1872. Stone Game VIII **Dynamic Programming** Game Theory Hard

# **Problem Description**

In this game, there are n stones with different values arranged in a row. Two players, Alice and Bob, take turns playing the game with Alice starting first. The goal for Alice is to maximize her score difference over Bob, and Bob aims to minimize this difference. During each turn, while there is more than one stone remaining, the following steps are taken: 1. The player chooses an integer x > 1 and removes the leftmost x stones.

**Prefix Sum** 

- 3. A new stone, with value equal to that sum, is placed at the left side of the row.

point, hence the use of max(f, pre\_sum[i] - f).

2. The player adds the sum of the removed stones' values to their score.

The task is to determine this score difference assuming both players play optimally.

The game stops when there is only one stone left. The final score difference is calculated as (Alice's score - Bob's score).

Intuition

The intuition behind solving this problem lies in dynamic programming and the observation that a player will always prefer the option that leads to the maximum difference in their favor at any given point. Since we want to know the outcome when both players act optimally, we can work backward from the end of the game.

The code provided uses 'prefixed sum' (pre\_sum) and formulates the solution based on dynamic programming principles. We

begin by calculating the prefix sums of the stones' array, which is used to easily compute the sum of any selected stones during each turn. As Alice starts first and wants to maximize the difference, she'll choose moves that will leave Bob with the minimum score

difference. Meanwhile, Bob will try to do the opposite. However, since Alice starts and makes the first move, she always has an upper hand. This kind of game falls under the umbrella of 'minimax' problems, where players minimize the maximum possible loss. To find the optimal strategy, we start from the right-most position in the game (which is the second to the last stone since the

last stone cannot be chosen due to the x > 1 rule) and move leftward to calculate the optimal score difference at each position

considering the current sum and the best outcome of the future state (dynamic programming). The variable f maintains the maximum possible score difference. To get the final answer, we loop backwards through the array starting from the second to last stone, and at each point, we consider two possibilities: taking the current prefix sum minus the optimal future score (as if Bob just played), or keeping the

previous optimal score (as if Alice is retaining her advantage). The max of these two values gives us Alice's optimal choice at this

After iterating through the stones, the score stored in f will be the maximum score difference Alice can achieve against Bob

Solution Approach The solution uses a dynamic programming approach as it computes the optimal score difference at each step by considering the

previous steps' outcomes. Dynamic programming is a method for solving complex problems by breaking them down into simpler

## Accumulate Function: The accumulate function from Python's itertools module is used to create the pre\_sum list, which is

f = pre\_sum[len(stones) - 1]

for i in range(len(stones) - 2, 0, -1):

At each step, we're considering two scenarios:

tries to maximize her score difference from this point onward.

when both play optimally.

pre\_sum = list(accumulate(stones)) **<u>Dynamic Programming</u> State**: The variable f is used to maintain the optimal score difference while iterating. It starts with the

value of the last <u>prefix sum</u> since this would be the maximum starting score for Alice if she were to take all stones except the first one in her first turn.

Here's the implementation detail with the algorithms, data structures, and patterns used in the provided Python code:

a prefix sum array of the stones. This array helps in calculating the sum of the stones taken in one move quickly.

subproblems, solving each of those subproblems just once, and storing their solutions.

 $\circ$  We start from the second-to-last stone since one cannot take just the last stone (x > 1).

• f: This is the best score difference before this move, considering we're optimizing for Alice.

- Iterating Backwards: The core of the dynamic programming approach happens when we iterate over the pre\_sum array in reverse, updating the state of f.
- $f = max(f, pre_sum[i] f)$ Here's what happens in the loop:

• pre\_sum[i] - f: This is the current sum of stones that can be taken subtracted by the best score difference from all future moves

• The max function is used to select the better option to update the score difference in Alice's favor. This simulates Alice's optimal play as she

(working backwards means we're considering as if Bob takes his turn now, and Alice tries to counter in the future).

approach to dynamic programming, considering the optimal future states while computing the current state.

value, as this is the best score Alice can secure if she takes all stones on her first move.

