

Problem 682: Baseball Game

Problem Information

Difficulty: [Easy](#)

Acceptance Rate: 0.00%

Paid Only: No

Problem Description

You are keeping the scores for a baseball game with strange rules. At the beginning of the game, you start with an empty record.

You are given a list of strings

operations

, where

operations[i]

is the

i

th

operation you must apply to the record and is one of the following:

An integer

x

.

Record a new score of

x

+

Record a new score that is the sum of the previous two scores.

'D'

Record a new score that is the double of the previous score.

'C'

Invalidate the previous score, removing it from the record.

Return

the sum of all the scores on the record after applying all the operations

The test cases are generated such that the answer and all intermediate calculations fit in a

32-bit

integer and that all operations are valid.

Example 1:

Input:

```
ops = ["5", "2", "C", "D", "+"]
```

Output:

30

Explanation:

"5" - Add 5 to the record, record is now [5]. "2" - Add 2 to the record, record is now [5, 2]. "C" - Invalidate and remove the previous score, record is now [5]. "D" - Add $2 * 5 = 10$ to the record, record is now [5, 10]. "+" - Add $5 + 10 = 15$ to the record, record is now [5, 10, 15]. The total sum is $5 + 10 + 15 = 30$.

Example 2:

Input:

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

Output:

27

Explanation:

"5" - Add 5 to the record, record is now [5]. "-2" - Add -2 to the record, record is now [5, -2]. "4" - Add 4 to the record, record is now [5, -2, 4]. "C" - Invalidate and remove the previous score, record is now [5, -2]. "D" - Add $2 * -2 = -4$ to the record, record is now [5, -2, -4]. "9" - Add 9 to the record, record is now [5, -2, -4, 9]. "+" - Add $-4 + 9 = 5$ to the record, record is now [5, -2, -4, 9, 5]. "+" - Add $9 + 5 = 14$ to the record, record is now [5, -2, -4, 9, 5, 14]. The total sum is $5 + -2 + -4 + 9 + 5 + 14 = 27$.

Example 3:

Input:

```
ops = ["1", "C"]
```

Output:

0

Explanation:

"1" - Add 1 to the record, record is now [1]. "C" - Invalidate and remove the previous score, record is now []. Since the record is empty, the total sum is 0.

Constraints:

$1 \leq \text{operations.length} \leq 1000$

$\text{operations}[i]$

is

"C"

,

"D"

,

"+"

, or a string representing an integer in the range

$[-3 * 10^4, 3 * 10^4]$

4

$, 3 * 10^4]$

4

]

For operation

"+"

, there will always be at least two previous scores on the record.

For operations

"C"

and

"D"

, there will always be at least one previous score on the record.

Code Snippets

C++:

```
class Solution {  
public:  
    int calPoints(vector<string>& operations) {  
  
    }  
};
```

Java:

```
class Solution {  
public int calPoints(String[] operations) {  
  
}  
}
```

Python3:

```
class Solution:  
    def calPoints(self, operations: List[str]) -> int:
```

Python:

```
class Solution(object):
    def calPoints(self, operations):
        """
        :type operations: List[str]
        :rtype: int
        """

```

JavaScript:

```
/**
 * @param {string[]} operations
 * @return {number}
 */
var calPoints = function(operations) {
}
```

TypeScript:

```
function calPoints(operations: string[]): number {
}
```

C#:

```
public class Solution {
    public int CalPoints(string[] operations) {
    }
}
```

C:

```
int calPoints(char** operations, int operationsSize) {
}
```

Go:

```
func calPoints(operations []string) int {
```

```
}
```

Kotlin:

```
class Solution {  
    fun calPoints(operations: Array<String>): Int {  
  
    }  
}
```

Swift:

```
class Solution {  
    func calPoints(_ operations: [String]) -> Int {  
  
    }  
}
```

Rust:

```
impl Solution {  
    pub fn cal_points(operations: Vec<String>) -> i32 {  
  
    }  
}
```

Ruby:

```
# @param {String[]} operations  
# @return {Integer}  
def cal_points(operations)  
  
end
```

PHP:

```
class Solution {  
  
    /**  
     * @param String[] $operations  
     * @return Integer  
     */
```

```
function calPoints($operations) {  
}  
}  
}
```

Dart:

```
class Solution {  
int calPoints(List<String> operations) {  
  
}  
}  
}
```

Scala:

```
object Solution {  
def calPoints(operations: Array[String]): Int = {  
  
}  
}
```

Elixir:

```
defmodule Solution do  
@spec cal_points(operations :: [String.t]) :: integer  
def cal_points(operations) do  
  
end  
end
```

Erlang:

```
-spec cal_points(Operations :: [unicode:unicode_binary()]) -> integer().  
cal_points(Operations) ->  
.
```

Racket:

```
(define/contract (cal-points operations)  
(-> (listof string?) exact-integer?)  
)
```

Solutions

C++ Solution:

```
/*
 * Problem: Baseball Game
 * Difficulty: Easy
 * Tags: array, string, stack
 *
 * Approach: Use two pointers or sliding window technique
 * Time Complexity: O(n) or O(n log n)
 * Space Complexity: O(1) to O(n) depending on approach
 */

class Solution {
public:
    int calPoints(vector<string>& operations) {

    }
};
```

