2517. Maximum Tastiness of Candy Basket

Medium | Array | Binary Search | Sorting | Leetcode Link

Problem Description

In this problem, you have an array called `price` where each element `price[i]` represents the price of the `i-th` candy. There's also a positive integer `k`, which denotes the number of candies in a basket. Note that all candies in a basket must have distinct prices.

You need to form a basket with `k` distinct candies such that the difference between the lowest and highest price in the basket (referred to as the basket's "tastiness") is as large as possible. However, tastiness is actually defined as the *smallest* absolute difference you can find among any two prices in the basket.

Your task is to determine the maximum tastiness that a basket can achieve and return that value.

Intuition

To solve this problem, we use a binary search approach due to its sorted nature and the need to maximize a specific smallest value (tastiness). The key idea is to find the maximum value for the smallest possible price difference in the basket—this is the tastiness. We want to ensure that we can find `k` distinct candies with at least this tastiness value.

Here is how we approach the solution:

1. Since the candies must be distinct, we first sort the `price` array. This will help us easily identify distinct candies and ensure that we can check intervals properly for tastiness.
2. We then apply binary search on the potential values of tastiness (the smallest price difference). The lower bound (`l`) is 0 (when two candies have the same price, which is not allowed, but serves as our minimum starting point), and the upper bound (`r`) is the largest possible price difference (the price of the most expensive candy minus the price of the cheapest one).
3. The function `check` takes a value `x` and determines if it's possible to select `k` distinct candies such that each pair of selected candies has at least `x` difference in their prices. We iterate through the sorted prices, counting how many candies can be added to the basket while maintaining the condition related to `x`.
4. In the binary search loop, we check the middle of our current range (`mid`) to see if there's a valid basket with tastiness equal to `mid`. If it's possible, we know we can attempt higher values for tastiness, so we move our lower bound `l` up to `mid`. If it's not possible, we know we need to reduce our expectations of tastiness, so we adjust the upper bound `r` to `mid - 1`.
5. When the loop ends, the range has narrowed down to the highest value of `l` that satisfies our conditions, which is the maximum possible tastiness for the basket.
6. Finally, we return `l` as the answer, which now holds the largest minimum tastiness possible.

This strategy works because we leverage the sorted nature of the array and use binary search to efficiently converge on the largest minimum tastiness value achievable.

Solution Approach

Let's break down the implementation step-by-step to understand how the given solution uses binary search to find the tastiness of a basket:

1. **Sorting the Prices:** The `price` array is sorted at the very beginning of the solution. Sorting is crucial because for binary search to work, the elements need to be in a sorted order, and this step also supports checking for distinct candies efficiently.
2. **Binary Search Setup:** Two pointers `l` and `r` are initialized to frame the search space for the maximum tastiness. `l` is set to `0` because tastiness cannot be negative, and `r` is set to the difference between the highest and lowest price, i.e., `price[-1] - price[0]`.
3. **The `check` Function:** The `check` function is the heart of the binary search implementation. It takes a tastiness value `x` and checks if there are at least `k` candies in the sorted price list where the adjacent selected candies have at least `x` difference between them. The function keeps a count (`cnt`) of how many candies can be a part of the basket with the given tastiness and returns `True` or `False` based on whether we can have `k` candies or not.
4. **Executing the Binary Search:** We use a `while` loop to perform the binary search between `l` and `r`. Inside the loop, we calculate `mid`, which is the midpoint between `l` and `r`, using `(l + r + 1) >> 1` (bitwise shift right is equivalent to division by 2, but since we're using `(l + r + 1)`, it ensures proper rounding).
5. **Decision Making:** With the `mid` value, we call the `check` function. If `check(mid)` returns `True`, it means that `mid` is a valid tastiness value for a basket with `k` distinct candies, so we can attempt to find an even larger tastiness value by adjusting `l` to `mid`. If `check(mid)` returns `False`, we know that `mid` is too high of a tastiness value to maintain `k` distinct candies, so we bring `r` down to `mid - 1`.
6. **Narrowing Down and Result:** The loop continues until `l` < `r` is no longer true, meaning `l` and `r` converge to the maximum value that satisfies the condition that `k` distinct candies have at least `l` difference between each other in price.
7. **Returning the Solution:** Once the loop is concluded, `l` is the required maximum tastiness value, and it's returned as the solution.

By implementing binary search, the solution efficiently navigates through the potential tastiness values to find the optimal solution. The code is optimized to reduce the number of iterations needed to pinpoint the maximum tastiness, which is key given the potential size of the input array.

Example Walkthrough

To illustrate the solution approach, let's use a small example:

Suppose we have an array of candy prices given by `price = [1, 5, 9, 14]` and we want to form a basket with `k = 2` candies, aiming for the highest possible tastiness.

