948. Bag of Tokens

Medium Greedy Array Two Pointers Sorting

Leetcode Link

Problem Description

In this problem, you are given an initial power level and a set of tokens each with a value. The goal is to maximize your score by playing these tokens. You can play a token in one of two ways: if you have enough power, you can play it face up, consume power equal to the token's value, and increase your score by 1. Alternatively, if you have at least one score, you can play a token face down to gain power equal to its value, but you will lose one score in the process. You don't have to play all tokens, but you can only play each token once. The challenge is to decide the best strategy to play the tokens to get the highest possible score.

Intuition

The solution leverages a greedy approach. To maximize the score, the strategy is to increase the score when possible by playing the lowest value tokens face up, as this costs less power per score point gained. Conversely, when lacking power, but having at least one score, it's best to play the highest value tokens face down to gain the most power and enable more plays.

1. **Sorting the Tokens**: Begin by sorting the tokens in ascending order. This allows us to play the tokens with the lowest values first to maximize score efficiently.
2. **Greedy Play**: Use two pointers to track the lowest and highest value tokens not yet played. Start at the beginning (lowest value) for playing face up and the end (highest value) for playing face down.
3. **Playing Tokens Face Up**: While there is enough power to play the next lowest value token, do so. This increases the score and the count of tokens played.
4. **Maximizing Score**: Keep track of the maximum score achieved after each play to ensure the result reflects the highest score possible throughout the game.
5. **Playing Tokens Face Down**: If there isn't enough power to play the next token face up and the score is at least 1, play the highest value token face down. This sacrifices one score to potentially gain enough power to play multiple lower value tokens, thus increasing the score.
6. **Ending Conditions**: If there's no power to play any token face up and no score to play a token face down, or all tokens have been played, then no further actions can be taken and the current maximum score is the result.

The overall approach is to maximize score increment opportunities while maintaining the flexibility to regain power when necessary, without wasting potential score maximization from low-value tokens.

Solution Approach

The implementation follows the intuition of playing tokens in the most efficient way, using a greedy strategy. The code organizes the solution approach into a clear sequence of steps:

1. **Sort the Tokens**: First, the list of tokens is sorted using Python's built-in `.sort()` method. This organizes the tokens so that we can access the smallest and largest values quickly.
2. **Initialize Pointers and Variables**: Two pointers, `i` for the lowest value tokens (start of the list) and `j` for the highest value tokens (end of the list), are set up. They are used to select the next token to play. We also set up `ans` to track the maximum score attained, and `t` to track the current score.
3. **Main Loop**: The `while` loop continues as long as there are tokens left to play (`i <= j`). Inside the loop, we have conditions to decide whether to play a token face up or face down.
4. **Playing Tokens Face Up**: If the current power is sufficient to play the next lowest value token (`power >= tokens[i]`), the power is reduced by the value of the token, the score counter `t` is incremented, and the lowest token pointer `i` is moved up. The optimal score `ans` is updated to the maximum of its current value and the new score `t`.
5. **Playing Tokens Face Down**: If there's not enough power to play the next token face up, but we have at least one score (`t`), then we play the highest value token face down (`power += tokens[j]`), gain power equivalent to that token's value, and decrease the token and score counters (`j` and `t`).
6. **Insufficient Resources**: If we have insufficient power to play a token face up and no score to play a token face down, the loop breaks as no further plays are possible.
7. **Result**: Once the loop finishes, the maximum score `ans` is returned.

By using sorting and a greedy strategy, the algorithm ensures that we play tokens in a way that is most beneficial at each step, guaranteeing the maximum score possible with the given initial power and set of tokens.

Example Walkthrough

Let's say our initial power level is 20, and we have a set of tokens with the following values: [4, 2, 10, 8].

