740. Delete and Earn

Medium | Array | Hash Table | Dynamic Programming

Problem Description

In this problem, you're given an array of integers called `nums`, where each element represents points you can earn. The goal is to accumulate the highest number of points possible by repeatedly performing a specific operation.

The operation consists of the following steps:

1. Choose any element from the array (`nums[i]`) and earn points equal to its value (`nums[i]` points).
2. Once you've earned points from `nums[i]`, you must then delete every element from the array that is either 1 less (`nums[i] - 1`) or 1 more (`nums[i] + 1`) than the chosen element.

You can repeat this operation as many times as you wish in order to maximize your points. The challenge resides in selecting the elements in an order that prevents you from eliminating potential points that could have been earned later on.

The task is to determine the maximum number of points you can earn by applying the operation optimally.

Intuition

The key observation to solve this problem is recognizing that if you choose any number, you should take all instances of that number because choosing one instance will force you to delete the others anyway.

To approach this, you can leverage dynamic programming. The first step is to create an array, `sums`, to accumulate the total sum of points that each unique number in `nums` can contribute. Essentially, for each value `i`, `sums[i]` holds the total points from all occurrences of `i` in `nums`.

Now, since you cannot pick numbers adjacent to each other (i.e., `nums[i]`, `nums[i]-1`, and `nums[i]+1` are mutually exclusive choices), you should maintain two states:

- `select[i]`: the maximum sum you can obtain if you choose to take `i`.
- `nonSelect[i]`: the maximum sum you can obtain if you decide not to take `i`.

The state transitions work as follows:

- If you take the number `i`, you couldn't have taken `i - 1`, so your current maximum if you select `i` is the maximum of not selecting `i - 1` plus the sum of `i`.
- If you don't take `i`, the maximum is the larger of the previous maxima regardless of whether `i - 1` was selected or not.

The relationship can then be defined as:

```
select[i] = nonSelect[i - 1] + sums[i]
nonSelect[i] = max(select[i - 1], nonSelect[i - 1])
```

The solution iterates over the values, updating `select` and `nonSelect` based on the points from `sums`. The final solution will be the maximum value between choosing and not choosing the last element in the array.

The given Python solution implements this using a simplified dynamic programming approach with a single array, `total`, which plays the role of `sums`, and two variables, `first` and `second`, to keep track of the `select` and `nonSelect` states for the last two processed numbers. The approach ensures that the solution is derived efficiently with optimal space usage.

Solution Approach

The solution follows a bottom-up dynamic programming approach where the operation of choosing a number encompasses a couple of algorithms and data structures to reach an optimal solution.

1. **Pre-processing step**: We go through the `nums` array to find the maximum number (`mx`) among all the numbers, as it will determine the size of the `total` array, which is analogous to the `sums` array in the reference approach. This `total` array holds the aggregate points that each number contributes.

2. **Calculating total points for each number**:
   - We initialize the `total` array with the size `mx + 1` (since arrays are 0-indexed).
   - We iterate through the `nums` array once again to sum the points for each number. For each `num` in `nums`, we add `num` to `total[num]`, which is adding the points for each occurrence of that number.
3. **Dynamic Programming (DP) State Transition**:
   - We define two states, `first` and `second`. Respectively, they correspond to `nonSelect` and `select` for two consecutive elements being processed in a DP manner. In the reference approach, `select` and `nonSelect` arrays are used, while in the solution code we only use two variables for space optimization.
   - Initialize these variables as follows:
     - `first` = `total[0]`, as this is the point contribution for the number 0.
     - `second` = `max(total[0], total[1])`, as this is the maximum between choosing 0 and choosing 1 but not choosing 0.
4. **Iterative Optimization**:
   - We start iterating from 2 to `mx` (inclusive) to update the `first` and `second` states:
     - For each number `i`, the current maximum points `cur` when choosing the number `i` is computed as `max(first + total[i], second)` which follows the transition:
       - If we choose `i`, we add the points from choosing `i` (`total[i]`) to the maximum points without including `i-1` (`first`). This reflects the concept that if we are taking number `i`, we must have not taken `i-1`.
       - If we don't choose `i`, the maximum stays at `second` which is the larger of the previous maxima irrespective of whether `i-1` was chosen or not.
     - We then update `first` and `second` for the next iteration:
       - `first` gets the value of `second`.
       - `second` gets the value of `cur`.
5. **Returning the Result**:
   - After the loop, `second` will hold the maximum number of points that can be achieved from `0` to `mx`, as it either includes or excludes the last number. Therefore, we return `second` as the solution.

