1962. Remove Stones to Minimize the Total

to note that the same pile can be chosen multiple times for successive operations.

which is exactly what's needed to find the pile with the most stones.

### **Problem Description**

Medium <u>Array</u> <u>Heap (Priority Queue)</u>

pile and remove half of the stones in it (rounded down). The goal is to minimize the total number of stones left after all the operations have been performed. In more detail, the operation consists of choosing any piles[i] and removing floor(piles[i] / 2) stones from it. It's important

You are given an integer array called piles, where each element piles[i] indicates the number of stones in the i-th pile. You

are also given an integer k. Your task is to perform exactly k operations on these piles, where in each operation you choose a

After applying k operations, your function should return the minimum possible total number of stones remaining across all piles.

Intuition

The intuition behind the solution is to always target the pile with the largest number of stones for the reduction operation. This is because removing half from a larger number results in a bigger absolute reduction compared to removing half from a smaller pile.

Over k operations, this strategy ensures the greatest possible reduction in the total number of stones. To effectively apply this strategy, a max-heap is used. A max-heap is a binary tree structure where the parent node is always larger than or equal to the child nodes. This property allows for the efficient retrieval of the largest element in the collection,

every value inserted into the heap is negated both when it's put in and when it's taken out. Here's the step-by-step thought process: 1. Negate all the elements in piles and store them in a heap to create a max-heap out of the Python min-heap implementation.

In Python, the heapq library provides a min-heap, which can be turned into a max-heap by inserting negative values. Therefore,

2. For k iterations, remove the biggest element from the heap (which is the most negative), negate it to get the original value, and then remove half of it (as per the rules of the operation). 3. Insert back the negated new value (after halving the biggest element) into the heap.

4. After performing k operations, the heap will contain the negated values representing the remaining stones. To get the total, negate the sum of the heap's contents.

Solution Approach

By always choosing the largest pile for halving, the algorithm ensures that the decrease in the total stone count is maximized with

The solution leverages a max-heap data structure to efficiently track and remove the largest piles first. Here's how the algorithm and data structures are utilized in the provided solution:

elements are negated before being pushed onto the heap to simulate a max-heap functionality.

A heap is created to maintain the current state of the piles. Since Python's heapq library implements a min-heap, the

### • The root of the heap (which in this case, is the maximum element, due to negation) is popped off the heap using heappop(). Since the

for \_ in range(k):

**Example Walkthrough** 

each operation.

elements are negated when stored in the heap, we negate it back to get the actual number of stones in the largest pile. We then take the floor of halved value of this pile. The halving operation is conducted by adding 1 to the pile count and right-shifting (>> 1) by one, which in essence is dividing by two and using the floor value in integer division.

heappush(). The code for this loop is as follows:

• This new halved number of stones is negated and pushed back onto the heap to maintain the max-heap order. This is done using

p = -heappop(h)heappush(h, -((p + 1) >> 1))

summed value as the total minimum possible number of stones remaining.

We use the heapq library to turn this into a heap: h = [-10, -4, -2, -7].

We iterate k times, each time performing the following steps:

This final step is summarized by this line of code: return -sum(h) This approach ensures that we are always working with the largest pile available after each operation, thus optimizing our

Finally, after k iterations, the heap contains the negated counts of stones in each pile after the operations have been applied.

To find the answer, we sum up all the elements in the heap (which involves negating them back to positive) and return this

• piles = [10, 4, 2, 7]

Now, let's walk through the solution step by step:

strategy for minimizing the total number of stones.

**Initial State** First, we negate all the elements in piles to simulate a max-heap using Python's min-heap implementation: [-10, -4, -2, -7].

Let's consider a small example to illustrate the solution approach. Suppose we have the following piles and value of k:

### Since -10 is the maximum value (or smallest when considering negation), it is popped from the heap. We negate it to find the original amount of stones: 10.

**Operation 2** 

**Operation 3** 

**Final Summation** 

•

**Operation 1** 

• k = 3

-5. Now the heap is [-7, -4, -2, -5].

Next, we remove half of the stones in this pile. Half of 10 is 5. Since we need to negate and re-insert into the heap, we insert

Removing half (rounded down) gives 7 / 2 which is 3 when rounded down. Negating and re-inserting -3, the heap is now

[-5, -4, -2, -3].

The max value in our heap is now -5. We pop, negate, halve (rounding down to 2), and re-insert -2 into the heap.

The current maximum (minimum in our negated heap) is -7. We pop it and negate it: 7.

The final state of the heap after 3 operations is [-4, -3, -2, -2].

# Initialize a max heap as Python only has a min heap implementation.

# Pop the largest pile and reduce its stones by half, rounded up.

# The sum of the heap is negated to return the actual sum of pile sizes.

PriorityQueue<Integer> maxHeap = new PriorityQueue<>((a, b) -> b - a);

reduced\_pile = (largest\_pile + 1) >> 1 # Using bit shift to divide by 2

This gives us 4 + 3 + 2 + 2 = 11 as the final answer.

