Username

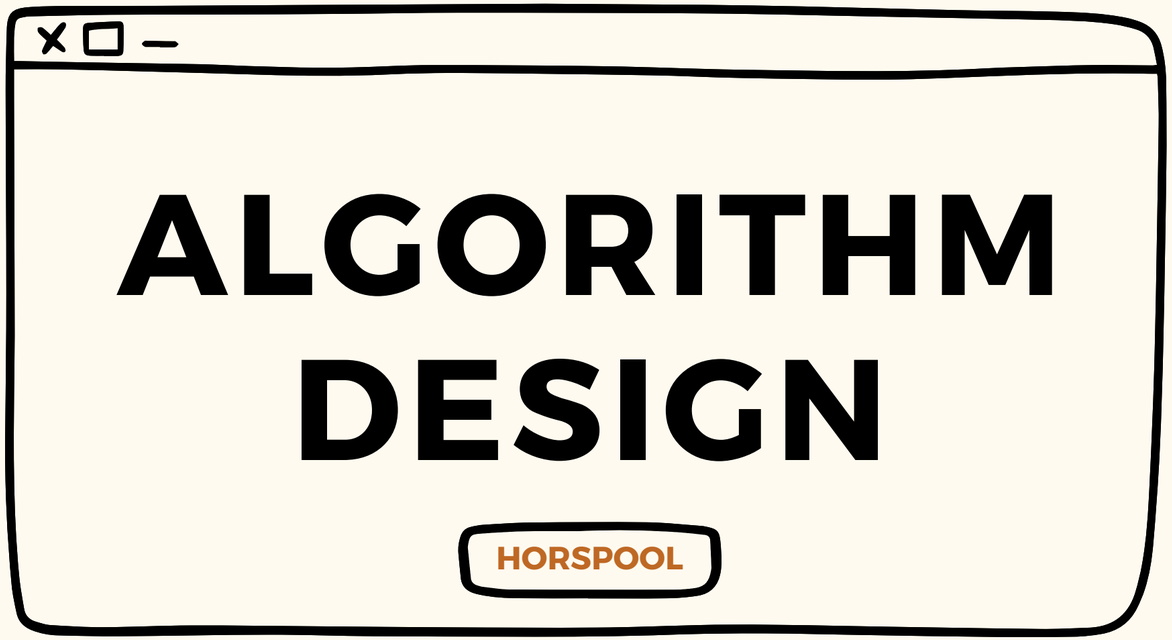
AVAILABILITY CHECKER

By Karl Apolonio

Midterm Exam

DESCRIPTION OF THE PROBLEM

This code addresses the problem of checking the availability of usernames in a list using the Horspool algorithm. The horspool function creates a shift table based on the pattern provided and searches for occurrences of the pattern in the text. The checkAvailability function uses this method to assess whether a username entered by the user is already in the predefined list of usernames, and provides feedback on its availability via a web interface.



Preprocessing Step

Construct a "shift table" that determines how far the pattern can be shifted based on the character encountered in the text.

Searching Step

- Start matching the pattern against the text from right to left.
- Shift the pattern according to the shift table based on the character encountered in the text.
- If a mismatch occurs, shift the pattern by the maximum possible distance using the shift table.

Code

IMPLEMENTATION

```
const usernames = ['user1', 'user2', 'user3', 'user4', 'horspool'];

function horspool(text, pattern) {
    const table = {};

    for (let i = 0; i < pattern.length - 1; i++) {
        table[pattern[i]] = pattern.length - 1 - i;
    }

    let i = pattern.length - 1;
    while (i < text.length) {
        let k = 0;
        while (k < pattern.length && pattern[pattern.length - 1 - k] === text[i - k]) {
            k++;
        }

        if (k === pattern.length) {
            return i - pattern.length + 1;
        } else {
            i += table[text[i]] || pattern.length;
        }
    }

    return -1;
}
```

Code

IMPLEMENTATION

```
function checkAvailability() {
    const usernameInput = document.getElementById('username');
    const username = usernameInput.value.trim();

    if (username === '') {
        document.getElementById('availability').textContent = 'Please enter a username';
        return;
    }

    const index = horspool(usernames.join(','), username);
    if (index !== -1) {
        document.getElementById('availability').textContent = `Username '${username}' is not available`;
    } else {
        document.getElementById('availability').textContent = `Username '${username}' is available`;
    }
}
```

Explanation

OF CODE

01

Preprocessing (Building the Shift Table):

In the horspool function, the shift table is constructed using a hash table (table) where each character of the pattern except the last one is mapped to the maximum shift value. The shift value represents how far the pattern can be shifted when a mismatch occurs.