By employing this dynamic programming technique, the problem avoids brute-force repetitions and overcomes the potential exponential time complexity we would face if we tried every possible combination of elements to delete from the `nums` array. This implementation ensures that the solution is reached in O(n + k) time, where 'n' is the length of `nums` and 'k' is the range of numbers in `nums`.

Example Walkthrough

Let's consider a small input array `nums` containing the following elements: `[3, 4, 2, 2, 3, 4]`.

1. **Pre-processing step**:
   - We find the maximum number in `nums`, which is `4`.
2. **Calculating total points for each number**:
   - We initialize the `total` array with a length of `5` (0-indexed, so we go from `0` to `4`).
   - By iterating through `nums`, we calculate the `total` as follows:
     - `total[2]` would be `4` (because there are two `2's` and `2*2=4`).
     - `total[3]` would be `6` (two `3's`, so `3*2=6`).
     - `total[4]` would be `8` (two `4's`, so `4*2=8`).
3. **Dynamic Programming (DP) State Transition**:
   - Initialize `first = total[0]` which is `0` (no contribution from the number 0).
   - Initialize `second = max(total[0], total[1])` which is `max(0, 0)` because the number `1` does not appear in `nums`. Thus, `second` is `0`.
4. **Iterative Optimization**:
   - For `i=2`, we calculate `cur = max(first + total[2], second)` which is `max(0+4, 0)` resulting in `cur = 4`. Update `first` to `0` (previous `second`) and `second` to `4`.
   - For `i=3`, we calculate `cur = max(first + total[3], second)` which is `max(0+6, 4)` resulting in `cur = 6`. Update `first` to `4` and `second` to `6`.
   - For `i=4`, we calculate `cur = max(first + total[4], second)` which is `max(4+8, 6)` resulting in `cur = 12`. Update `first` to `6` and `second` to `12`.
5. **Returning the Result**:
   - The loop ends with `second` being `12`, which signifies that by following the described selection process, we have maximally earned `12` points.

Therefore, for the array `[3, 4, 2, 2, 3, 4]`, the maximum number of points that can be earned is `12`.

Python Solution

```python
from typing import List

class Solution:
    def deleteAndEarn(self, nums: List[int]) -> int:
        # Find the maximum value in nums array
        max_value = max(nums)

        # Create a list to store the total points for each number
        total_points = [0] * (max_value + 1)

        # Fill the total_points list where the index represents the number
        # and the value at that index is the total points that can be earned from that number
        for num in nums:
            total_points[num] += num

        # Initialize first and second variables to store the
        # total points earned up to the previous and current positions
        earn_prev_prev = total_points[0]
        earn_prev = max(total_points[0], total_points[1])

        # Iterate over the total_points list starting from the second index
        for i in range(2, max_value + 1):
            # Calculate the current max points by either taking the current number
            # and the points from two steps before, or the points from the previous step
            current_max = max(earn_prev_prev + total_points[i], earn_prev)

            # Update the points from two steps before and the previous step
            earn_prev_prev = earn_prev
            earn_prev = current_max

        # The last 'earn_prev' contains the maximum points that can be earned
        return earn_prev