1. **Sort the Tokens**: We sort the tokens to get [2, 4, 8, 10].
2. **Initialize Pointers and Variables**: We set `i = 0`, `j = 3` (since there are 4 tokens, and indexes start at 0), `ans = 0` for the maximum score, and `t = 0` for the current score.
3. **Main Loop**:
   - With 20 power, we start the loop. `i = 0`, `j = 3`.
4. **Playing Tokens Face Up**:
   - We have enough power for the token at `i = 0` (value = 2). We play this token face up.
   - Power is now 18 (20 - 2), `t = 1`, and `i = 1`, `ans` is updated to 1 since it's the maximum score till now.
   - Next, token at `i = 1` (value = 4). Play it face up.
   - Power is now 14 (18 - 4), `t = 2`, and `i = 2`. Update `ans` to 2.
   - Now, token at `i = 2` (value = 8). Play it face up.
   - Power is now 6 (14 - 8), `t = 3`, and `i = 3`. Update `ans` to 3.
5. **Insufficient Power to Play Face Up**:
   - Token at `i = 3` is worth 10, but power is only 6, so we can't play it face up.
6. **Playing Tokens Face Down**:
   - We have a score (`t == 1`), so we can sacrifice one score to regain power.
   - Play token at `j = 3` face down. Gain 10 power.
   - Power is now 16 (6 + 10), `t = 2` (as we lose one score), and `j = 2` (since token at `j = 3` has been played).
   - Now, we have enough power to play the token at `i = 3` which is still at value 10.
   - Play it face up. Power is now 6 (16 - 10), `t = 3`, and `i` moves beyond `j`, meaning we've played all tokens we can play face up.
   - Update `ans` to 3 (it still remains the maximum score).
7. **Result**:
   - We have played all the tokens we could with the power we had, maximizing the score.
   - The resulting maximum score `ans` is 3.

This walkthrough demonstrates the implementation of the solution approach using a simple example. The player expertly navigates through the tokens, deciding when to increase the score and when to regain power, ultimately achieving the highest score possible given the initial conditions.

Python Solution

```python
from typing import List

class Solution:
    def bag_of_tokens_score(self, tokens: List[int], power: int) -> int:
        # Sort the tokens list to play them in increasing order of value.
        tokens.sort()

        # Initialize two pointers and a current score.
        # i is the start pointer, j is the end pointer.
        left_index, right_index = 0, len(tokens) - 1
        max_score = current_score = 0

        # Loop until the start pointer is less than or equal to the end pointer.
        while left_index <= right_index:
            # If there is enough power to play the smallest token.
            if power >= tokens[left_index]:
                # Spend power to gain a score.
                power -= tokens[left_index]
                left_index += 1
                current_score += 1
                # Update max_score to be the highest score achieved so far.
                max_score = max(max_score, current_score)
            # If not enough power and there is score to spend,
            elif current_score > 0 and left_index < right_index:
                # Use a score to get token to get more power.
                power += tokens[right_index]
                right_index -= 1
                current_score -= 1
            else:
                # No moves left, break loop.
                break

        # Return the maximum score achieved.
        return max_score
```