Java Solution:

```
/**
 * Problem: Baseball Game
 * Difficulty: Easy
 * Tags: array, string, stack
 *
 * Approach: Use two pointers or sliding window technique
 * Time Complexity: O(n) or O(n log n)
 * Space Complexity: O(1) to O(n) depending on approach
 */

class Solution {
    public int calPoints(String[] operations) {

    }
}
```

Python3 Solution:

```

"""
Problem: Baseball Game
Difficulty: Easy
Tags: array, string, stack

Approach: Use two pointers or sliding window technique
Time Complexity: O(n) or O(n log n)
Space Complexity: O(1) to O(n) depending on approach
"""

class Solution:

def calPoints(self, operations: List[str]) -> int:
    # TODO: Implement optimized solution
    pass

```

Python Solution:

```

class Solution(object):
    def calPoints(self, operations):
        """
        :type operations: List[str]
        :rtype: int
        """

```

JavaScript Solution:

```

/**
 * Problem: Baseball Game
 * Difficulty: Easy
 * Tags: array, string, stack
 *
 * Approach: Use two pointers or sliding window technique
 * Time Complexity: O(n) or O(n log n)
 * Space Complexity: O(1) to O(n) depending on approach
 */

/**
 * @param {string[]} operations
 * @return {number}
 */
var calPoints = function(operations) {

```

```
};
```

TypeScript Solution:

```
/**  
 * Problem: Baseball Game  
 * Difficulty: Easy  
 * Tags: array, string, stack  
 *  
 * Approach: Use two pointers or sliding window technique  
 * Time Complexity: O(n) or O(n log n)  
 * Space Complexity: O(1) to O(n) depending on approach  
 */  
  
function calPoints(operations: string[]): number {  
  
};
```

C# Solution:

```
/*  
 * Problem: Baseball Game  
 * Difficulty: Easy  
 * Tags: array, string, stack  
 *  
 * Approach: Use two pointers or sliding window technique  
 * Time Complexity: O(n) or O(n log n)  
 * Space Complexity: O(1) to O(n) depending on approach  
 */  
  
public class Solution {  
    public int CalPoints(string[] operations) {  
  
    }  
}
```

C Solution:

```
/*  
 * Problem: Baseball Game  
 * Difficulty: Easy
```

```

* Tags: array, string, stack
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/
int calPoints(char** operations, int operationsSize) {
}

```

Go Solution:

```

// Problem: Baseball Game
// Difficulty: Easy
// Tags: array, string, stack
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(1) to O(n) depending on approach

func calPoints(operations []string) int {
}

```

Kotlin Solution:

```

class Solution {
    fun calPoints(operations: Array<String>): Int {
        }
    }
}
```

Swift Solution:

```

class Solution {
    func calPoints(_ operations: [String]) -> Int {
        }
    }
}
```

Rust Solution:

```
// Problem: Baseball Game
// Difficulty: Easy
// Tags: array, string, stack
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(1) to O(n) depending on approach

impl Solution {
    pub fn cal_points(operations: Vec<String>) -> i32 {
        let mut stack = Vec::new();
        for operation in operations {
            match operation.as_ref() {
                "S" => stack.push(1),
                "D" => stack.push(stack.last().unwrap().clone() * 2),
                "C" => stack.pop(),
                "+" => {
                    let sum = stack.last().unwrap().clone() + stack[stack.len() - 2].clone();
                    stack.push(sum);
                },
                _ => {}
            }
        }
        stack.iter().sum()
    }
}
```

Ruby Solution:

```
# @param {String[]} operations
# @return {Integer}
def cal_points(operations)

end
```

PHP Solution:

```
class Solution {

    /**
     * @param String[] $operations
     * @return Integer
     */
    function calPoints($operations) {
        $stack = [];
        foreach ($operations as $operation) {
            if ($operation == "S") {
                $stack[] = 1;
            } elseif ($operation == "D") {
                $stack[] = $stack[count($stack) - 1] * 2;
            } elseif ($operation == "C") {
                array_pop($stack);
            } elseif ($operation == "+") {
                $stack[] = $stack[count($stack) - 1] + $stack[count($stack) - 2];
            }
        }
        return array_sum($stack);
    }
}
```

Dart Solution:

```
class Solution {
    int calPoints(List<String> operations) {
```

```
}
```

```
}
```

Scala Solution:

```
object Solution {  
    def calPoints(operations: Array[String]): Int = {  
  
    }  
    }  
}
```

Elixir Solution:

```
defmodule Solution do  
  @spec cal_points(operations :: [String.t]) :: integer  
  def cal_points(operations) do  
  
  end  
end
```

Erlang Solution:

```
-spec cal_points(Operations :: [unicode:unicode_binary()]) -> integer().  
cal_points(Operations) ->  
.
```

Racket Solution:

```
(define/contract (cal-points operations)  
  (-> (listof string?) exact-integer?))  
)
```