# 682. Baseball Game

Easy Stack Array Simulation

## Problem Description

In this problem, we are given a set of operations that simulate the scoring of a baseball game, but with unique rules. A score record starts empty, and the operations are applied in sequence as given in the list of strings called `operations`. Each element in this list `operations[i]` represents an action to modify the score record. The operations include:

1. A positive or negative integer `x` which means adding a new score of `x` to the record.
2. A `'+'` which means adding a new score that is the sum of the two most recent scores.
3. A `'D'` which signifies that you need to double the latest recorded score and add this new value to the record.
4. A `'C'` which means removing the last score from the record, effectively invalidating the last operation.

The primary goal is to calculate the sum of all the scores on the record after all the operations have been applied.

The inputs are designed such that the final result, as well as any intermediate results, will fit within a 32-bit integer, and that all operations can be applied without error (operations are valid).

## Intuition

To solve this problem, we can use a stack, which is perfectly suited for situations where we need to keep track of a list of elements and frequently add or remove items from the end.

1. Iterate through each operation in the `operations` list.
2. If the operation is a positive or negative integer, convert it to an integer and push it onto the stack.
3. If the operation is a `'+'`, calculate the sum of the last two scores in the stack and push the result back onto the stack.
4. If the operation is a `'D'`, double the last score in the stack (using the left shift operation `<< 1` for efficiency, which is equivalent to multiplying by 2) and push the result onto the stack.
5. If the operation is a `'C'`, pop the last score off the stack to remove it from the record.
6. After processing all operations, the stack will contain all the valid scores. Summing these scores will give us the final result required.

The key to this solution is realizing that the last score is always at the top of the stack, and previous scores are below it, which aligns perfectly with the operation requirements of the problem. Applying each operation modifies the top portion of the stack, and in the end, we only need to sum the elements present in the stack to get the total score.

## Solution Approach

The implementation of the solution employs a stack to manage the operations on the scores effectively. Here is a step-by-step breakdown of how the algorithm works:

1. Initialize an empty stack `stk`, which will be used to keep track of the scores as they are recorded and modified.
2. Loop through each `op` in the `ops` list, which contains all the operations that need to be applied to the record in sequence.
3. Within the loop, determine the type of operation:
   - If `op` is `'+'`, we need to add the last two scores. Since the stack is LIFO (Last In, First Out), the last two scores are `stk[-1]` and `stk[-2]`. We sum these two and push the result back onto the stack with `stk.append(stk[-1] + stk[-2])`.
   - If `op` is `'D'`, we need to double the last score. Doubling can be efficiently done by using the bitwise left shift operation `<< 1` (which is similar to multiplying the last score by 2). The result is then pushed onto the stack with `stk.append(stk[-1] << 1)`.
   - If `op` is `'C'`, we need to remove the last score from the record. Popping from the stack with `stk.pop()` removes the last element.
   - If `op` is neither of these special characters, it is an integer in string format. Convert it to an integer with `int(op)` and push it onto the stack with `stk.append(int(op))`.
4. Once all the operations are applied, the scores that remain on the stack represent the valid scores after following all the operations.
5. The sum of the entire stack is calculated using the `sum()` function, which adds up all the integers in the stack. This sum is then returned as the final result of the function `sum(stk)`.

This solution approach relies on the ability of the stack to efficiently manage elements where the most recent elements need to be accessed or modified. It makes use of the fact that each operation only affects the most recent scores, which are always at the top of the stack, making the rest of the list irrelevant for that specific operation. By pushing the results of operations onto the stack and popping when necessary, the algorithm correctly simulates the scoring system of the game using a last-in, first-out structure.

The Python implementation is straightforward and makes use of the native list data structure in Python, which can be used as a stack with its append and pop methods.