1. **Sorting the Prices:** Our array `price` is already sorted (`[1, 5, 9, 14]`), so this step is complete.
2. **Binary Search Setup:** We set `l = 0` and `r = 14 - 1 = 13`, the range within which we'll look for the maximum tastiness value.
3. **The `check` Function:** We would define our `check` function, which would check if we can select `k` candies with at least `x` difference in price between any two of them.
4. **Executing the Binary Search:** With `l = 0` and `r = 13`, we proceed with binary search. The first `mid` value to check is `(0 + 13 + 1) >> 1 = 7`. We need to see if it's possible to pick 2 candies at least 7 units apart in price.
5. **Decision Making:** We call `check(7)`. Starting with the cheapest candy, we can choose `price[0] = 1`. Then, we see that `price[2] = 9`, which is 8 units apart from `price[0]`. Thus, we found a valid basket, `[1, 9]`, with tastiness 7 (since 9 - 1 = 8 and we needed at least 7). Since the `check` function returned `True`, we move the lower bound up: now, `l = 7`.

   The next `mid` is `(7 + 13 + 1) >> 1 = 10`. For `check(10)`, we can start again by choosing `price[0] = 1`. Then, we find that there is no other candy price at least 10 units apart from 1 in the given prices. Hence, `check(10)` returns `False`, and we lower the upper bound: now, `r = 10 - 1 = 9`.

   We continue the binary search. With `l = 7` and `r = 9`, `mid = (7 + 9 + 1) >> 1 = 8`. A `check(8)` call is `True` with the basket `[1, 9]`. Update `l` to `8`.

   Next, `mid = (8 + 9 + 1) >> 1 = 9`. `check(9)` is `False` as we can't find two prices at least 9 units apart that would satisfy `k = 2`.
6. **Narrowing Down and Result:** The binary search continues but the next iteration won't change the bounds because `l` (now 8) and `r` (now 8, after updating from 9) are equal, thus ending the loop. They converge to `8`, which is the tastiness level we can achieve with `k = 2`.
7. **Returning the Solution:** We would return `l`, which is `8`, as this is the maximum tastiness realized by a basket under the given constraints with `k` distinct candies.

Python Solution

```python
from typing import List

class Solution:
    def maximumTastiness(self, prices: List[int], target_count: int) -> int:
        def is_possible(minimum_distance: int) -> bool:
            # Initialize counter for number of cakes and the previous cake's price
            count, previous_price = 0, -minimum_distance

            # Iterate through sorted prices
            for current_price in prices:
                # If the distance from the previous selected cake price is
                # at least the minimum distance, select this cake
                if current_price - previous_price >= minimum_distance:
                    previous_price = current_price
                    count += 1

            # Check if we can select at least `target_count` number of cakes
            return count >= target_count

        # Sort the price list to enable binary search
        prices.sort()

        # Initialize binary search boundaries for the minimum distance
        left, right = 0, prices[-1] - prices[0]

        # Perform binary search to find the maximum minimum distance
        while left < right:
            # Compute the middle value between left and right
            middle = (left + right + 1) // 2

            # If it's possible to choose `target_count` cakes with at least
            # `middle` distance between them, move to the right half
            if is_possible(middle):
                left = middle
            else:
                # Otherwise, continue with the left half
                right = middle - 1

        # The maximum minimum distance will be `left` after binary search ends
        return left

```

Java Solution

```java
class Solution {

    /**
    * Finds the maximum difference between the tastiness values of any two selected items.
    *
    * @param prices Array representing the tastiness values of different items.
    * @param k The number of items to select.
    * @return The maximum difference possible between the tastiness values.
    */
    public int maximumTastiness(int[] prices, int k) {
        // Sort the array to make it easier to find the tastiness differences.
        Arrays.sort(prices);

        // Use binary search to find the maximum difference.
        int left = 0;
        int right = prices[prices.length - 1] - prices[0];

        // Perform the binary search.
        while (left < right) {
            int mid = (left + right + 1) / 2;

            // Check if the current difference can satisfy the condition.
            if (canSelectItems(prices, k, mid)) {
                left = mid; // If so, try a bigger difference.
            } else {
                right = mid - 1; // Otherwise, try a smaller difference.
            }
        }

        // The maximum tastiness difference that can be achieved.
        return left;
    }

    /**
    * Helper method to check if it's possible to select k items with at least x
    * difference between every pair of selected items.
    *
    * @param prices Array representing the tastiness values of different items.
    * @param k The number of items to select.
    * @param x The minimum difference required between any two selected items.
    * @return A boolean indicating whether it is possible to select the items.
    */
    private boolean canSelectItems(int[] prices, int k, int x) {
        int count = 0; // Tracks the number of items selected.
        int prevSelected = Integer.MIN_VALUE; // Stores the tastiness of the last selected item.

        // Iterate through the sorted prices array.
        for (int curTastiness : prices) {
            // Select the item if the difference with the last selected item is at least x.
            if (curTastiness - prevSelected >= x) {
                prevSelected = curTastiness; // Update the last selected item.
                count++; // Increment the count of items selected.
            }
        }

        // Return true if we can select at least k items, false otherwise.
        return count >= k;
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <algorithm>

class Solution {
public:
    // Function to find the maximum tastiness level.
    // The 'price' vector contains different prices and 'k' is the required number of tastiness levels.
    int maximumTastiness(vector<int>& price, int k) {
        // Sort the price vector in ascending order.
        std::sort(price.begin(), price.end());

        // Initialize left and right pointers for binary search.
        int left = 0;
        int right = price.back() - price.front();

        // Define the check function to check if a given difference could yield 'k' tastiness levels.
        auto check = [&](int diff) -> bool {
            int count = 0;       // Count of items with at least 'diff' difference.
            int prev = -diff;    // Initialize previous item position. Start with -'diff' to include the first item.
            for (int currentPrice : price) {
                // If the current and previous item's difference is at least 'diff'.
                if (currentPrice - prev >= diff) {
                    prev = currentPrice;  // Update the previous item position.
                    ++count;              // Increment the count.
                }
            }
            // Check if we can find at least 'k' items with at least 'diff' distance.
            return count >= k;
        };

        // Perform the binary search.
        while (left < right) {
            // Find the middle value to use as potential tastiness level.
            int mid = (left + right + 1) / 2;

            // Use the 'check' lambda function to decide whether to go left or right.
            if (check(mid)) {
                left = mid;  // If 'mid' works, we try to find a potentially larger difference.
            } else {
                right = mid - 1;  // If 'mid' doesn't work, we have to look for a smaller difference.
            }
        }

        // 'left' will contain the maximum difference that allows for at least 'k' items.
        return left;
    }
};

