2560. House Robber IV

Medium Array Binary Search

Leetcode Link

Problem Description

In this problem, we are considering a street lined with consecutive houses, each containing a certain amount of money. A robber aims to steal money from these houses, but with one condition: they are not willing to rob two adjacent houses. This constraint is dictated by the need to minimize the risk of getting caught. Additionally, there is a requirement that the robber must steal from at least `k` houses. The robber's "capability" is defined as the maximum amount of money stolen from a single house during the heist.

We are given an array of integers `nums`, where `nums[i]` represents the amount of money in the `i`th house. We also have an integer `k`, which represents the minimum number of houses from which the robber must steal.

Our task is to calculate the minimum "capability" of the robber to achieve the heist of at least `k` houses. In other words, among all the possible strategies to rob at least `k` houses while avoiding adjacent homes, we want to find the one that involves the robber stealing the least amount of money from any single house they choose to rob. This will be the smallest maximum amount of money taken from one house in any valid robbing strategy.

Intuition

The intuition behind the solution is to use binary search along with a custom feasibility function to find the desired minimum capability. The first insight is that if the robber can complete the heist with a certain capability, they can also complete it with any higher capability. This makes the problem a good candidate for binary search, where the answer is the smallest capability that allows the robber to rob at least `k` non-adjacent houses.

Here's the step-by-step approach:

1. Define a feasibility function `f(x)` that takes a capability value `x` and returns `True` if it's possible for the robber to complete the heist with capability `x`, or `False` otherwise. This function simulates the robber's actions, stealing from non-adjacent houses with values less than or equal to `x` until they have reached the target of `k` houses.
2. Apply binary search to find the minimum `x` for which `f(x)` is `True`. The search space is from 0 to the maximum amount in any house (inclusive), as the robber could potentially need to rob the house with the most money if `x` equals the total number of houses.
3. The feasibility function keeps two variables: a counter `cnt` for the number of successfully robbed houses, and `j` to track the index of the last house the robber stole from.
4. For each house, if its value is greater than `x` or it's adjacent to the last robbed house (index `j`), the robber skips it.
5. Whenever a house is eligible (it satisfies the conditions and is not next to the last robbed house), increment `cnt` and update `j` to the current house's index.
6. If `cnt` reaches at least `k`, `f(x)` returns `True`, indicating that robbing at least `k` houses is feasible with the current capability `x`.
7. Use Python's `bisect_left` function with a range of capabilities and the feasibility function as the key. This will find the leftmost index where `f(x)` is `True`, which corresponds to the minimum capability required.

By using binary search, we efficiently narrow down the minimum capability required rather than checking every possible capability value.

Solution Approach

The solution is implemented in Python and utilizes a binary search algorithm. Binary search is a highly efficient algorithm used to find an element in a sorted sequence by repeatedly dividing the search interval in half.

Here's a detailed explanation of the solution implementation:

1. **Binary Search Rationale**: Since we are looking for the minimum capability, we set up a binary search over a range of possible capabilities, starting from 0 to `max(nums)`. Binary search works on a sorted sequence, and while capability values aren't inherently sorted, their feasibility is monotonically increasing: if it's possible to rob at least `k` houses with capability `x`, it's possible with any `x' > x` as well.
2. **Feasibility Function `f(x)`**: This function takes a capability value `x` as input and determines whether it is possible to rob at least `k` houses without robbing adjacent ones and not exceeding capability `x`. The function iterates over all the houses, maintaining a count `cnt` of houses robbed and an index `j` of the last house robbed to enforce the non-adjacent constraint.
3. **Conditional House Selection**: While iterating through the houses, two conditions must be checked for each house to decide whether it can be robbed:
   - Its value is less than or equal to the capability `x`.
   - It is not adjacent to the last house robbed, which is checked by comparing the current index `i` with `j + 1`.
4. **Incrementing and Checking Count**: Whenever a house meets the above conditions, we increment `cnt` and update `j` to the current index. If `cnt` becomes greater or equal to `k`, `f(x)` returns `True`, signaling that this capability level is feasible.
5. **Binary Search Execution**: Python's `bisect_left` function is used to find the smallest index in the range `[0, max(nums) + 1)` where `f(x)` returns `True`. Here, `bisect_left(range(max(nums) + 1), True, key=f)` is searching for the position to insert `True` in a sorted sequence of boolean values obtained by applying `f(x)` on each `x` within the range so that the list remains sorted. Since `f(x)` returns `True` for all `x` greater than or equal to the minimum capability, `bisect_left` effectively finds the smallest such `x`.
6. **Time Complexity**: The time complexity of this solution is $O(n \log m)$, where n is the number of houses and m is the maximum amount of money in any house. The function `f(x)` takes $O(n)$ time per check, and the binary search requires $O(\log m)$ checks.
7. The use of binary search greatly optimizes the search process compared to a brute-force approach, which would have been significantly slower and less efficient.

