1753. Maximum Score From Removing Stones

Medium Greedy Math Heap (Priority Queue)

Problem Description

You are tasked with a solitaire game involving three piles of stones. The sizes of the piles are given by three integers `a`, `b`, and `c`. The game is played by removing one stone from two different non-empty piles at a time, which adds one point to your score. This process continues until you are left with fewer than two non-empty piles, which means there are no more moves to make. The objective is to determine the maximum score possible by the end of the game.

Intuition

The solution approach involves some straightforward observations.

1. Whenever you take a stone from two piles, you want to reduce the difference between the number of stones in the piles to maximize the number of moves you can make. This is because removing stones from just the two largest piles when another pile is empty will end the game prematurely.
2. If the sum of the stones in the two smaller piles is less than the number of stones in the largest pile (`a + b < c`), then you will eventually have to pick from the largest pile (`c`) until it equals the sum of the other two piles. You cannot score more points than the total stones available in the two smallest piles.
3. Otherwise, if `a + b >= c`, you can keep taking stones from the two piles with the most stones until all piles have the same number of stones. Then, you simply alternate between pairs of piles, making each of them empty around the same time, ending the game with only one stone left in one pile or none.

Consequently, the maximum score is the sum of the stones in all piles divided by two, using integer division, since each move decreases the total number of stones by two (`(a + b + c) >> 1`). Integer division is important because if there's an odd number of total stones, one will remain at the end unpaired, and thus, not contributing to the score.

The solution uses a bitwise right shift `>>` by 1 to perform integer division by 2 on the sum of the stone counts, which is a common practice in programming for efficiency.

Solution Approach

The implementation of the solution is direct and doesn't require complicated algorithms or data structures. It's an arithmetic problem that is solved using basic mathematical operations and sorting. Here's a step-by-step walkthrough of the implementation:

1. **Sorting the Inputs**: The first step is to sort the integers `a`, `b`, and `c`. Since the problem doesn't require you to work with the original piles but just their sizes, sorting simplifies the logic by always ensuring that `a <= b <= c` (`a`, being the smallest, and `c` the largest after sorting).
2. **Handling the Edge Case**: We then check if the sum of the smaller two piles (`a + b`) is less than the size of the largest pile (`c`). If this condition is true, this means that you can continue to take stones from the two smaller piles until one or both of them are empty, and you can score no more than `a + b` points. Therefore, if `a + b < c`, we simply return `a + b`.
3. **Calculating Maximum Score Otherwise**: If the previous condition isn't met (i.e., `a + b >= c`), it indicates that you can balance the stones between the three piles while making moves. In this case, the maximum score you can get is half of the total number of stones, since each move reduces the total stone count by 2. This is calculated using the expression `(a + b + c) >> 1` which is equivalent to dividing the sum by two, but it is computed more efficiently through bitwise shifting.

No complex data structures are needed because we track only the sizes of the piles, not the individual stones. The problem doesn't require keeping track of the moves or the configuration of the piles after each move. As such, the algorithm is a clear instance of greedy strategy, where we make a local optimal choice (balancing the piles while we make moves) that leads to a global optimum (maximum score).

Example Walkthrough

Let's illustrate the solution approach with a small example. Consider three piles of stones with sizes `a = 2`, `b = 4`, and `c = 6`.

Following the steps outlined in the solution approach:

1. **Sorting the Inputs**: We sort the numbers so that `a <= b <= c`. In this case, `a = 2`, `b = 4`, and `c = 6`, which are already sorted.
2. **Handling the Edge Case**: We check if `a + b < c`. Here, `a + b = 2 + 4 = 6`, which is equal to `c`. Since it's not smaller than `c`, we don't return `a + b` as the score.
3. **Calculating Maximum Score Otherwise**: Since `a + b >= c`, we proceed to calculate the maximum score as the sum of all the stones divided by two. The total number of stones is `a + b + c = 2 + 4 + 6 = 12`. We then perform integer division by two, which can be done efficiently with a bitwise right shift: `(12 >> 1) = 6`.

Thus, the maximum score possible with these three piles is `6`. This means we can make 6 moves wherein each time, we remove one stone from two different piles until we can't make any more moves.

To visualize it, the sequence of moves could be:

- Move 1: Remove one stone from piles `b` and `c` (`a = 2`, `b = 3`, `c = 5`), score = 1.
- Move 2: Remove one stone from piles `b` and `c` (`a = 2`, `b = 2`, `c = 4`), score = 2.
- Move 3: Remove one stone from piles `a` and `c` (`a = 1`, `b = 2`, `c = 3`), score = 3.
- Move 4: Remove one stone from piles `a` and `b` (`a = 0`, `b = 1`, `c = 3`), score = 4. (Now `a` is empty)
- Move 5: Remove one stone from piles `b` and `c` (`a = 0`, `b = 0`, `c = 2`), score = 5. (Now `b` is also empty)
- Move 6: We cannot make any more moves as two piles are empty.

The games end with a maximum score of `6`, in alignment with our calculated maximum score.