```

Java Solution

```java
public class Solution {

    // Method to calculate the maximum points you can earn by deleting elements
    public int deleteAndEarn(int[] nums) {
        // Return 0 if the array is empty
        if (nums.length == 0) {
            return 0;
        }

        // Create an array to store the sum of points for each number
        int[] valueSums = new int[10010];
        // Create an array to store the maximum points if we select the current number
        int[] dpSelect = new int[10010];
        // Create an array to store the maximum points if we do not select the current number
        int[] dpNonSelect = new int[10010];

        // Variable to store the maximum value in nums
        int maxValue = 0;
        // Populate the valueSums array and find the maximum value
        for (int num : nums) {
            valueSums[num] += num;
            maxValue = Math.max(maxValue, num);
        }

        // Dynamic programming to decide whether to select or not select a particular number
        for (int i = 1; i <= maxValue; i++) {
            // If we select i, we can't use i-1, so we add the points of i to dpNonSelect[i-1]
            dpSelect[i] = dpNonSelect[i - 1] + valueSums[i];
            // If we don't select i, take the maximum points from the previous selection or non-selection
            dpNonSelect[i] = Math.max(dpSelect[i - 1], dpNonSelect[i - 1]);
        }
        // The result is the max points of selecting or not selecting the highest value
        return Math.max(dpSelect[maxValue], dpNonSelect[maxValue]);
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // The deleteAndEarn function.
    int deleteAndEarn(vector<int>& numbers) {
        // We create vals with size enough to cover all potential
        // input numbers, initializing with 0s. This will store the cumulative values.
        vector<int> cumulativeValues(10001, 0);

        // Populate cumulativeValues so that each index's value is the sum
        // of all the occurrences of that number in the numbers vector.
        for (int num : numbers) {
            cumulativeValues[num] += num;
        }

        // Now we use our rob function to find the maximum amount we can earn
        // following the delete and earn rule.
        return rob(cumulativeValues);
    }

    // The rob function uses dynamic programming to find the maximum earnable amount.
    int rob(vector<int>& values) {
        // Initialize the two variables to keep track of two states:
        // prevMax: the maximum amount we can get from [0...i-2]
        // currentMax: the maximum amount we can get from [0...i-1]
        int prevMax = 0, currentMax = values[0];

        for (int i = 1; i < values.size(); ++i) {
            // Calculate the new max amount that can be earned
            // including the current number or by excluding it.
            // c decides whether to take the current number or not.
            int tempMax = max(values[i] + prevMax, currentMax);

            // Move currentMax to prevMax for the next iteration, and
            // update currentMax to the newly calculated tempMax.
            prevMax = currentMax;
            currentMax = tempMax;
        }

        // At the end, currentMax will contain the maximum amount that
        // can be earned by either taking or skipping each number.
        return currentMax;
    }
};

```

Typescript Solution

```typescript
// Define an array to represent the cumulative values of numbers.
let cumulativeValues: number[] = new Array(10001).fill(0);

// The deleteAndEarn function, which takes an array of numbers and returns the maximum points that can be earned.
function deleteAndEarn(numbers: number[]): number {
    // Populate cumulativeValues so that each index's value is the sum
    // of all occurrences of that number in the input numbers array.
    for (let num of numbers) {
        cumulativeValues[num] += num;
    }

    // Use the rob function to find the maximum amount we can earn
    // following the delete and earn rule.
    return rob(cumulativeValues);
}

// The rob function uses dynamic programming to calculate the maximum earnable amount.
function rob(values: number[]): number {
    // Initialize variables to keep track of the two states:
    // prevMax stores the maximum amount we can get from [0...i-2]
    // currentMax stores the maximum amount we can get from [0...i-1]
    let prevMax = 0;
    let currentMax = values[0];

    for (let i = 1; i < values.length; i++) {
        // Calculate the tempMax which is the new maximum amount that can
        // be earned by including or excluding the current number.
        let tempMax = Math.max(values[i] + prevMax, currentMax);

        // Update prevMax to the previous currentMax for the next iteration,
        // and set currentMax to the newly calculated max (tempMax).
        prevMax = currentMax;
        currentMax = tempMax;
    }

    // currentMax will hold the maximum amount that can be earned
    // after considering all the numbers.
    return currentMax;
}

```

Time and Space Complexity

The provided code is designed to solve the problem by first calculating the maximum value in the list of numbers, creating an array that sums the same elements' values, and then using a dynamic programming approach similar to the house robber problem. The complexities are as follows:

Time Complexity

The time complexity of the algorithm is `O(N + M)`, where `N` is the length of the `nums` array and `M` is the maximum number in the `nums`. This is because:

1. The first `for` loop which calculates the maximum number, `mx`, iterates over all elements in `nums`, which takes `O(N)`.
2. The second `for` loop creates the `total` array by summing the value of each number where the `num` appears. This also takes `O(N)` time as it iterates over all elements in `nums`.
3. The third `for` loop iterates over the range from `2` to `mx`, to calculate the maximum points that can be earned without adjacent numbers. This runs in `O(M)` time, where `M` is the maximum number in `nums`.

Therefore, adding these up we get `O(N) + O(N) + O(M)` which simplifies to `O(N + M)` as the dominant terms.

Space Complexity

The space complexity of the algorithm is `O(M)`, where `M` is the maximum number in `nums`. This is due to:

1. The `total` array, which has a length of `mx + 1`, accounting for all numbers from `0` to `mx` inclusive.
2. Constant space for variables `first`, `second`, `cur`, and `mx`.

Therefore, the additional space used by the algorithm is predominated by the size of the `total` array.