## 948. Bag of Tokens

Two Pointers

Sorting

**Problem Description** 

Medium

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

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

first to maximize score efficiently.

and the count of tokens played.

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. **Sorting the Tokens:** Begin by sorting the tokens in ascending order. This allows us to play the tokens with the lowest values

- **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.
- Playing Tokens Face Up: While there is enough power to play the next lowest value token, do so. This increases the score
- Maximizing Score: Keep track of the maximum score achieved after each play to ensure the result reflects the highest score possible throughout the game.
- 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.
- 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.

we can access the smallest and largest values quickly.

decide whether to play a token face up or face down.

breaks as no further plays are possible.

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

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: Sort the Tokens: First, the list of tokens is sorted using Python's built-in sort() method. This organizes the tokens so that

**Example Walkthrough** 

Solution Approach

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.

Initialize Pointers and Variables: Two pointers, i for the lowest value tokens (start of the list) and j for the highest value

Main Loop: The while loop continues as long as there are tokens left to play (i <= j). Inside the loop, we have conditions to

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.

guaranteeing the maximum score possible with the given initial power and set of tokens.

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).

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),

Insufficient Resources: If we have insufficient power to play a token face up and no score to play a token face down, the loop

- **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,
- Let's say our initial power level is 20, and we have a set of tokens with the following values: [4, 2, 10, 8]. Sort the Tokens: We sort the tokens to get [2, 4, 8, 10].

Initialize Pointers and Variables: We set i = 0, j = 3 (since there are 4 tokens, and indexes start at 0), ans = 0 for the

Main Loop:

maximum score, and t = 0 for the current score.

• With 20 power, we start the loop. i = 0, j = 3.

 $\circ$  Next, token at i = 1 (value = 4). Play it face up.

Now, token at i = 2 (value = 8). Play it face up.

Play token at j = 3 face down. Gain 10 power.

Insufficient Power to Play Face Up:

score possible given the initial conditions.

Playing Tokens Face Down:

 $\circ$  Power is now 14 (18 - 4), t = 2, and i = 2. Update ans to 2.

 $\circ$  Power is now 6 (14 - 8), t = 3, and i = 3. Update ans to 3.

 $\circ$  Token at i = 3 is worth 10, but power is only 6, so we can't play it face up.

 $\circ$  We have a score (t >= 1), so we can sacrifice one score to regain power.

 $\circ$  Now, we have enough power to play the token at i = 3 which is still at value 10.

• We have played all the tokens we could with the power we had, maximizing the score.

 $\circ$  Power is now 16 (6 + 10), t = 2 (as we lose one score), and j = 2 (since token at j = 3 has been played).

Playing Tokens Face Up:  $\circ$  We have enough power for the token at i = 0 (value = 2). We play this token face up.  $\circ$  Power is now 18 (20 - 2), t = 1, and i = 1. ans is updated to 1 since it's the maximum score till now.

 $\circ$  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).

Result:

**Python** 

class Solution:

from typing import List

tokens.sort()

- The resulting maximum score ans is 3.
- Solution Implementation

def bag of tokens score(self, tokens: List[int], power: int) -> int:

# If there is enough power to play the smallest token.

# If not enough power and there is score to spend,

# then play the largest token to get more power.

# Initialize two pointers and a current score.

left index, right index = 0, len(tokens) - 1

max\_score = current\_score = 0

while left index <= right index:</pre>

elif current score > 0:

right index -= 1

current\_score -= 1

# Return the maximum score achieved.

if (power >= tokens[low]) {

else if (currentScore > 0) {

// Return the maximum score achieved.

if power >= tokens[left index]:

power += tokens[right\_index]

# No moves left, break loop.

public int bagOfTokensScore(int[] tokens. int power) {

// Initialize pointers for the two ends of the array.

// Sort the tokens array to prioritize the lowest cost tokens first.

// If we have enough power to buy the next cheapest token, do it.

// Update the max score if the current score is greater.

// 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) {

// Initialize left and right pointers for the tokens array.

// currentScore will keep track of the current score while playing.

// If we have enough power to play the smallest token, play it to gain a score.

currentScore += 1: // Increase the score as we've played a token.

power -= tokens[left]; // Reduce power by the value of the played token.

power += tokens[right]; // Increase power by the value of the traded token.

currentScore -= 1; // 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.

maxScore = Math.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.

// maxScore will hold the maximum score we can achieve.

// Play until either end of the token array is reached.

left += 1; // Move to the next token.

right -= 1: // Move past the traded token.

def bag of tokens score(self, tokens: List[int], power: int) -> int:

# If there is enough power to play the smallest token.

max score = max(max score, current score)

# Initialize two pointers and a current score.

# i is the start pointer, i is the end pointer.

left index, right index = 0, len(tokens) - 1

# Spend power to gain a score.

power -= tokens[left\_index]

# No moves left, break loop.

# Return the maximum score achieved.

# Sort the tokens list to play them in increasing order of value.

# Loop until the start pointer is less than or equal to the end pointer.

# Update max score to be the highest score achieved so far.

maxScore = Math.max(maxScore, currentScore);

// If it's not possible to do either, exit the loop.

// we can sell the most expensive token.

power -= tokens[low++]; // Buy the token and increase the low pointer.

power += tokens[high--]; // Sell the token and decrease the high pointer.

currentScore++; // Increase the score by 1 for each token bought.

// If the power is not enough to buy, but we have some score (points),

currentScore--; // Decrease the score by 1 for each token sold.

# i is the start pointer, i is the end pointer.

# Sort the tokens list to play them in increasing order of value.

# Loop until the start pointer is less than or equal to the end pointer.

# Spend power to gain a score. power -= tokens[left\_index] left index += 1current score += 1 # Update max score to be the highest score achieved so far. max score = max(max score, current score)

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

```
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.
```

Java

C++

public:

class Solution {

let left = 0:

let maxScore = 0;

} else {

return maxScore;

from typing import List

tokens.sort()

class Solution:

break;

let currentScore = 0;

while (left <= right) {</pre>

let right = tokens.length - 1;

if (power >= tokens[left]) {

} else if (currentScore > 0) {

// Return the maximum score achieved.

max\_score = current\_score = 0

while left index <= right index:</pre>

left index += 1

current score += 1

if power >= tokens[left index]:

class Solution {

else:

break

return max\_score

Arrays.sort(tokens);

while (low <= high) {</pre>

else {

return maxScore;

break;

```
// 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;
        // Plav until either end of the token array is reached.
        while (left <= right) {</pre>
            // 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
// 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);
```

```
# If not enough power and there is score to spend,
# then play the largest token to get more power.
elif current score > 0:
    power += tokens[right_index]
    right index -= 1
    current_score -= 1
```

break

Time and Space Complexity

else:

return max\_score

**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.

• 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

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

loop will end once we've iterated through all the tokens, or we cannot make any more moves.

Sorting the list of tokens takes O(n log n) time, where n is the number of tokens.

**Space Complexity** The space complexity of the code is 0(1). Apart from the input list, there are only a constant number of integer variables (i, j, 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 0(1).