### Example Walkthrough

Let's consider a sample round of the baseball game with the following series of operations:

```
operations = ["5", "-2", "4", "C", "D", "9", "+", "+"]
```

Following is the walk-through of how the scoring will be updated step-by-step, applying the solution approach:

1. The stack is initially empty: `stk = []`
2. Process the first operation `"5"`. Since `"5"` is an integer (not a special character), push it onto the stack:
   - `stk = [5]`
3. Next operation is `"-2"`. It's an integer, so push it onto the stack:
   - `stk = [5, -2]`
4. The operation `"4"` is also an integer. Append it to the stack:
   - `stk = [5, -2, 4]`
5. The operation `"C"` indicates that we should remove the last score. Pop the last element from the stack:
   - `stk = [5, -2]`
6. The operation `"D"` requires doubling the last score and pushing the result onto the stack. Double the last score (`-2`) to get `-4` and push it onto the stack:
   - `stk = [5, -2, -4]`
7. The operation `"9"` is an integer. Append it to the stack:
   - `stk = [5, -2, -4, 9]`
8. For the operation `"+"`, add the last two scores (`9 + (-4) = 5`) and push the result onto the stack:
   - `stk = [5, -2, -4, 9, 5]`
9. Finally, apply another `"+"` operation, add the last two scores again (`5 + 9 = 14`) and push the result onto the stack:
   - `stk = [5, -2, -4, 9, 5, 14]`
10. With all operations processed, we now sum the elements of the stack to obtain the final result.
    - The final score: `5 - 2 - 4 + 9 + 5 + 14 = 27`

Thus, the sum of the scores on the stack is 27, which would be the output of the solution for these operations.

The walk-through vividly demonstrates the use of a stack to manage the scores throughout the operation sequence, complying with the rules of the special baseball game. Each operation is applied in turn, modifying the state of the stack until all operations have been processed, at which point the sum of the stack represents the final score.

## Solution Implementation

### Python

```python
# Class to define the solution for the problem.
class Solution:
    # Function to calculate the final score using a list of operations.
    def calPoints(self, ops: List[str]) -> int:
        # Initialize a stack to keep track of the scores.
        scores_stack = []

        # Loop over each operation in the list of operations.
        for op in ops:
            # If operation is "+", add the sum of the last two scores to the stack.
            if op == '+':
                scores_stack.append(scores_stack[-1] + scores_stack[-2])
            # If operation is "D", double the last score and add to the stack.
            elif op == 'D':
                scores_stack.append(scores_stack[-1] * 2)
            # If operation is "C", remove the last score from the stack.
            elif op == 'C':
                scores_stack.pop()
            # Otherwise, the operation is a number, so parse and add to the stack.
            else:
                scores_stack.append(int(op))

        # Return the sum of the scores on the stack, which is the final score.
        return sum(scores_stack)
```

### Java

```java
class Solution {
    public int calPoints(String[] ops) {
        // Create a deque to use as a stack to keep track of the points.
        Deque<Integer> stack = new ArrayDeque<>();

        // Loop through the operations.
        for (String op : ops) {
            switch (op) {
                case "+": // If the operation is "+", add the sum of the last two scores.
                    int last = stack.pop(); // Remove the last score from the stack.
                    int newTop = stack.peek(); // Peek at the new top without removing it.
                    stack.push(last); // Push the last score back onto the stack.
                    stack.push(last + newTop); // Push the sum of last two scores onto the stack.
                    break;
                case "D": // If the operation is "D", double the last score.
                    stack.push(stack.peek() * 2); // Peek the last score, double it, and push onto the stack.
                    break;
                case "C": // If the operation is "C", remove the last score.
                    stack.pop(); // Remove the last score from the stack.
                    break;
                default: // For any number, parse it and put onto the stack.
                    stack.push(Integer.parseInt(op)); // Parse the string to an integer and push it onto the stack.
                    break;
            }
        }

        // Sum up and return the points in the stack.
        int sum = 0;
        for (int score : stack) {
            sum += score; // Accumulate the scores.
        }

        return sum; // Return the final sum of the scores.
    }
}
```