■ New option: pre\_sum[2] - f = 12 - 14 = -2 (if Bob took a turn, considering optimal future moves)

```
final result.
This algorithm is particularly efficient since only a single pass through the game states is necessary after the accumulation,
resulting in a time complexity of O(n), where n is the length of the stones array. This strategy effectively relies on a bottom-up
```

Finally, f will represent the optimal score difference Alice can obtain when both players play optimally, and we return this as the

7, 3, 4]. Here's how we would apply the algorithm described in the solution approach: Initial Preparations: Compute the prefix sums of the stones array to facilitate quick sums.

Let's walk through the solution approach with an example where the game starts with a row of stones with values stones = [2,

stones: [2, 7, 3, 4] pre\_sum: [2, 9, 12, 16] // Prefix sums using the accumulate function

**Dynamic Programming Initialization**: Initialize the variable f with the value of the last prefix sum minus the first stone's

## Iterating Backwards: Iterate through the pre\_sum list from right to left, starting from the second-to-last position.

f: 16 - 2 = 14

Solution Implementation

from itertools import accumulate

return score

from typing import List

**Python** 

class Solution:

**Example Walkthrough** 

■ Alice decides the max value for  $f: \max(14, -2) = 14$  (Alice will select 14 to maximize her score difference) Iteration 2 (i = 1): Evaluate the score if taking the first two stones. Current f: 14 (stored best score difference so far)

After this iteration, Alice cannot make more decisions since there needs to be more than one stone to play the game.

By the end of the iterations, f is 14. Since Alice can choose to remove the last three stones in her first turn, leaving the first stone

for Bob, this leads to a final score difference of final score = Alice's Score - Bob's Score = 14 - 2 = 12. Therefore, using the solution approach described earlier, we determined that the optimal final score difference is 12 in Alice's

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

prefix\_sum = list(accumulate(stones))

for i in range(len(stones) -2.0.-1):

public int stoneGameVIII(int[] stones) {

int[] prefixSum = new int[n];

for (int i = 1; i < n; ++i) {

int maxScore = prefixSum[n - 1];

for (int i = n - 2; i > 0; --i) {

for (int i = 1; i < n; ++i) {

int maxScore = prefixSum[n - 1];

for (int i = n - 2; i > 0; --i) {

return maxScore;

**}**;

**TypeScript** 

// Return the final calculated 'maxScore'

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

// 'n' is the total number of stones.

let prefixSums: number[] = new Array(n);

// Return the final calculated 'maxScore'.

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

# Calculate the prefix sum array of the stones list

process to compute the maximum score that Alice can achieve.

return maxScore;

from typing import List

class Solution:

from itertools import accumulate

let n: number = stones.length;

// Return the final calculated maxScore

prefixSum[0] = stones[0];

return maxScore;

int n = stones.length;

// n represents the total number of stones

score = max(score, prefix\_sum[i] - score)

# Calculate the prefix sum array of the stones list

• Iteration 1 (i = 2): Evaluate the score if taking first three stones.

Current f: 14 (stored best score difference so far)

favor when both Alice and Bob play optimally.

■ New option: pre\_sum[1] - f = 9 - 14 = -5 (if Bob took his turn here)

■ Alice decides the max value for f: max(14, -5) = 14 (Alice still selects 14)

# Initialize the 'score' variable with the last value of the prefix sum, # which corresponds to the sum of all stones. score = prefix\_sum[-1] # Reverse iterate over the prefix sum array starting from the second last element

# The loop goes till the second element as the first move can't use only one stone

# Update the 'score' to be the maximum value between the current 'score'

# and the difference between the current prefix sum and the 'score'

# Return the final score after both players have played optimally

// preSum array is used to store the prefix sum of the stones

// The first element of preSum is the first stone itself

prefixSum[i] = prefixSum[i - 1] + stones[i];

// f represents the maximum score that can be achieved

// Calculate the prefix sum for the entire array of stones

// Traverse backwards through the stones, updating maxScore

// Calculate the prefix sum for the entire vector of stones

// 'maxScore' represents the maximum score that can be achieved

// Traverse backwards through the stones, updating 'maxScore'

maxScore = std::max(maxScore, prefixSum[i] - maxScore);

// 'prefixSums' array is used to store the running sum of the stones.

