1046. Last Stone Weight

Easy | Array | Heap (Priority Queue) | Leetcode Link

Problem Description

In this problem, we have a collection of stones with different weights, given in an array `stones`, where each stone's weight is represented by `stones[i]`. We are simulating a game where we repeatedly smash the two heaviest stones together and determine the outcome according to the following rules:

- If both stones have the same weight (`x == y`), both stones get completely destroyed.
- If the weights are different (`x != y`), the lighter stone gets destroyed, and the weight of the heavier stone gets reduced by that of the lighter stone (`y - x`).

The game continues until there is either one stone left or no stones remaining. The goal is to return the weight of the last remaining stone. If there are no stones left as a result of the smashes, we should return `0`.

Intuition

The solution requires us to repeatedly find and remove the two heaviest stones. Since we need to do this repeatedly, a heap is an ideal data structure as it allows for efficient retrieval and updating of the largest elements.

A max heap keeps the maximum element at the top. However, Python's `heapq` module provides a min heap, so we insert the negative of stone weights to simulate the behavior of a max heap.

Here is the step-by-step approach to the solution:

1. Convert all stone weights to their negative and create a min heap. This negation is necessary because the Python `heapq` module only supports min heaps.
2. While there are at least two stones in the heap:
   1. Pop the heaviest stone (which is the smallest in the min heap due to negation) and store its negated value in `y`.
   2. Pop the second heaviest stone (again the smallest in our min heap) and store its negated value in `x`.
   3. If `x` and `y` are not equal, we push the weight difference negated (`x - y`) back onto the heap since stone `x` got destroyed and the weight of stone `y` reduced by `x`.
3. After the loop, if the heap is empty, it means no stones are left and we return `0`. Else, we return the negation of the weight of the last stone left in the heap (as we stored negative values, we need to negate it back to return the actual weight).

Using this approach, we efficiently simulate the stone smashing game and find the weight of the last stone or determine that no stones are left.

Solution Approach

The method `lastStoneWeight` is implemented using a min heap to efficiently manage the stones according to their weights.

Here's a breakdown of the solution:

- A min heap is created from the list of stones. Each stone's weight is negated prior to insertion because Python's heapq module works as a min heap. A min heap allows quick access to the smallest element, so by negating the weights, we get quick access to the largest element.

```python
h = [-x for x in stones]
heapify(h)
```

- The solution iteratively processes the heap until there's one or zero stones left. This is handled by a while loop that continues as long as the length of the heap (`h`) is greater than one.

- Inside the loop, the two heaviest stones (which are actually the two smallest values in the heap due to negation) are popped from the heap. This is done using `heappop(h)`, and the negated value of the pops represents `y` and `x`.

```python
y, x = -heappop(h), -heappop(h)
```

- If the weight of the two stones is not equal (`x != y`), it means that after smashing the stones, one of them (the lighter stone `x`) is completely destroyed, while the other (`y`) is reduced in weight. The difference (`y - x`) is computed, negated, and pushed back into the heap.

```python
if x != y:
    heappush(h, x - y)
```

- After the loop exits, which happens when only one stone is left or none at all, the function checks if the heap is empty. If it is empty (`not h`), the function returns `0`. Otherwise, it returns the weight of the last stone left, negating it to convert it back to the original weight value.

```python
return 0 if not h else -h[0]
```

Using heapq for maintaining a heap and negating values is an efficient approach to simulate a max heap in Python. This problem showcases an excellent use of the heap data structure to solve a problem related to constant removal and insertion of elements to achieve a sorted characteristic (largest or smallest).

Example Walkthrough

Let's consider a small example to illustrate the solution approach: Suppose we have an array of stone weights `stones = [2, 7, 4, 1, 8, 1]`. We need to perform the following steps:

1. Convert the weights to their negative and create a min heap:
   - We negate the weights: `[-2, -7, -4, -1, -8, -1]`
   - Create a min heap from these negated weights: `h = [-8, -7, -4, -1, -2, -1]`
2. While there are at least two stones in the heap:
   1. Pop the heaviest stone (smallest in negated form):
      - `y = -heappop(h)` gives us `y = 8`, and now `h = [-7, -2, -4, -1, -1]`
   2. Pop the second heaviest stone:
      - `x = -heappop(h)` gives us `x = 7`, and now `h = [-4, -2, -1, -1]`
   3. Push the weight difference negated back onto the heap since `x != y`:
      - The difference is `8 - 7 = 1`. After negating, we push `-1` back onto the heap.
      - Now, `h = [-4, -2, -1, -1, -1]`
3. Repeat the process:
   - Next, `y = -heappop(h)` gives us `y = 4`, `h = [-2, -1, -1, -1]`
   - Then, `x = -heappop(h)` gives us `x = 2`, `h = [-1, -1, -1]`
   - The difference is `4 - 2 = 2`. After negating, we push `-2` back onto the heap.
   - Now, `h = [-2, -1, -1]`
4. Continue:
   - Next, `y = -heappop(h)` gives us `y = 2`, `h = [-1, -1]`
   - Then, `x = -heappop(h)` gives us `x = 1`, `h = [-1]`
   - The difference is `2 - 1 = 1`. After negating, we push `-1` back onto the heap.
   - Now, `h = [-1, -1]`
5. Final iterations will destroy both of the stones because they have the same weight:
   - Next, `y = -heappop(h)` gives us `y = 1`, `h = [-1]`
   - Then, `x = -heappop(h)` gives us `x = 1`, `h = []`
   - Since `x == y`, we don't push anything back onto the heap.
6. Now the heap is empty (`h = []`), that is, no stones are left. We return `0`.

The final result for the example input `stones = [2, 7, 4, 1, 8, 1]` is `0`, as all stones are eventually destroyed.