Solution Implementation

Python

```python
class Solution:
    def maximumScore(self, num1: int, num2: int, num3: int) -> int:
        # Sort the input numbers to ensure num1 <= num2 <= num3
        num1, num2, num3 = sorted([num1, num2, num3])

        # If the sum of the two smallest numbers is less than the largest,
        # the maximum score is the sum of the smaller two.
        # This is because we can only take one stone each time,
        # and we should pair each stone from a smaller pile with one from
        # the largest pile until the smaller piles are empty.
        if num1 + num2 <= num3:
            return num1 + num2

        # Otherwise, the maximum score is half of the total number of stones,
        # rounded down, because we take one stone from two different piles
        # until it's not possible anymore.
        return (num1 + num2 + num3) // 2
        # Note: The right shift operator (>>) was replaced with floor division (//)
        # for better readability, as it's more commonly used to represent integer division.
```

Java

```java
class Solution {
    public int maximumScore(int a, int b, int c) {
        // Create an array to hold the input values
        int[] sides = new int[] {a, b, c};

        // Sort the array in ascending order
        Arrays.sort(sides);

        // Check if the sum of the two smaller numbers is smaller than the largest number
        if (sides[0] + sides[1] < sides[2]) {
            // The maximum score is the sum of the two smaller numbers
            return sides[0] + sides[1];
        }

        // Otherwise, the maximum score is half the sum of all three numbers
        // Using bitwise shift to the right to divide by 2
        return (a + b + c) >> 1;
    }
}
```

C++

```cpp
#include <vector> // Include the vector header for using std::vector
#include <algorithm> // Include the algorithm header for using std::sort

class Solution {
public:
    // Function to calculate the maximum score by choosing stones from three piles
    int maximumScore(int a, int b, int c) {
        // Create a vector to store the values of the three piles
        std::vector<int> piles = {a, b, c};

        // Sort the vector in non-decreasing order
        std::sort(piles.begin(), piles.end());

        // After sorting, piles[0] is the smallest, and piles[2] is the largest.
        // If the sum of the two smaller piles is less than the largest pile,
        // the maximum score is the sum of those two smaller piles.
        if (piles[0] + piles[1] < piles[2]) {
            return piles[0] + piles[1];
        } else {
            // Otherwise, the maximum score is half the sum of all piles.
            // because we will eventually be taking stones from two piles,
            // one by one, until one or both of the two smaller piles is empty.
            return (a + b + c) / 2; // Using integer division for the right shift equivalent
        }
    }
};
```

TypeScript

```typescript
// Function to calculate the maximum score by choosing stones from three piles
function maximumScore(a: number, b: number, c: number): number {
  // Create an array to store the values of the three piles
  const piles: number[] = [a, b, c];

  // Sort the array in non-decreasing order
  piles.sort((x, y) => x - y);

  // After sorting, piles[0] is the smallest, and piles[2] is the largest.
  // If the sum of the two smaller piles is less than the largest pile,
  // The maximum score is the sum of those two smaller piles.
  if (piles[0] + piles[1] < piles[2]) {
    return piles[0] + piles[1];
  } else {
    // Otherwise, the maximum score is half the sum of all piles,
    // because we will eventually be taking stones from two piles,
    // one by one, until one or both of the two smaller piles is empty.
    // Using Math.floor for integer division to get the equivalent result as right shifting in other languages like C++.
    return Math.floor((a + b + c) / 2);
  }
}

// Example usage:
// const score = maximumScore(2, 4, 6);
// console.log(`The maximum score is: ${score}`);
```

```python
class Solution:
    def maximumScore(self, num1: int, num2: int, num3: int) -> int:
        # Sort the input numbers to ensure num1 <= num2 <= num3
        num1, num2, num3 = sorted([num1, num2, num3])

        # If the sum of the two smallest numbers is less than the largest,
        # the maximum score is the sum of the smaller two.
        # This is because we can only take one stone each time,
        # and we should pair each stone from a smaller pile with one from
        # the largest pile until the smaller piles are empty.
        if num1 + num2 <= num3:
            return num1 + num2

        # Otherwise, the maximum score is half of the total number of stones,
        # rounded down, because we take one stone from two different piles
        # until it's not possible anymore.
        return (num1 + num2 + num3) // 2
        # Note: The right shift operator (>>) was replaced with floor division (//)
        # for better readability, as it's more commonly used to represent integer division.
```

Time and Space Complexity

Time Complexity

The time complexity of the code is determined by the sorting operation and the subsequent conditional checks. The sorting of three numbers has a constant time complexity, `O(1)`, since there are a finite number of permutations for three elements regardless of their values.

After sorting, we perform a conditional check and return a value based on the comparison. These operations are also done in constant time `O(1)` since we're just using arithmetic operations which take a fixed amount of time.

Therefore, the total time complexity of the code is `O(1)`.

Space Complexity

The space complexity of the code is also `O(1)` because the space required does not depend on the size of the input. All we're doing is assigning sorted values to the variables `a`, `b`, `c`, and doing a few arithmetic operations which do not require additional space that depends on the input size. The additional space used is constant.

So, the space complexity is `O(1)`.