**STACK**

1. [https://leetcode.com/problems/**next-greater-element-i**/](https://leetcode.com/problems/next-greater-element-i/)

**A screenshot of a computer program

Description automatically generated**

**#Approach**

To solve the problem, I used a stack and a dictionary. The stack helps keep track of elements for which I need to find the next greater element. I traversed `nums2`, and for each number, I checked if it is greater than the number at the top of the stack. If it is, this means that number is the next greater element for the top element of the stack. I then popped the stack and updated the dictionary with this mapping. I continued this process until the stack was empty or the current number was not greater than the stack's top. Then, I pushed the current number onto the stack. After processing all numbers in `nums2`, the elements left in the stack did not have a next greater element, so I mapped them to -1. Finally, I built the result list for `nums1` by looking up the next greater elements in the dictionary.

**#Challenges Faced**

The main challenge was finding an efficient solution that operates in linear time. Using the stack helped achieve this by ensuring that each element is pushed and popped from the stack at most once, giving an O(n) time complexity. Another challenge was ensuring that all elements were correctly mapped to their next greater elements, and handling cases where there is no greater element by mapping them to -1. This required careful management of the stack and dictionary to ensure correct mappings.

2. [https://leetcode.com/problems/**baseball-game**/](https://leetcode.com/problems/baseball-game/)

A screenshot of a computer program

Description automatically generated**#Approach**

I used a list to keep track of the scores as I processed each operation. For each operation, if the operation was an integer, I converted it to an integer and added it to the list. If the operation was '+', I added the sum of the last two scores in the list. If the operation was 'D', I added double the last score in the list. If the operation was 'C', I removed the last score from the list. Finally, I returned the sum of all scores in the list.

**# Challenges Faced**

Just a small challenge which is handling different types of operations correctly. I needed to ensure that the operations '+', 'D', and 'C' were handled properly. For '+', I had to ensure there were at least two previous scores to sum. For 'D', I had to ensure there was at least one previous score to double. For 'C', I had to ensure there was a score to remove. This was straightforward given the problem constraints.

3.[https://leetcode.com/problems/**score-of-parentheses**/](https://leetcode.com/problems/score-of-parentheses/)

A screenshot of a computer program

Description automatically generated

**#Approach**

I used a stack to keep track of scores at each level of nested parentheses. When I encountered an '(', I pushed a 0 onto the stack to represent a new score level. Upon encountering a ')', I popped the last score from the stack. If this score was 0, it indicated a "()", which should contribute a score of 1. If the score was not 0, it indicated a nested structure, and the score should be doubled. I then added this calculated score to the previous level's score on the stack. Finally, the result was the value left on the stack.

**#Challenges Faced**

Handling the nesting and calculating the score based on the type of structure ("()" or nested) correctly was the main challenge. I had to ensure the stack operations managed the scores at different levels of nested parentheses accurately. Avoiding mistakes in the score calculation logic, especially differentiating between direct and nested scores, was crucial to ensure the correctness of the solution.

4. <https://leetcode.com/problems/final-prices-with-a-special-discount-in-a-shop/>

A screenshot of a computer

Description automatically generated

**#Approach**

I initialized an empty stack to keep track of indices from the prices array. As I iterated through each price, I managed discounts by checking the stack. If the current price was less than or equal to the price at the index on top of the stack, I popped that index and updated its price by subtracting the current price. After applying these discounts, I pushed the current index onto the stack. This approach allowed me to efficiently handle and apply discounts in a single pass through the list. Finally, I returned the updated list of prices.

5. <https://leetcode.com/problems/valid-parentheses/>

A screenshot of a computer

Description automatically generated

To determine if a given string of brackets is valid, I used a stack data structure to track opening brackets and verify if they match with corresponding closing brackets.

**# Approach**

I began by initializing an empty stack. As I traversed the input string character by character, I pushed opening brackets (i.e., '(', '{', '[') onto the stack. For closing brackets (i.e., ')', '}', ']'), I checked the stack: if it was empty, it indicated an unmatched closing bracket, so I returned false. If the stack was not empty, I popped the top element and checked if it matched the current closing bracket; a mismatch meant the brackets were not valid, so I returned false. After processing the entire string, if the stack was empty, all opening brackets had matching closing brackets, so I returned true. Otherwise, if the stack was not empty, it meant some opening brackets had no matching closing brackets, so I returned false.

**#Complexity**

The time complexity of the solution is O(n), where n is the length of the input string, as each character is processed once with constant time operations. The space complexity is also O(n), as in the worst case, all characters could be opening brackets, requiring the stack to store them all.

