2600. K Items With the Maximum Sum

Easy Greedy Math

Problem Description

In this LeetCode challenge, we are dealing with a theoretical bag full of items, each marked with a number: `1`, `0`, or `-1`. We're given specific counts for each type of number, which are:

- `numOnes`: the count of items with `1` written.
- `numZeros`: the count of items with `0` written.
- `numNegOnes`: the count of items with `-1` written.

Our goal is to choose exactly `k` items from the bag such that the sum of numbers on those chosen items is maximized. The problem asks us to find out what that maximum sum could be.

It's worth noting that because `0`s do not contribute to the sum, they only affect our ability to reach the exact count of `k` items without altering the sum. Also, adding `-1`s would decrease the sum, and so we would only want to add `-1`s if we have to reach `k` items and have no other options.

Intuition

For the given solution approach, here's the intuition:

1. **If we have enough `1`s to fill our quota of `k` items**, then we can simply take `k` items all of which are `1`s. This gives us the highest possible sum, which is `k` (since each item adds `1` to the sum).
2. **If we don't have enough `1`s**, we must use `0`s and possibly `-1`s to reach `k` total items. The `0`s don't affect the sum, so we'll pick as many of them as possible.
3. **Once we've exhausted the `1`s and `0`s**, if we still haven't reached the `k` items, we have no choice but to include `-1` items. Each `-1` item we include will decrease our sum by `1`.

In code:

- We first check if `numOnes >= k`. If yes, we just return `k` because we can fill our selection with just `1`s and that would give us the maximum sum possible.
- When `numOnes` is not enough, we add in `0`s to reach `k`. If `numZeros >= k - numOnes`, we can fill the rest with `0`s. Hence, the sum will be just `numOnes`.
- If we still haven't reached `k` items after using all the `1`s and `0`s, we must use `-1`s. Here, we subtract the deficit from `numOnes`, which represents having to use `(k - numOnes - numZeros)` negative ones.

Therefore, the trick to solving this problem lies in understanding that `1`s add to the sum, `0`s don't affect the sum, and `-1`s subtract from the sum—and you always want to minimize the number of `-1`s you're forced to choose.

Solution Approach

The solution provided uses a simple conditional logic, which involves no complex algorithms or data structures. It is a straightforward iterative approach based on the business rules outlined by the problem statement. Let's break down the solution approach provided:

- **When `numOnes >= k`**: This is the simplest case. If we have equal or more than `k` items with `1` written on them, we take `k` of them. This is because `1` contributes to the highest possible individual value for an item in the bag. The sum here is the maximum we can obtain, which is simply the value `k` because taking `k` ones will give us a sum of `k`.
- **When `numOnes` is not enough, `numZeros >= k - numOnes`**: The second condition kicks in if we don't have enough ones to reach `k`. We'll then use zeros to fill the remaining spots as they do not detract from the sum. If we have enough zeros to fill the remaining spots up to `k`, the sum remains `numOnes`. This is because `0` has no effect on the cumulative sum.
- **When we need to use `-1`s, `numZeros + numOnes < k`**: In the case where even our zeros are exhausted, and we haven't reached the count of `k`, we are forced to use items with `-1`. The count of `-1`s used is `(k - numOnes - numZeros)`, and as each `-1` subtracts `1` from the sum, we deduct this number from `numOnes`. This scenario gives us a sum of `numOnes - (k - numOnes - numZeros)` which would be less than `numOnes` by exactly as many `-1`s as we had to include.

The patterns used in the solution are:

- **Greedy approach**: By always taking `1`s where possible, the solution aligns with greedy algorithms' principle of making the locally optimal choice at each stage with the hope of finding a global optimum.
- **Conditional branching**: Using simple if-else logic helps navigate through the decisions based on the business rules defined.