Java Solution

```java
class Solution {
    public int bagOfTokensScore(int[] tokens, int power) {
        // Sort the tokens array to prioritize the lowest cost tokens first.
        Arrays.sort(tokens);
        // Initialize pointers for the two ends of the array.
        int low = 0, high = tokens.length - 1;
        // Initialize the maximum score and the current score (tokens turned into points).
        int maxScore = 0, currentScore = 0;

        // Continue as long as the low pointer does not cross the high pointer.
        while (low <= high) {
            // If we have enough power to buy the next cheapest token, do it.
            if (power >= tokens[low]) {
                power -= tokens[low++]; // Buy the token and increase the low pointer.
                currentScore++; // Increase the score by 1 for each token bought.
                // Update the max score if the current score is greater.
                maxScore = Math.max(maxScore, currentScore);
            }
            // If the power is not enough to buy, but we have some score (points),
            // we can sell the most expensive token.
            else if (currentScore > 0) {
                power += tokens[high--]; // Sell the token and decrease the high pointer.
                currentScore--; // Decrease the score by 1 for each token sold.
            }
            // If it's not possible to do either, exit the loop.
            else {
                break;
            }
        }
        // Return the maximum score achieved.
        return maxScore;
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    // This function calculates the maximum score we can achieve by playing tokens.
    // 'tokens' is a list of token values and 'power' is the initial power.
    int bagOfTokensScore(vector<int>& tokens, int power) {
        // Sort the tokens to facilitate the playing strategy.
        sort(tokens.begin(), tokens.end());

        // Initialize left and right pointers for the tokens array.
        int left = 0;
        int right = tokens.size() - 1;

        // 'maxScore' will hold the maximum score we can achieve.
        // 'currentScore' will keep track of the current score while playing.
        int maxScore = 0;
        int currentScore = 0;

        // Play until either end of the token array is reached.
        while (left <= right) {
            // If we have enough power to play the smallest token, play it to gain score.
            if (power >= tokens[left]) {
                power -= tokens[left++];  // Reduce power by the value of the played token.
                currentScore++;           // Increase the score as we've played a token.
                maxScore = max(maxScore, currentScore);  // Update maxScore if it's less than currentScore.
            // If we don't have enough power but have some score, we can trade the largest token for power.
            } else if (currentScore > 0) {
                power += tokens[right--];  // Increase power by the value of the traded token.
                currentScore--;            // Decrease the score as we've traded a token for power.
            // If we neither have the power to play nor score to trade, we cannot proceed further.
            } else {
                break;
            }
        }

        // Return the maximum score achieved.
        return maxScore;
    }
};
```

Typescript Solution

```typescript
// Importing the sort function to be able to sort an array
import { sort } from 'some-sorting-package';

// The function calculates the maximum score achievable by playing tokens.
// tokens: an array of token values; power: the initial power.
function bagOfTokensScore(tokens: number[], power: number): number {
    // Sort the tokens to facilitate the playing strategy.
    sort(tokens);

    // Initialize left and right pointers for the tokens array.
    let left = 0;
    let right = tokens.length - 1;

    // maxScore will hold the maximum score we can achieve.
    // currentScore will keep track of the current score while playing.
    let maxScore = 0;
    let currentScore = 0;

    // Play until either end of the token array is reached.
    while (left <= right) {
        // If we have enough power to play the smallest token, play it to gain a score.
        if (power >= tokens[left]) {
            power -= tokens[left++];  // Reduce power by the value of the played token.
            currentScore += 1;  // Move to the next token.
            currentScore += 1;  // Increase the score as we've played a token.
            maxScore = Math.max(maxScore, currentScore);  // Update maxScore if it's less than currentScore.
        } else if (currentScore > 0) {
            // If we don't have enough power but have some score, we can trade the largest token for power.
            power += tokens[right];  // Increase power by the value of the traded token.
            right -= 1;  // Move past the traded token.
            currentScore -= 1;  // Decrease the score as we've traded a token for power.
        } else {
            // If we neither have the power to play nor score to trade, we cannot proceed further.
            break;
        }
    }

    // Return the maximum score achieved.
    return maxScore;
}
```

Time and Space Complexity

Time Complexity

The time complexity of this code is mainly due to the sorting operation and the while loop that iterates through the list of tokens.

- Sorting the list of tokens takes $O(n \log n)$ time, where n is the number of tokens.
- The while loop runs in $O(n)$ time since each token is considered at most once when either increasing or decreasing the `t` (score) or `power`. The loop will end once we've iterated through all the tokens, or we cannot make any more moves.

Hence, the overall time complexity of the code is $O(n \log n)$ due to the sorting step.

Space Complexity

The space complexity of the code is $O(1)$. Apart from the input list, there are only a constant number of integer variables (`i`, `j`, `ans`, `t`, `power`) being used, regardless of the input size. Sorting is done in-place, which doesn't require extra space proportional to the input size.

Therefore, the extra space used by the program is constant, leading to a space complexity of $O(1)$.