# We are inverting the values to simulate a max heap.

# Perform 'k' operations to reduce the stones in the piles

# Push the reduced pile size back into the max heap.

// Create a max-heap to store the piles in descending order

To get the total number of stones remaining, we sum the negated values of the heap: -(-4 + -3 + -2 + -2).

Solution Implementation

from heapq import heappop, heappush

for in range(k):

def minStoneSum(self, piles, k):

## So, after performing k = 3 operations on the piles, the minimum number of stones left is 11.

**Python** 

class Solution:

class Solution {

- max heap = []for pile in piles: # The minus sign ensures we create a max heap. heappush(max\_heap, -pile)
- return -sum(max\_heap) Java

// Perform the reduction operation k times

int minStoneSum(vector<int>& piles, int k) {

priority\_queue<int> maxHeap;

maxHeap.push(pile);

// Perform the operation k times

// Retrieve the pile with the most stones

int largestPile = maxHeap.top();

maxHeap.push((largestPile + 1) / 2);

remainingStones += maxHeap.top();

// Return the total number of remaining stones

for (int pile : piles) {

while (k-- > 0) {

maxHeap.pop();

int remainingStones = 0;

maxHeap.pop();

return remainingStones;

// Import the necessary components for Heap

// Perform the operation 'k' times

for (let i = 0; i < k; i++) {

while (!maxHeap.empty()) {

// Create a max-heap to keep track of the stones in the piles

// Populate the max-heap with the number of stones in each pile

// Calculate the remaining number of stones after k operations

// Remove half of the stones from the largest pile, round up if necessary

public int minStoneSum(int[] piles, int k) {

// Add each pile to the max-heap

for (int pile : piles) {

while (k-- > 0) {

maxHeap.offer(pile);

largest pile = -heappop(max heap)

heappush(max\_heap, -reduced\_pile)

### // Retrieve and remove the largest pile from the heap int largestPile = maxHeap.poll(); // Calculate the reduced number of stones and add back to the heap maxHeap.offer((largestPile + 1) >> 1); // Calculate the total number of stones after k reductions int totalStones = 0; while (!maxHeap.isEmptv()) { // Remove the stones from the heap and add them to the total count totalStones += maxHeap.poll(); // Return the total number of stones remaining after k operations return totalStones; C++ #include <vector> #include <queue> class Solution {

#### // Function signature in TypeScript with types defined function minStoneSum(piles: number[], k: number): number { // Create a max-heap to keep track of the stones in the piles let maxHeap = new Heap<number>(piles, null, (a: number, b: number) => b - a);

**}**;

**TypeScript** 

public:

// Retrieve the pile with the most stones let largestPile = maxHeap.pop(); // Assumes that Heap.pop retrieves the max element // Remove half of the stones from the largest pile, round up if necessary maxHeap.push(Math.ceil(largestPile / 2)); // Calculate the remaining number of stones after 'k' operations let remainingStones = 0; while (maxHeap.length > 0) { remainingStones += maxHeap.pop(); // Return the total number of remaining stones return remainingStones; // Note that the `Heap` class I'm using above does not exist in default JavaScript/TvpeScript and must be imported // from a library that provides heap data structure implementation. The example uses a fictional 'collections/heap.js'. // You'll need to find a suitable library with heap implementation, or you would have to implement your own // heap structure to use this function as intended. from heapq import heappop, heappush class Solution: def minStoneSum(self, piles, k): # Initialize a max heap as Python only has a min heap implementation. # We are inverting the values to simulate a max heap. max heap = []for pile in piles: # The minus sign ensures we create a max heap. heappush(max\_heap, -pile)

import { Heap } from 'collections/heap.js'; // This library or similar would need to be imported for heap functionality.

# Time and Space Complexity

**Time Complexity** 

return -sum(max\_heap)

for in range(k):

The time complexity of the given code is determined by the following major operations: 1. Converting the list piles into a heap: This operation is O(n) where n is the length of piles.

# Perform 'k' operations to reduce the stones in the piles

# Push the reduced pile size back into the max heap.

largest pile = -heappop(max heap)

heappush(max\_heap, -reduced\_pile)

# Pop the largest pile and reduce its stones by half, rounded up.

# The sum of the heap is negated to return the actual sum of pile sizes.

reduced\_pile = (largest\_pile + 1) >> 1 # Using bit shift to divide by 2

- 2. The main loop which is executed k times: The heap operation heappop: Each pop operation has a complexity of O(log n). • The update and heappush back into the heap: These have a complexity of O(log n) for each push operation.
- Therefore, the overall time complexity of the loop is 0(k \* log n). Combining this with the heap conversion, the total time

complexity is 0(n + k \* log n). **Space Complexity** 

The space complexity is determined by the space needed to store the heap. Since we are not using any additional data structures that grow with input size other than the heap, the space complexity is O(n), where n is the length of the list piles.