k` is the number of elements in the list after splitting, which is always 3 in this case given the date format, so we consider this `O(1)`.

3. `index()`: Indexing into a string to find the position of a substring, such as finding the month in the `months` string. In the worst case, this could be `O(m)`, where `m` is the length of the `months` string, but since `m` is a constant (it's always 36 in this code), we can consider this operation `O(1)`.

4. `zfill()`: The `zfill()` operation is `O(d)` where `d` is the max length of the string being filled. Here, `d` is constant 2, so the complexity is `O(1)`.

5. `join()`: The join operation complexity is `O(n)`, based on the total length of all strings being joined, which is in this case the length of the output string, which we presume is also proportional to `n`.

Considering these operations and knowing that some are constant time, we can approximate the overall time complexity as `O(n)` as the `split()` and `join()` operations dominate the overall time.

Space Complexity

For space complexity, the main concern is any additional space aside from the input that we need to allocate for processing.

1. Split list `s`: This will take `O(k)` space where `k` is the number of parts after splitting, which for a date string is always 3, so this is `O(1)`.

2. Months string: The space used by the months string is a constant `O(1)` since it does not grow with the input.

3. Temporary storage for transformation, such as when creating strings for `zfill()` and when using `join()`. These are proportional to the size of the output which will be a constant length string, so this is also considered `O(1)` space.

Hence, the overall additional space used by the algorithm is constant, or `O(1)`.