The solution is efficient and effective due to the simplicity of the problem—it's a problem that doesn't require optimization techniques or complex data structures, as it boils down to elementary arithmetic operations driven by a few if-else statements. The time complexity here is O(1), meaning it requires a constant time to run, regardless of the values of `numOnes`, `numZeros`, `numNegOnes`, or `k`. This is because there are no loops or recursive calls in the code.

Example Walkthrough

Let's suppose we have the following counts of items marked with `1`, `0`, and `-1`, and we want to maximize the sum by choosing `k` items:

- `numOnes = 4` (items with `1`)
- `numZeros = 3` (items with `0`)
- `numNegOnes = 2` (items with `-1`)
- `k = 6` (number of items to choose)

We want to choose `k=6` items to maximize the sum. Let's apply the solution approach:

1. The first check is whether we have enough `1`s to fulfill the `k` items. Here, `numOnes` (4) is less than `k` (6), so we don't have enough `1`s.
2. Since we don't have enough `1`s, we look at `0`s. We have `numZeros = 3`, which is more than we need to top up to `k` because `k - numOnes = 6 - 4 = 2` and `numZeros >= 2`. Hence, we can fill up to `k` using all `1`s and some `0`s without needing to use any `-1`s. Thus, we can ignore `-1`s, given they will decrease our sum.
3. The sum is determined by the `1`s we can use because `0`s do not contribute to the sum. We have `numOnes = 4` `1`s, and we are using `4` `1`s and `2` `0`s (since we need 6 items and we take as many `1`s as available and the rest are `0`s).
4. After choosing `4` `1`s and `2` `0`s, the sum of the chosen items is `4` because `0`s do not contribute, and we have avoided using any `-1`s.

Therefore, the maximum sum we can achieve by choosing `k=6` items from this bag is `4`.

Solution Implementation

Python

```python
class Solution:
    def kItemsWithMaximumSum(self, num_positives: int, num_zeros: int, num_negatives: int, target_count: int) -> int:
        # The function calculates the maximum sum of 'target_count' items
        # considering there are 'num_positives' +1s, 'num_zeros' 0s, and 'num_negatives' -1s.

        # If the number of +1s is greater than or equal to the target count k,
        # the maximum sum we can get is k (since +1 contributes the maximum to the sum).
        if num_positives >= target_count:
            return target_count

        # If we don't have enough +1s, we can include all the +1s and then complement them with 0s.
        # Since 0s do not contribute to the sum, if the remaining number (k - number of +1s) is less than
        # or equal to the number of 0s, we can fill the rest with 0s. The sum remains equal to the count of +1s.
        if num_zeros >= target_count - num_positives:
            return num_positives

        # If we run out of +1s and 0s, we will have to include -1s to reach the target count.
        # Each -1 will reduce the sum by 1. Therefore, we subtract the count of extra -1s needed
        # from the total number of +1s we had.
        return num_positives - (target_count - num_positives - num_zeros)

# Example usage:
# Create a Solution object
solution = Solution()
# Call the method with a hypothetical case
max_sum = solution.kItemsWithMaximumSum(4, 2, 3, 5)
print(max_sum)  # Expected output: 4
```

Java

```java
class Solution {

    // Function to find the maximum sum of `k` items consisting of 1s, 0s, and -1s.
    public int kItemsWithMaximumSum(int numOnes, int numZeros, int numNegOnes, int k) {
        // If the count of 1s is itself greater than or equal to k, then the sum is k
        // since selecting k 1s gives you the maximum sum.
        if (numOnes >= k) {
            return k;
        }

        // If after selecting all 1s, the count of 0s is enough to reach k,
        // we return the count of 1s because adding 0s does not change the sum.
        if (numZeros >= (k - numOnes)) {
            return numOnes;
        }

        // If there aren't enough 1s and 0s to reach k, then we have to include some -1s.
        // Every -1 included will reduce the sum by 1. Hence, we subtract the number
        // of -1s to be included from the count of 1s to get the final sum.
        return numOnes - (k - numOnes - numZeros);
    }
}
```