The provided Python solution leverages the algorithmic strength of binary search in combination with a custom-designed feasibility checker to solve this problem of optimizing the robber's capability within the given constraints.

Example Walkthrough

Let's go through a small example to illustrate the solution approach. Suppose we are given the following array of integers representing the amounts of money in the houses and a minimum number of houses `k` the robber needs to steal from:

```
nums = [2, 3, 1, 5, 1, 2]
k = 3
```

Our task is to find the minimum "capability" for the robber so that they can rob at least `k` non-adjacent houses.

Step 1: Binary Search Rationale

We initiate a binary search for capabilities ranging from 0 to the maximum value in `nums`, which is 5 in this case. Hence, our search space is `0` to `5`.

Step 2: Feasibility Function `f(x)`

The feasibility function checks if the robber can rob `k` non-adjacent houses with a capability `x`. Let us see how this function works with an example capability of `x = 3`.

Step 3: Conditional House Selection

When iterating through nums using the feasibility function with `x = 3`, the robber can rob houses with value less than or equal to 3 and must skip adjacent houses.

- House 0: Value is 2 (≤ 3), robber can steal from this house.
- House 1: Value is 3 (≤ 3), but it's adjacent to House 0, so the robber skips this house.
- House 2: Value is 1 (≤ 3), and not adjacent to the last robbed house (House 0), so the robber can steal.
- House 3: Value is 1 (≤ 3), but adjacent to House 2, so it's skipped.
- House 4: Value is 5 (> 3), so this house can't be robbed with capability of 3.
- House 5: Value is 1 (≤ 3), and not adjacent to the last robbed house (House 2), so the robber can steal.

After this process, the robber has stolen from 3 houses (0, 2, and 5), which means `f(3) = True`.

Step 4: Incrementing and Checking Count

We check if at least `k` houses can be robbed without exceeding the capability `x`. Since `f(3)` returned `True`, we can proceed with the binary search to see if there's a lower capability that still allows for `k` houses to be robbed.

Step 5: Binary Search Execution

Using the `bisect_left` function, we find that the smallest capability where `f(x)` returns `True` is actually 2, because the robber could steal from houses 0, 2, and 5 with amounts of 2, 1, and 1, respectively, satisfying the requirement to steal from at least `k = 3` houses without robbing adjacent houses.

Step 6: Time Complexity

The time complexity of this operation for our list `nums` is $O(n \log m)$ where n is 7 and m is 5, resulting from the binary search operation and feasibility checks for each capability value.

By applying this algorithm, we have efficiently determined the minimum "capability" for the robber, allowing them to complete their heist according to the given constraints. In our example, the minimum capability is 2. This is the least amount of money that the robber would need to steal from any single house in their strategy to rob at least `k = 3` non-adjacent houses.

Python Solution

```python
from typing import List
from bisect import bisect_left

class Solution:
    # This function aims to find the minimum capability required to form k pairs
    def minCapability(self, nums: List[int], k: int) -> int:
        # The inner function 'is_feasible' checks if a given capability 'capability'
        # is sufficient to form at least k pairs.
        def is_feasible(capability):
            current_index, pairs_count = 0, 0  # Initialize counters
            for current_index, value in enumerate(nums):
                # Skip if the value is greater than the capability or if the element
                # is right after the previously used element (to ensure non-adjacency).
                if value > capability or current_index == last_used_index + 1:
                    continue
                # If neither condition is met, increment the count
                # and update 'last_used_index'.
                pairs_count += 1
                last_used_index = current_index
            # Check if the capability is sufficient to form k pairs.
            return pairs_count >= k

        # Perform a binary search over the range [0, max(nums) + 1),
        # using the 'is_feasible' function as the key.
        # The 'bisect_left' function finds the leftmost value in the range
        # where 'is_feasible' returns True, which is the minimum capability.
        return bisect_left(range(max(nums) + 1), True, key=is_feasible)
```

Java Solution

```java
class Solution {
    // Determine the minimum capability to partition the array in such a way that
    // the sum of each sub-array is less than or equal to k
    public int minCapability(int[] nums, int k) {
        // Start with the lowest possible capability
        int left = 0;
        // Set an upper limit for the search space, assuming the max value according to problem constraints
        // might be 1 billion
        int right = 1000000000;

        // Perform a binary search to find the minimum capability
        while (left < right) {
            // Calculate the midpoint of the current search space
            int mid = left + (right - left) / 2;

            // Check if the current capability can achieve the required partition
            if (isValidPartition(nums, mid) >= k) {
                // If it qualifies, search the lower half to find a smaller capability
                right = mid;
            } else {
                // Otherwise, search the upper half
                left = mid + 1;
            }
        }

        // Left is now the minimum capability that can achieve the required partition
        return left;
    }

    // Helper method to calculate the number of partitions formed by capability x
    private int calculatePartitions(int[] nums, int x) {
        int count = 0; // Initialize the count of partitions
        int lastPartitionIndex = -2; // Initialize the index of the last partition start

        // Iterate over the array
        for (int i = 0, n = nums.length; i < n; ++i) {
            // Skip if the current number exceeds the capability or is the next immediate number after the last partition
            if (nums[i] > x || i == lastPartitionIndex + 1) {
                continue;
            }
            // Increment the partition count and update lastPartitionIndex
            ++count;
            lastPartitionIndex = i;
        }

        // Return the total number of partitions that can be made with capability x
        return count;
    }
}
```