**QUEUE**

1. [**https://leetcode.com/problems/first-unique-character-in-a-string/**](https://leetcode.com/problems/first-unique-character-in-a-string/)

**A screenshot of a computer

Description automatically generated**

**#Approach**

To solve the problem, I first counted the occurrences of each character by traversing the string and using a dictionary. This helped me identify characters that appeared only once. I then traversed the string again, checking the count of each character in the dictionary. The first character with a count of 1 was identified as the first unique character.

**#Challenges Faced**

The solution involves two passes through the string, resulting in a time complexity of O(n), where n is the length of the string. This approach is efficient and suitable for large strings given the problem's constraints. The space complexity is O(1) with respect to the alphabet size, as I only needed to store counts for a fixed number of lowercase English letters (26 in total). This ensured that the solution was both time-efficient and space-efficient, adhering to the problem constraints.

1. <https://leetcode.com/problems/implement-queue-using-stacks/>

A screenshot of a computer program

Description automatically generated

**#Approach**

To implement a queue using two stacks while maintaining FIFO (First In, First Out) behavior, I used two stacks, s1 and s2, where s1 holds the elements in the correct queue order, and s2 is used temporarily to reverse the order of elements during the push operation. When pushing a new element, I first moved all the existing elements from s1 to s2 to clear s1. Then, I added the new element to s1, ensuring it is placed at the end of the queue. After that, I moved all the elements back from s2 to s1, which restores their original order but with the new element at the end. This way, the queue order is maintained correctly. For the pop and peek operations, I simply interacted with the top of s1 since it already holds the elements in the correct order. This method is straightforward but results in an O(n) time complexity for the push operation due to the reordering of elements.

3. <https://leetcode.com/problems/number-of-recent-calls/description/>

A screenshot of a computer

Description automatically generated

The goal of the RecentCounter class is to count the number of requests that occurred in the last 3000 milliseconds (ms). Each time I make a request using the ping method, I record the time and maintain a count of all requests within that 3000 ms window. To do this efficiently, I track only the relevant requests, which allows me to quickly compute the result.

For my approach, I use an array called records to store the timestamps of requests, which functions similarly to a queue. I manage two indices, start and end, which mark the window of valid requests within the last 3000 ms. When I call the ping(int t) method, I first add the current timestamp t at the end position in the array. Then, I increment the start index until all requests within the window satisfy the condition t - records[start] <= 3000, effectively removing outdated requests. Finally, I return the count of valid requests in the window, which is given by end - start.

In terms of complexity, the time complexity of this approach is O(n), where n is the number of requests, as I may need to process each request individually. The space complexity is O(1) since I only maintain the necessary indices and store timestamps in a fixed-size window.

3.<https://leetcode.com/problems/number-of-students-unable-to-eat-lunch>

A screenshot of a computer

Description automatically generated

#Approach

I approached this problem by efficiently separating and handling students based on their sandwich preferences using two deques: one for students who prefer circular sandwiches and another for those who prefer square sandwiches. First, I looped through the list of students, placing them into either the `circular\_students` deque or the `square\_students` deque according to their preferences. This allowed me to create two distinct queues, each representing the students who prefer a specific type of sandwich.

Next, I processed the sandwiches stack by checking the type of sandwich at the top. If it was circular, I looked to see if there were any students in the `circular\_students` queue. If there were, the first student in line took the sandwich, and I removed them from the queue using `popleft`. If the sandwich was square, I followed the same process with the `square\_students` queue. However, if at any point the preferred queue was empty—meaning there were no students left who wanted the current type of sandwich—I stopped the process because it meant no more students could eat.

At last, I calculated the number of students who were left without a sandwich by summing the lengths of the two deques. This approach worked well because it kept the students organized by their preferences and ensured that the process of distributing sandwiches was as efficient and accurate as possible.

5. <https://leetcode.com/problems/time-needed-to-buy-tickets/>

A screenshot of a computer

Description automatically generated

My approach simulates the process of distributing tickets in a circular manner until the ticket count at the target index `k` reaches zero. I use a while loop that continuously iterates until the ticket at index `k` is fully decremented. Inside the loop, a for loop goes through each ticket in the `tickets` list. If a ticket is greater than zero, I decrement it by one, and I increment the counter `c` to keep track of the number of steps taken.

The process stops when the ticket count at `k` becomes zero, and I return the total count `c`, which represents the total time it took for the person at index `k` to get their tickets fully processed. This method efficiently handles the decrements and counts each step until the target is met.