Python Solution

```python
from heapq import heapify, heappush, heappop

class Solution:
    def lastStoneWeight(self, stones: List[int]) -> int:
        # Create a max heap by inverting the values of the stones
        max_heap = [-stone for stone in stones]
        heapify(max_heap)

        # Continue processing until there is one or no stones left
        while len(max_heap) > 1:
            # Pop the two largest stones from the heap
            # Stones are negated again to get their original values
            stone1 = -heappop(max_heap)
            stone2 = -heappop(max_heap)

            # If the largest stones are not of the same weight
            if stone1 != stone2:
                # The result of the collision is added back to the heap
                heappush(max_heap, -(stone1 - stone2))

        # If the heap is empty, return 0; else return the weight of the last stone
        return 0 if not max_heap else -max_heap[0]

```

Java Solution

```java
class Solution {
    /**
     * Simulate the process of smashing stones together and return
     * the weight of the last remaining stone (if any).
     *
     * @param stones An array of stone weights.
     * @return The weight of the last stone, or 0 if no stones are left.
     */
    public int lastStoneWeight(int[] stones) {
        // Create a max-heap to store and compare the stone weights in descending order
        PriorityQueue<Integer> maxHeap = new PriorityQueue<>((a, b) -> b - a);

        // Add all stone weights to the max-heap
        for (int stone : stones) {
            maxHeap.offer(stone);
        }

        // Continue until there is only one stone left or none at all
        while (maxHeap.size() > 1) {
            // Get the two heaviest stones
            int stoneOne = maxHeap.poll();
            int stoneTwo = maxHeap.poll();

            // If they are not the same weight, put the difference back into the max-heap
            if (stoneOne != stoneTwo) {
                maxHeap.offer(stoneOne - stoneTwo);
            }
            // If they are equal, both stones are completely smashed, and nothing is added back
        }

        // Return the last stone's weight or 0 if no stones are left
        return maxHeap.isEmpty() ? 0 : maxHeap.poll();
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <queue>

class Solution {
public:
    // Function to return the last stone's weight after smashing the largest two until one or none are left
    int lastStoneWeight(vector<int>& stones) {
        // Priority queue to store the stones with max heap property to easily retrieve the heaviest stones
        priority_queue<int> maxHeap;

        // Insert all stones into the priority queue
        for (int stone : stones) {
            maxHeap.push(stone);
        }

        // Loop until there is only one stone left or none
        while (maxHeap.size() > 1) {
            // Take out the heaviest stone
            int heaviestStone = maxHeap.top();
            maxHeap.pop();

            // Take out the second heaviest stone
            int secondHeaviestStone = maxHeap.top();
            maxHeap.pop();

            // If the two stones have different weights, push the difference back into the queue
            if (heaviestStone != secondHeaviestStone) {
                maxHeap.push(heaviestStone - secondHeaviestStone);
            }

            // If the stones have the same weight, both get destroyed and nothing goes back into the queue
        }

        // If there are no stones left, return 0, otherwise return the weight of the remaining stone
        return maxHeap.empty() ? 0 : maxHeap.top();
    }
};

```

Typescript Solution

```typescript
// Import the necessary module for Priority Queue
import { MaxPriorityQueue } from '@datastructures-js/priority-queue';

/**
 * Simulates a process where stones smash each other. If two stones have
 * different weights, the weight of the smaller one is subtracted from the other.
 * The smaller stone is then considered destroyed. The process repeats until
 * there is one stone left or none. The function returns the weight of the remaining
 * stone, or 0 if none are left.
 * @param {number[]} stones - An array of stone weights.
 * @return {number} The weight of the last remaining stone, or 0 if none.
 */
function getLastStoneWeight(stones: number[]): number {
    // Initialize a max priority queue for the stones
    const priorityQueue = new MaxPriorityQueue<number>();

    // Enqueue all the stones to the priority queue
    for (const stone of stones) {
        priorityQueue.enqueue(stone);
    }

    // Loop until there is either one stone left or none
    while (priorityQueue.size() > 1) {
        // Dequeue the two heaviest stones
        const heavierStone = priorityQueue.dequeue().element;
        const lighterStone = priorityQueue.dequeue().element;

        // If there is a weight difference, enqueue the difference as a new stone
        if (heavierStone !== lighterStone) {
            priorityQueue.enqueue(heavierStone - lighterStone);
        }
    }

    // If the priority queue is empty, return 0; otherwise, return the weight of the last stone
    return priorityQueue.isEmpty() ? 0 : priorityQueue.dequeue().element;
}

```

Time and Space Complexity

The given code implements a heap to solve the last stone weight problem. Let's analyze both the time complexity and the space complexity of the given code.

Time Complexity

The main operations in the algorithm are:

1. Converting the `stones` list into a heap which takes `O(n)` time, where `n` is the number of stones.
2. The `while` loop. In the worst case, the heap contains `n - 1` elements, and `heappop()` is called twice per iteration. Since each `heappop()` operation takes `O(log n)` time, and in each iteration, we might do a `heappush()` which also takes `O(log n)` time.
3. The loop runs at most `n - 1` times because in each iteration at least one stone is removed.

Putting it all together, the worst-case scenario would involve `(2 * log n + log n)` operations per iteration due to two `heappop()` calls and one potential `heappush()` call, across `n - 1` iterations. Hence, the total time complexity is `O(n log n)`.

Space Complexity

The space complexity consists of:

1. The heap `h` which stores at most `n` integers, thus requiring `O(n)` space.
2. Constant extra space for variables `x` and `y`.

So, the overall space complexity of the algorithm is `O(n)` since the heap size is proportional to the input size, and other space usage is constant.