C++ Solution

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

class Solution {
public:
    // Function to find the minimum capability needed to meet the condition.
    int minCapability(std::vector<int>& nums, int k) {
        // Lambda function to check if a given capability meets the condition.
        auto isCapabilityValid = [&](int capability) {
            int count = 0; // Initialize the count of qualified elements.
            int prevIndex = -1; // Initialize the previous index (-1 is out of possible index range).

            // Iterate through the numbers in the vector.
            for (int i = 0; i < nums.size(); ++i) {
                // Skip the number if it's greater than the capability or
                // if it's the direct neighbor of the previously considered element.
                if (nums[i] > capability || i == prevIndex + 1) {
                    continue;
                }
                // Increment the count of elements and update the previous index.
                ++count;
                prevIndex = i;
            }
            // Return true if the count is greater than or equal to k.
            return count >= k;
        };

        // Binary search setup.
        // Set the left boundary of the search to 0.
        int left = 0;
        // Find the maximum number in the vector and use it as the right boundary.
        int right = *std::max_element(nums.begin(), nums.end());

        // Perform binary search.
        while (left < right) {
            // Calculate the middle value between left and right.
            int mid = (left + right) >> 1;
            // If the current capability meets the condition, adjust the right boundary.
            if (isCapable(mid)) {
                right = mid;
            } else {
                // Otherwise, adjust the left boundary.
                left = mid + 1;
            }
        }
        // After exiting the loop, the smallest capability is found.
        return left;
    }
};
```

Typescript Solution

```typescript
/**
 * Determine the minimum capability required to select at least k elements,
 * such that every two selected elements have at least one other element between them.
 *
 * @param capabilities - Array of capabilities.
 * @param k - Number of elements to select.
 * @return - The minimum capability required.
 */
function minCapability(capabilities: number[], k: number): number {
    /**
     * Helper function to determine if it's possible to select k elements
     * given the constraint of at least one element between selected ones.
     *
     * @param maxCapability - The maximum capability to allow for selection.
     * @return - True if at least k elements can be selected, false otherwise.
     */
    const canSelectWithCapability = (maxCapability: number): boolean => {
        let count = 0; // Count of elements selected
        let lastSelectedIndex = -2; // Index of last selected element

        // Loop through all elements to determine if we can select them
        for (let i = 0; i < capabilities.length; ++i) {
            // Check if the current element can be selected (has desired capability and has gap)
            if (capabilities[i] <= maxCapability && lastSelectedIndex + 1 < i) {
                ++count; // Increment the selected count
                lastSelectedIndex = i; // Update the index of last selected element
            }
        }

        // If we have selected at least k elements, return true.
        return count >= k;
    };

    // Find the left element with the maximum capability.
    let left = 0, right = Math.max(...capabilities);

    // Binary search to find the minimum capability required.
    while (left < right) {
        // Calculate the middle of the current range as the candidate capability.
        const mid = left + ((right - left) >> 1); // Equivalent to Math.floor((left + right) / 2)

        // Use the helper function to check if mid can be a solution.
        if (canSelectWithCapability(mid)) {
            // We can select k elements with the mid capability, try lower capabilities.
            right = mid;
        } else {
            // We cannot select k elements with the mid capability, try higher capabilities.
            left = mid + 1;
        }
    }

    // Return the lowest capability found by the binary search.
    return left;
}
```

Time and Space Complexity

Time Complexity

The given Python function `minCapability` uses binary search through the `bisect_left` function to find the minimum capability required. It applies a helper function `f` as a key which processes the `nums` list on each iteration.

- The `bisect_left` function performs a binary search over a range of size `max(nums) + 1`, which involves $O(\log(\text{Max}))$ iterations, where `Max` is the maximum element in `nums`.
- Within each binary search iteration, the helper function `f` performs a linear scan over the `nums` list of size n, to check how many elements can be skipped without exceeding capability `x`. The time complexity for this will be $O(n)$.

Combining these two, the overall time complexity will be $O(n * \log(\text{Max}))$, where n is the number of elements in `nums`, and `Max` is the maximum element in `nums`.

Space Complexity

The space complexity of the `minCapability` function is $O(1)$ assuming that the list `nums` is given and does not count towards the space complexity (as it's an input). There are no additional data structures used that grow with the input size. The variables `cnt`, `j`, `i`, and `x` use a constant amount of space.