02

Searching:

The horspool function performs the search. It starts searching from the end of the pattern in the text. If a mismatch occurs, it shifts the pattern according to the shift table, or shifts by the length of the pattern if the character is not present in the pattern.

Explanation

OF CODE

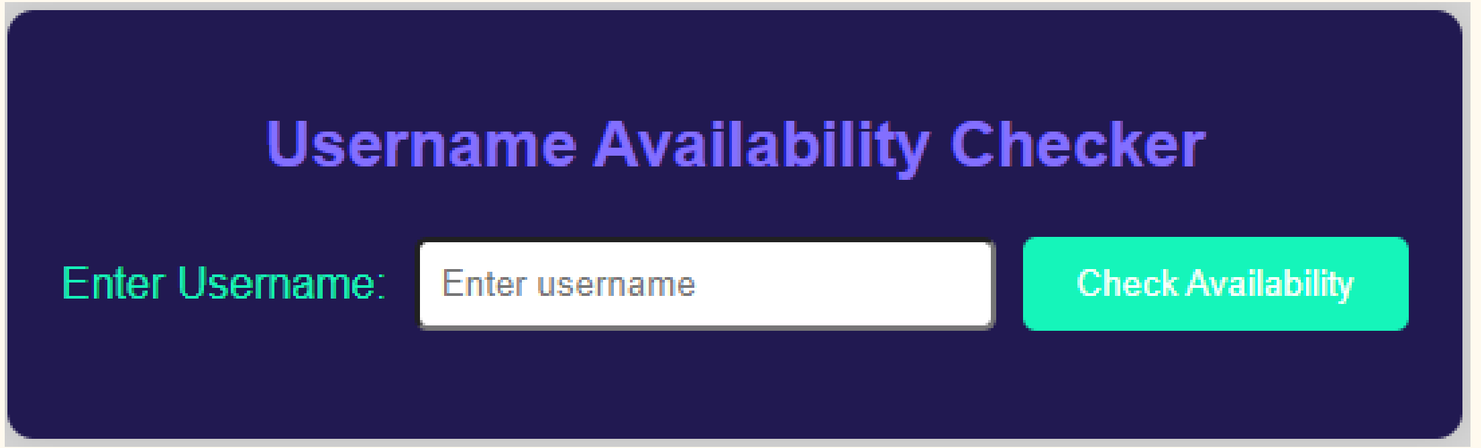
03

Checking Availability

The checkAvailability function takes input from a form field and checks if the entered username is available by calling the horspool function to search for it in the usernames array.