```

Typescript Solution

```typescript
function maximumTastiness(prices: number[], target: number): number {
    // Sort the prices array in non-decreasing order.
    prices.sort((a, b) => a - b);

    // Initialize left and right pointers for binary search.
    // Left pointer starts from 0, right pointer starts from the max difference.
    let left = 0;
    let right = prices[prices.length - 1] - prices[0];

    // Helper function to check if it's possible to pick `target` treats with a minimum difference of `minDiff`.
    const canPickTreats = (minDiff: number): boolean => {
        let count = 0;
        let previousPrice = -minDiff;

        // Iterate through the sorted prices array.
        for (const currentPrice of prices) {
            // If the current price is greater than or equal to the previous picked treat plus `minDiff`,
            // it means we can pick this treat.
            if (currentPrice - previousPrice >= minDiff) {
                previousPrice = currentPrice;
                count++;
            }
        }
        // Return true if we can pick at least `target` number of treats, otherwise false.
        return count >= target;
    };

    // Binary search to find the maximum tastiness using the canPickTreats function to guide the search.
    while (left < right) {
        // Calculate the mid-point with a bit-shift, equivalent to dividing by 2 and flooring the result.
        const mid = (left + right + 1) >> 1;

        // If we can pick the treats with at least `mid` difference, move the left pointer to mid.
        if (canPickTreats(mid)) {
            left = mid;
        } else {
            // Otherwise, move the right pointer to mid - 1.
            right = mid - 1;
        }
    }

    // The left pointer at the end will hold the maximum minimum tastiness achievable.
    return left;
}

```

Time and Space Complexity

The given Python code snippet is designed to find the maximum minimum distance (tastiness) between any two selected elements in the array `price`, while picking exactly `k` elements. It leverages binary search to find the solution efficiently. Here is the computational complexity analysis:

Time Complexity

The overall time complexity of the code is `O(n * log m)`, where `n` is the number of elements in the input list `price`, and `m` is the range of possible tastiness values, which is the difference between the minimum and maximum values in `price`.

1. `price.sort()`: The sorting of the input list has a time complexity of `O(n * log n)` since Python uses Timsort (a hybrid sorting algorithm derived from merge sort and insertion sort) for its sorting.
2. The binary search loop: The `while` loop runs in `O(log m)` time, where `m` is the difference between the maximum and minimum elements in the sorted `price` list. Because we are halving our search space in each iteration, the number of iterations needed is proportional to the logarithm of the range.
3. Inside the binary search loop, the `check` function is called. It iterates over the entire `price` list in `O(n)` time, as it potentially goes through all elements to count how many valid selections of tastiness can be made.

Combining the sorting step and the binary search steps, the time complexity becomes `O(n * log n) + O(log m) * O(n)`, which simplifies to `O(n * log n) + O(n * log m)` since `O(n * log m)` dominates `O(n * log n)`. Thus, the final time complexity is `O(n * log m)`.

Space Complexity

The space complexity of the code is `O(1)`.

1. The list is sorted in-place, not requiring additional space relative to the input size.
2. The `check` function uses constant space, only needing additional variables to store the count (`cnt`), previous selected element (`pre`), and iteratively checking each element (`cur`).
3. Since there are no additional data structures that grow with the input size, the space complexity is maintained at a constant level.

Therefore, the space complexity is `O(1)`, indicating that the space required does not increase with the size of the input list `price`.