### C++

```cpp
#include <vector>
#include <string>
#include <numeric>

class Solution {
public:
    int calPoints(vector<string>& operations) {
        vector<int> recordStack; // Stack to maintain the record of points

        // Iterate through each operation
        for (const auto& operation : operations) {
            int currentSize = recordStack.size(); // Current size of the record stack

            // If operation is "+", push the sum of the last two scores onto the recordStack
            if (operation == "+") {
                int lastScore = recordStack[currentSize - 1];
                int secondLastScore = recordStack[currentSize - 2];
                recordStack.push_back(lastScore + secondLastScore);
            }
            // If operation is "D", double the last score and push onto the recordStack
            else if (operation == "D") {
                recordStack.push_back(recordStack[currentSize - 1] * 2);
            }
            // If operation is "C", remove the last score from the recordStack
            else if (operation == "C") {
                recordStack.pop_back();
            }
            // Otherwise, operation is assumed to be a number, convert to int and push onto the recordStack
            else {
                recordStack.push_back(stoi(operation));
            }
        }

        // Sum up all the scores in the recordStack to get the total points
        return accumulate(recordStack.begin(), recordStack.end(), 0);
    }
};
```

### TypeScript

```typescript
function calculatePoints(operations: string[]): number {
    // Initialize a stack to store the valid round points
    const pointsStack: number[] = [];

    // Iterate over each operation in the array
    for (const operation of operations) {
        const length = pointsStack.length; // The current length of the stack

        switch (operation) {
            case '+': // If the operation is '+', add the last two round's points
                if (length >= 2) {
                    const lastRoundPoints = pointsStack[length - 1];
                    const secondLastRoundPoints = pointsStack[length - 2];
                    pointsStack.push(lastRoundPoints + secondLastRoundPoints);
                }
                break;
            case 'D': // If the operation is 'D', double the last round's points
                if (length >= 1) {
                    const lastRoundPoints = pointsStack[length - 1];
                    pointsStack.push(lastRoundPoints * 2);
                }
                break;
            case 'C': // If the operation is 'C', remove the last round's points
                if (length >= 1) {
                    pointsStack.pop();
                }
                break;
            default: // If the operation is a number, parse it and add to the stack
                const roundPoints = Number(operation);
                if (!isNaN(roundPoints)) {
                    pointsStack.push(roundPoints);
                }
                break;
        }
    }

    // Sum up all the points in the stack and return the total
    const totalPoints = pointsStack.reduce((accumulator, currentValue) => accumulator + currentValue, 0);
    return totalPoints;
}
```

```python
# Class to define the solution for the problem.
class Solution:
    # Function to calculate the final score using a list of operations.
    def calPoints(self, ops: List[str]) -> int:
        # Initialize a stack to keep track of the scores.
        scores_stack = []

        # Loop over each operation in the list of operations.
        for op in ops:
            # If operation is "+", add the sum of the last two scores to the stack.
            if op == '+':
                scores_stack.append(scores_stack[-1] + scores_stack[-2])
            # If operation is "D", double the last score and add to the stack.
            elif op == 'D':
                scores_stack.append(scores_stack[-1] * 2)
            # If operation is "C", remove the last score from the stack.
            elif op == 'C':
                scores_stack.pop()
            # Otherwise, the operation is a number, so parse and add to the stack.
            else:
                scores_stack.append(int(op))

        # Return the sum of the scores on the stack, which is the final score.
        return sum(scores_stack)
```

## Time and Space Complexity

The time complexity of the given code is $O(n)$, where $n$ is the length of the input list `ops`. This is because the code iterates through each element in `ops` once, and the operations performed within the loop (push, pop, addition, and doubling) are all constant-time operations, occurring in $O(1)$.

The space complexity of the code is also $O(n)$. In the worst case, if there are no 'C' operations to cancel the previous scores, the stack `stk` will have to store an integer for each input operation in `ops`. Therefore, the space used by the stack is directly proportional to the input size.