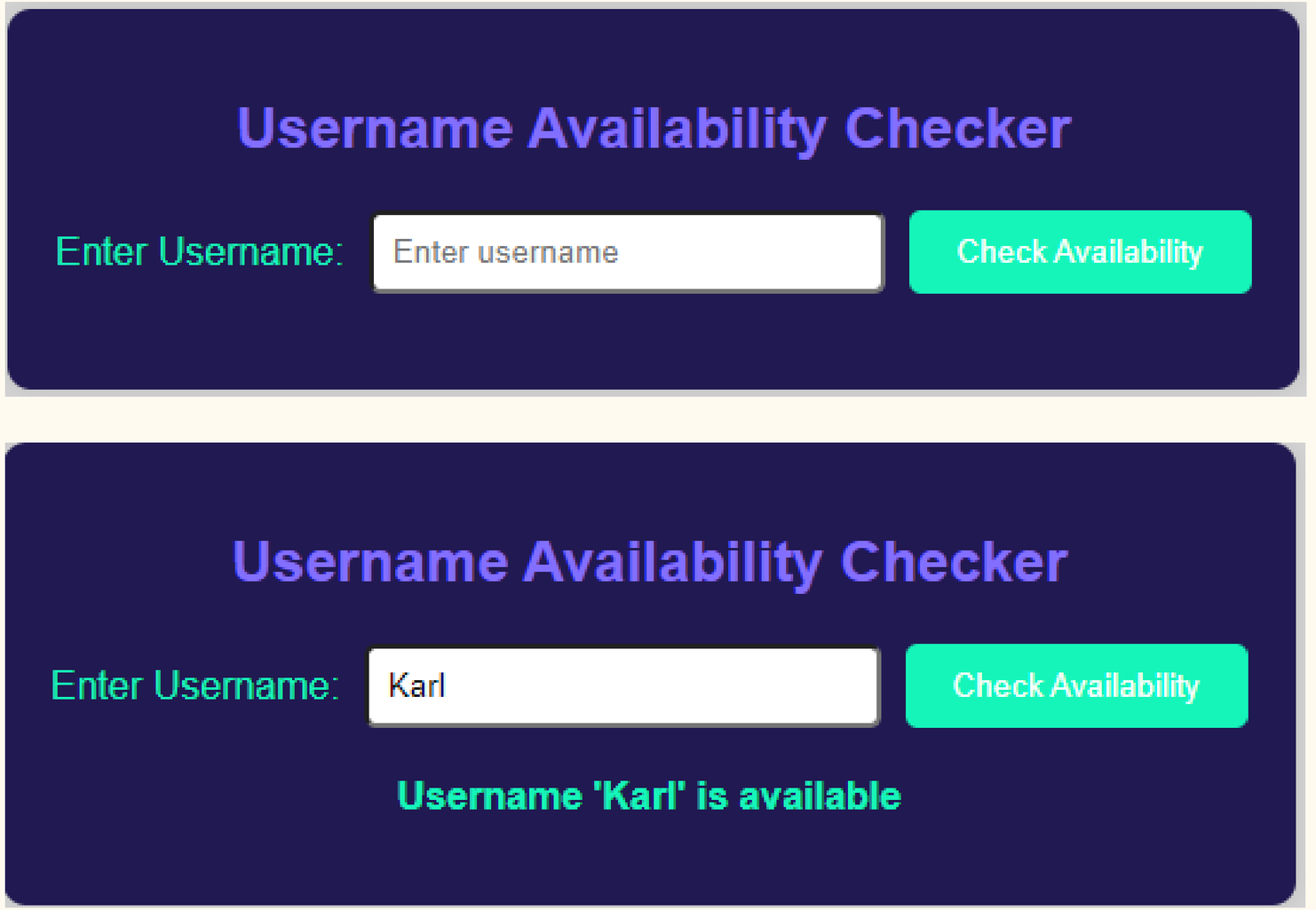
Screenshot

CODE OUTPUT



Screenshot

CODE OUTPUT



TIME COMPLEXITY

Preprocessing:

- Constructing the shift table takes O(m) time, where m is the length of the pattern.

Searching:

- In the worst case, the searching step takes O(n/m) time, where n is the length of the text. This is because the pattern can be shifted by a maximum of m positions for each comparison, and there are n/m possible comparisons. However, in practice, it can offer better performance than traditional O(n*m) algorithms like brute force.

Overall:

- The overall time complexity is O(m + n/m), where m is the length of the pattern and n is the length of the text. This represents a space-time trade-off where preprocessing the pattern allows for faster searching.

LEARNING REFLECTION

During the midterm period, I studied the topic of Space and Time Tradeoffs in algorithm design. This included looking into algorithms that make strategic trade-offs between memory use and runtime efficiency in order to improve overall performance. Understanding these trade-offs was previously difficult, particularly determining whether to prioritize memory conservation above computational performance and vice versa. However, through practice and trial, I came to understand how carefully balancing these aspects might result in more efficient algorithms. Key takeaways include understanding the need of taking into account both space and temporal difficulties when designing and implementing algorithms, as well as selecting the best trade-offs based on individual problem requirements and limitations.