1049. Last Stone Weight II

Dynamic Programming

Problem Description

Array

picked in every turn and smashed against each other. If the weights of the stones are equal (x == y), both stones are destroyed. If they are not (x != y), the lighter stone is destroyed and the heavier stone's weight is reduced by the weight of the lighter stone. The aim of the game is to minimize the weight of the last remaining stone after all turns have been played. If no stones remain, the weight is considered to be 0.

The problem involves a game played with a collection of stones, each with a certain weight. During the game, two stones are

To solve this problem, we can think of it in terms of a 0/1 Knapsack problem, a common strategy used in dynamic programming.

Intuition

Medium

exceeding the weight limit.

In the 0/1 Knapsack problem, we aim to maximize the value of items that can be fit into a knapsack of limited capacity without

For this game, we can imagine having a "knapsack" with a capacity that is half the sum of all stone weights. The reason we use half the sum is that we seek to partition the stones into two groups that are as equal in total weight as possible. By doing so,

So, we construct a dynamic programming array dp where dp[j] keeps the maximum achievable sum of stones that does not exceed the capacity j. We iterate over each stone and update the dp array, considering the current stone to either be included

when the stones from these groups battle each other, the resulting stone (if any) will have the minimal possible weight.

or excluded from our "knapsack". Eventually, dp[-1] will contain the largest weight that is less than or equal to the sum of all stones divided by two. The smallest possible weight of the final stone can then be calculated as the total weight of all stones minus twice the weight

indicated by dp[-1], because for every weight we put in one group, we are effectively removing it from what would be the

Solution Approach The solution uses dynamic programming to solve this modified knapsack problem. Here's a step-by-step walkthrough of the

Calculate the sum s of all stones. The objective is to partition the stones into two groups with a total weight that is as close

implementation:

opposing group of stones in the "game".

to s / 2 as possible. Define n as s / 2 because this represents the largest possible weight our "knapsack" may store. Since we only need an integer knapsack weight, we use a bitwise right shift s >> 1, equivalent to s / 2 but faster computationally.

- Initialize a dynamic programming array dp of size n + 1 with zeros. This array will help us to keep track of the maximum weight that can be achieved for each capacity up to n. dp[j] will represent the maximum weight that can be accumulated
- with a knapsack capacity of j. The outer loop goes through each stone's value v in the stones array.

The inner loop is a reversed loop from n to v. This loop updates the dp array. For each capacity j from n down to the

situations: not taking the current stone (leaving dp[j] as it is) or taking the stone (which would be dp[j - v] + v). If the

- weight of the current stone v, we check whether including the stone would lead to a better result compared to not including it. Within the inner loop, the expression dp[j] = max(dp[j], dp[j - v] + v) updates the current dp[j] value. It evaluates two
- After filling the dp array, the maximum sum we can get from one group of stones is dp[-1]. The smallest possible weight of the last remaining stone is then s - dp[-1] * 2. We subtract double dp[-1] from the total sum s because, effectively, for

Let's go through a small example to illustrate the solution approach with a stones array $\frac{1}{2}$, $\frac{1}{2}$, $\frac{1}{2}$, $\frac{1}{2}$, $\frac{1}{2}$, $\frac{1}{2}$.

every stone placed in one group, we remove that weight from the potential remaining stone.

latter offers a higher weight without exceeding the capacity j, we update dp[j].

By using dynamic programming, this solution efficiently determines the best way to "pack" the stones into two groups with the goal of minimizing the weight of the last remaining stone after all possible smashes. **Example Walkthrough**

groups with a total weight that is as close to $\frac{1}{5}$ / $\frac{2}{5}$ = $\frac{23}{5}$ / $\frac{2}{5}$ = $\frac{11.5}{5}$ as possible for minimizing the last stone's weight. We define n as the integer part of s / 2, which gives us n = 11 since we're dealing with integer weights.

We first calculate the sum s of all stones, which is 2 + 7 + 4 + 1 + 8 + 1 = 23. The goal is to partition the stones into two

We initialize a dynamic programming array dp of size n + 1 with zeros: dp = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],

corresponding to knapsack capacities from 0 to 11. We start iterating through the stones array:

2, 2, 2, 2, 2, 2]

 \circ For stone value v = 2, we update dp from end to start where j >= v. \circ For stone value v = 7, we repeat the updating process.