C++

```cpp
class Solution {
public:
    // Function to calculate the maximum sum of 'k' elements from a set containing
    // 'num_positives' number of 1's, 'num_zeros' number of 0's, and 'num_negatives' number of -1's
    int kItemsWithMaximumSum(int num_positives, int num_zeros, int num_negatives, int k) {
        // If the count of positive numbers (1's) is greater than or equal to k,
        // we can take all k items as 1's to get the maximum sum which is k.
        if (num_positives >= k) {
            return k;
        }

        // If the count of zeros and positives combined is enough to fill all k items,
        // no negatives will be included, so the sum is just the number of positive numbers.
        if (num_zeros >= k - num_positives) {
            return num_positives;
        }

        // If we don't have enough zeros to fill the gap between the number of positives
        // and k, we will have to include negative numbers which reduce the overall sum.
        // The maximum sum in this case is reduced by the number of negatives that are used,
        // which is the total count minus the count of positives and zeros.
        return num_positives - (k - num_positives - num_zeros);
    }
};
```

TypeScript

```typescript
/**
 * Function to find maximum sum of 'k' elements from a set of 1s, 0s, and -1s.
 * @param {number} onesCount - The number of ones in the set.
 * @param {number} zerosCount - The number of zeros in the set.
 * @param {number} negOnesCount - The number of negative ones in the set.
 * @param {number} k - The number of elements to sum up for the maximum sum.
 * @returns {number} The maximum sum of 'k' elements.
 */
function kItemsWithMaximumSum(
    onesCount: number,
    zerosCount: number,
    negOnesCount: number,
    k: number,
): number {
    // If the number of ones is greater than or equal to k, the maximum sum is k.
    if (onesCount >= k) {
        return k;
    }

    // If the number of zeros is enough to fill the gap after taking all ones, the sum stays the same.
    if (zerosCount >= k - onesCount) {
        return onesCount;
    }

    // If there are not enough zeros, then we must include some negative ones.
    // The sum decreases as we take each negative one.
    return onesCount - (k - onesCount - zerosCount);
}
```


```python
class Solution:
    def kItemsWithMaximumSum(self, num_positives: int, num_zeros: int, num_negatives: int, target_count: int) -> int:
        # The function calculates the maximum sum of 'target_count' items
        # considering there are 'num_positives' +1s, 'num_zeros' 0s, and 'num_negatives' -1s.

        # If the number of +1s is greater than or equal to the target count k,
        # the maximum sum we can get is k (since +1 contributes the maximum to the sum).
        if num_positives >= target_count:
            return target_count

        # If we don't have enough +1s, we can include all the +1s and then complement them with 0s.
        # Since 0s do not contribute to the sum, if the remaining number (k - number of +1s) is less than
        # or equal to the number of 0s, we can fill the rest with 0s. The sum remains equal to the count of +1s.
        if num_zeros >= target_count - num_positives:
            return num_positives

        # If we run out of +1s and 0s, we will have to include -1s to reach the target count.
        # Each -1 will reduce the sum by 1. Therefore, we subtract the count of extra -1s needed
        # from the total number of +1s we had.
        return num_positives - (target_count - num_positives - num_zeros)

# Example usage:
# Create a Solution object
solution = Solution()
# Call the method with a hypothetical case
max_sum = solution.kItemsWithMaximumSum(4, 2, 3, 5)
print(max_sum)  # Expected output: 4
```


Time and Space Complexity

Time Complexity

The time complexity of the given code snippet is `O(1)`. This is because all operations in the `kItemsWithMaximumSum` method consist of simple arithmetic comparisons and subtractions which are performed in constant time, regardless of the input sizes of `numOnes`, `numZeros`, `numNegOnes`, and `k`.

Space Complexity

The space complexity of the code is also `O(1)`. The function only uses a fixed amount of extra space for a few integer variables to hold the inputs and perform computations. There are no data structures like arrays or lists that would grow with the size of the input; therefore, the amount of space used does not scale with the input size.