// Start by assuming the last element of 'prefixSum' is the maximum score

// The 'maxScore' is updated to be the maximum of current 'maxScore'

// and the difference of the current prefix sum and 'maxScore'

prefixSum[i] = prefixSum[i - 1] + stones[i];

// Start by assuming the last element of prefixSum is the maximum score

// The maxScore is updated to be the maximum of current maxScore

// and the difference of the current prefix sum and maxScore

maxScore = Math.max(maxScore, prefixSum[i] - maxScore);

# This represents choosing the optimal score after each player's turn

Java class Solution {

```
C++
#include <vector>
#include <algorithm>
class Solution {
public:
    int stoneGameVIII(std::vector<int>& stones) {
       // n represents the total number of stones
        int n = stones.size();
        // 'prefixSum' vector is used to store the prefix sum of stones
        std::vector<int> prefixSum(n);
        // The first element of 'prefixSum' is the first stone itself
        prefixSum[0] = stones[0]:
```

```
// The first element of 'prefixSums' is the first stone itself.
prefixSums[0] = stones[0];
// Compute the running sum for the entire array of stones.
for (let i = 1; i < n; i++) {
    prefixSums[i] = prefixSums[i - 1] + stones[i];
// 'maxScore' represents the maximum score achievable.
// Initialize it with the sum of all stones, as if you took all.
let maxScore: number = prefixSums[n - 1];
// Iterate backwards through the stones to update 'maxScore'.
for (let i = n - 2; i > 0; i--) {
   // Update 'maxScore' to be the maximum between the current 'maxScore'
   // and the sum of stones up to 'i', minus the subsequent 'maxScore'.
   maxScore = Math.max(maxScore, prefixSums[i] - maxScore);
```

```
prefix_sum = list(accumulate(stones))
        # Initialize the 'score' variable with the last value of the prefix sum,
        # which corresponds to the sum of all stones.
        score = prefix_sum[-1]
        # Reverse iterate over the prefix sum array starting from the second last element
        # The loop goes till the second element as the first move can't use only one stone
        for i in range(len(stones) -2, 0, -1):
           # Update the 'score' to be the maximum value between the current 'score'
           # and the difference between the current prefix sum and the 'score'
           # This represents choosing the optimal score after each player's turn
           score = max(score, prefix_sum[i] - score)
        # Return the final score after both players have played optimally
        return score
Time and Space Complexity
Time Complexity
```

### Calculating the prefix sums uses accumulate from Python's itertools, which iterates through the stones list once. This operation has a time complexity of O(n), where n is the length of the stones list.

The for-loop iterates from the second-to-last element to the first non-inclusive, meaning it executes n - 2 times. Each

The time complexity of the code is primarily determined by two operations: the calculation of the prefix sums and the iterative

- iteration performs a constant time operation, which is checking and updating the value of f. Therefore, the time complexity of this loop is also O(n).
- Combining these two operations, since they are sequential and not nested, the overall time complexity of the code remains O(n).

**Space Complexity** The space complexity is influenced by the additional space required for storing the prefix sums and the space for the variable f.

- The prefix sums are stored in pre\_sum, which is a list of the same length as stones, requiring O(n) space. The variable f is a single integer, which takes 0(1) space.
- Hence, the total space complexity of the code is O(n) due to the space needed for the pre\_sum list. The space needed for the integer f is negligible compared to the size of pre\_sum.