- We continue until all stones are considered. For the first iteration with v = 2, the inner loop will check capacities from 11 down to 2. The update looks like:
- \circ For j = 11, we consider including the stone or not: dp[11] = max(dp[11], dp[11 2] + 2). \circ For j = 10, decide between dp[10] and dp[10 - 2] + 2.
- \circ And so on, down to j = 2. After we process v = 2, the dp array changes indicating the best way to pack stones up to now: dp = [0, 0, 2, 2, 2,

The maximum sum we can get from one group of stones is dp[-1] which is 11. This means that one group of stones has a

So, the algorithm leads to the partitioning of the stones into two groups whose weights differ by at most 1. The weight of the last

stone—if any—is minimized, which in this case is 1, as this is the least weight we can achieve by using dynamic programming

We process all stones this way, and the dp array continues to update with the maximum weights for each capacity.

At the end, assuming after all updates our dp array looks like: dp = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]

total weight of 11, and so does the second group, as we wanted two equal groups. The smallest possible weight of the last remaining stone is then s - dp[-1] * 2 = 23 - 11 * 2 = 1. 10.

Calculate the maximum possible weight that can be reached by any subset of stones

- and the knapsack analogy.
- **Python** from typing import List class Solution:

Initialize the dynamic programming table with zeros. The table will have 'max_possible_weight + 1' elements.

This ensures that each stone is only used once when calculating the maximum sum for each subset.

The final answer is the difference between the total weight and twice the weight of one of the subsets

This is because the other subset will have a total weight of `total_weight - dp[-1]`, resulting in the minimum possible dif

At each position, we want to take the maximum between: # (the current value at dp[current weight]) and (the value at dp[current weight - stone] + stone) # This represents either not taking the current stone or taking it and adding its weight to the subset. dp[current_weight] = max(dp[current_weight], dp[current_weight - stone] + stone)

int sumOfStones = 0;

for (int weight : stones) {

sumOfStones += weight;

// The number of stones in the array

// smallest last stone weight possible

for (int stone : stones) { // for each stone

return totalWeight - dp[targetWeight] * 2;

for (int i = targetWeight; i >= stone; --i) {

* Solves the Last Stone Weight II problem using dynamic programming.

* @param {number[]} stones An array of integers representing stones' weights.

* @return {number} The minimum possible weight difference between two groups.

* The function calculates the minimum possible weight difference

* between two groups into which the stones can be divided.

function lastStoneWeightII(stones: number[]): number {

let dp: number[] = new Array(target + 1).fill(0);

const target = sum >> 1;

// Update dp[i] to the higher of the two values;

dp[j] = max(dp[j], dp[j - stone] + stone);

// Traverse dp array backwards for this pass, to avoid using a stone twice

// This represents the minimal possible weight difference between two groups

// Initialize the dp array with zeros for storing the maximum sum possible for each subset sum.

// either the current dp[i] or the sum of stone and dp[j - stone] if we include the stone

// The answer is the total weight minus twice the optimized closest sum to half of the total weight

int numOfStones = stones.length;

for stone in stones:

Solution Implementation

def lastStoneWeightII(self, stones: List[int]) -> int:

Calculate the sum of all stones

max_possible_weight = total_weight >> 1

Iterate over each stone in the input list

// Calculate the sum of all stones' weights

// Set the target as half of the sum, because we are trying

// to minimize the difference between two groups to get the

int target = sumOfStones >> 1; // Equivalent to sumOfStones / 2

Update the dynamic programming table in reverse order.

for current weight in range(max possible weight, stone -1, -1):

 $dp = [0] * (max_possible_weight + 1)$

return total_weight - dp[-1] * 2

total weight = sum(stones)

```
Java
class Solution {
    public int lastStoneWeightII(int[] stones) {
```

```
// Initialize a DP array where dp[i] will store the maximum
        // weight that can be achieved with a sum not exceeding i
        int[] dp = new int[target + 1];
        // Loop through each stone
        for (int weight : stones) {
            // Update dp array in a reverse manner to ensure that
            // each stone is only used once
            for (int j = target; j >= weight; --j) {
                // Determine whether to include the current stone or not
                // by comparing which choice gives us a higher weight
                // that does not exceed the current weight limit j
                dp[j] = Math.max(dp[j], dp[j - weight] + weight);
        // The result is the difference between the sum of all stones
        // and twice the weight of the heavier group
        // This is because we are trying to partition the array into
        // two groups such that the difference between the sum of two groups
        // is minimized (typical partition problem that can be solved by DP).
        return sumOfStones - dp[target] * 2;
C++
#include <vector>
#include <numeric>
using namespace std;
class Solution {
public:
    int lastStoneWeightII(vector<int>& stones) {
        // Calculate the sum of all stone weights
        int totalWeight = accumulate(stones.begin(), stones.end(), 0);
        // Target weight is to try splitting stones into two groups with equal weight
        int targetWeight = totalWeight >> 1;
        // Initialize a DP array to store maximum achievable weight for each possible weight up to targetWeight
        vector<int> dp(targetWeight + 1, 0);
        // DP approach to find the closest sum to the targetWeight
```

// Calculate the sum of all stones. let sum = stones.reduce((acc, v) => acc + v, 0); // Calculate half of the sum as the target sum for one subset.

};

/**

TypeScript

```
// Update the dp array to find the maximum subset sum less than or equal to half the total sum.
    for (let stone of stones) {
        for (let i = target; i >= stone; ---i) {
            dp[j] = Math.max(dp[j], dp[j - stone] + stone);
    // The result is the total sum minus twice the maximum subset sum
    // which gives the smallest possible difference.
    return sum - dp[target] * 2;
// Example usage:
// const stones = [2, 7, 4, 1, 8, 1];
// const result = lastStoneWeightII(stones);
// console.log(result); // Output should be the minimum possible weight difference
from typing import List
class Solution:
    def lastStoneWeightII(self, stones: List[int]) -> int:
       # Calculate the sum of all stones
        total weight = sum(stones)
       # Calculate the maximum possible weight that can be reached by any subset of stones
       max_possible_weight = total_weight >> 1
       # Initialize the dynamic programming table with zeros. The table will have 'max_possible_weight + 1' elements.
        dp = [0] * (max_possible_weight + 1)
       # Iterate over each stone in the input list
        for stone in stones:
           # Update the dynamic programming table in reverse order.
           # This ensures that each stone is only used once when calculating the maximum sum for each subset.
            for current weight in range(max possible weight, stone -1, -1):
               # At each position, we want to take the maximum between:
               # (the current value at dp[current weight]) and (the value at dp[current weight - stone] + stone)
               # This represents either not taking the current stone or taking it and adding its weight to the subset.
               dp[current_weight] = max(dp[current_weight], dp[current_weight - stone] + stone)
       # The final answer is the difference between the total weight and twice the weight of one of the subsets
       # This is because the other subset will have a total weight of `total_weight - dp[-1]`, resulting in the minimum possible dif
        return total_weight - dp[-1] * 2
Time and Space Complexity
```

Time Complexity The time complexity of the code can be determined by looking at the nested loops. There is an outer loop that iterates over each

current stone "v" (i.e., "range(n, v - 1, -1)"). The inner loop ensures that each sub-sum only considers each stone once. • The outer loop runs once for each stone, so it runs "m" times where "m" is the number of stones ("len(stones)"). • The inner loop runs (at most) "n" times for each outer iteration, where "n" is half the sum of all stone weights, rounded down ("s >> 1").

The given code is an implementation of a dynamic programming solution to solve a variation of the classic knapsack problem. To

stone, and an inner loop that runs in reverse from "n" (where "n" is half of the sum of stones, "s >> 1") to the value of the

Therefore, the overall time complexity is O(mn), where "m" is the number of stones and "n" is half the sum of the stones' weights.

analyze the time complexity and space complexity, let us explore the code:

Space Complexity

The space complexity of the code is determined by the storage requirements. The array "dp" of size "n + 1" is created to store the maximum achievable weight for each sub-sum. No other data structures are used that grow with the size of the input. Thus, the space complexity is O(n), where "n" is half the sum of the stones' weights.