Problem Description

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 ith house. We also have an integer k, which represents the minimum number of houses from which the robber must steal.

In this problem, we are considering a street lined with consecutive houses, each containing a certain amount of money. A robber

Leetcode Link

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 allo

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

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 k equals the total number of

index of the last house the robber stole from.

house (inclusive), as the robber could potentially need to rob the house with the most money if k 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

values less than or equal to 'x' until they have reached the target of k houses.

index where f(x) is True, which corresponds to the minimum capability required.

an element in a sorted sequence by repeatedly dividing the search interval in half.

Here's a detailed explanation of the solution implementation:

Its value is less than or equal to the capability x.

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.

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.

- 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
- 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
- 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

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.

whether it can be robbed:

significantly slower and less efficient.

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

Step 1: Binary Search Rationale

Step 2: Feasibility Function f(x)

with an example capability of x = 3.

possible with any x' > x as well.

It is not adjacent to the last house robbed, which is checked by comparing the current index i with j + 1.
 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

3. Conditional House Selection: While iterating through the houses, two conditions must be checked for each house to decide

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.

The use of binary search greatly optimizes the search process compared to a brute-force approach, which would have been

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:

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

The provided Python solution leverages the algorithmic strength of binary search in combination with a custom-designed feasibility

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.

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

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.

Step 4: Incrementing and Checking Count

Step 5: Binary Search Execution

Step 6: Time Complexity

Python Solution

class Solution:

1 class Solution {

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

9

10

11

12

13

14

15

16

17

1 from typing import List

from bisect import bisect_left

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

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

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.

House 0: Value is 2 (≤ 3), robber can steal from this house.

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 1: Value is 3 (≤ 3), but it's adjacent to House 0, so the robber skips this house.

binary search to see if there's a lower capability that still allows for k houses to be robbed.

House 2: Value is 1 (≤ 3), and not adjacent to the last robbed house (House 0), so the robber can steal.

House 5: Value is 1 (≤ 3), and not adjacent to the last robbed house (House 2), so the robber can steal.

The time complexity of this operation for our list nums is $0(n \log m)$ 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.

This function aims to find the minimum capability required to form k pairs

count, last_used_index = 0, -2 # Initialize counters

If neither condition is met, increment the count

Check if the capability is sufficient to form k pairs.

// Determine the minimum capability to partition the array in such a way that

// Check if the current capability can achieve the required partition

// Function to find the minimum capability needed to meet the condition.

// Lambda function to check if a given capability meets the condition.

int count = 0; // Initialize the count of qualified elements.

// Skip the number if it's greater than the capability or

if (nums[i] > capability || i == prevIndex + 1) {

// Return true if the count is greater than or equal to k.

// Find the maximum number in the vector and use it as the right boundary.

// If the current capability meets the condition, adjust the right boundary.

// if it's the direct neighbor of the previously considered element.

// Increment the count of elements and update the previous index.

int prevIndex = -2; // Initialize the previous index (-2 is out of possible index range).

int minCapability(std::vector<int>& nums, int k) {

// Iterate through the numbers in the vector.

for (int i = 0; i < nums.size(); ++i) {

// Set the left boundary of the search to 0.

int mid = (left + right) >> 1;

int right = *std::max_element(nums.begin(), nums.end());

// Otherwise, adjust the left boundary.

// After exiting the loop, the smallest capability is found.

// Calculate the middle value between left and right.

auto isCapable = [&](int capability) {

continue;

prevIndex = i;

++count;

return count >= k;

// Binary search setup:

// Perform binary search.

if (isCapable(mid)) {

left = mid + 1;

right = mid;

while (left < right) {</pre>

} else {

int left = 0;

Perform a binary search over the range [0, max(nums) + 1),

The inner function 'is_feasible' checks if a given capability 'capability'

Skip if the value is greater than the capability or if the element

if value > capability or current_index == last_used_index + 1:

is right after the previously used element (to ensure non-adjacency).

def minCapability(self, nums: List[int], k: int) -> int:

for current_index, value in enumerate(nums):

and update 'last_used_index'.

last_used_index = current_index

// the sum of each sub-array is less than or equal to k

// Perform a binary search to find the minimum capability

// Get the midpoint of the current search space

if (calculatePartitionCount(nums, mid) >= k) {

// Start with the least possible capability

public int minCapability(int[] nums, int k) {

int mid = (left + right) >> 1;

is sufficient to form at least k pairs.

def is_feasible(capability):

continue

count += 1

return count >= k

Java Solution

// Set an upper limit for the search space, assuming the max value according to problem constraints

using the 'is_feasible' function as the key. # The 'bisect_left' will find the leftmost value in the range # where 'is_feasible' returns True (i.e., the minimum capability). return bisect_left(range(max(nums) + 1), True, key=is_feasible) return bisect_left(range(max(nums) + 1), True, key=is_feasible)

int left = 0;

int right = (int) 1e9;

while (left < right) {</pre>

```
// If it qualifies, search the lower half to find a smaller capability
18
                    right = mid;
19
               } else {
20
                    // Otherwise, search the upper half
21
22
                    left = mid + 1;
23
24
25
26
           // left is now the minimum capability that can achieve the required partition
27
            return left;
28
29
       // Helper method to calculate the number of partitions formed by capability x
30
       private int calculatePartitionCount(int[] nums, int x) {
31
32
            int count = 0; // Initialize the partition count
33
            int lastPartitionIndex = -2; // Initialize the index of the last partition start
34
35
           // Iterate over the array
36
           for (int i = 0; i < nums.length; ++i) {</pre>
                // Skip if the current number exceeds the capability or is the next immediate number after the last partition
37
                if (nums[i] > x || i == lastPartitionIndex + 1) {
39
                    continue;
40
                // Increment the partition count and update lastPartitionIndex
41
42
                ++count;
                lastPartitionIndex = i;
44
45
46
           // Return the total number of partitions that can be made with capability x
            return count;
48
49
50
```

47 return left; 48 } 49 }; 50

C++ Solution

public:

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

1 #include <vector>

class Solution {

};

#include <algorithm>

```
Typescript Solution
 1 /**
    * Determine the minimum capability required to select at least k elements,
    * such that every two selected ones 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.
13
14
      * @param maxCapability - The maximum capability to allow for selection.
      * @return - True if at least k elements can be selected, false otherwise.
15
16
      */
     const canSelectKElements = (maxCapability: number): boolean => {
17
       let count = 0; // Count of elements selected
18
       let lastSelectedIndex = -2; // Index of last selected element
20
       // Loop through all elements to determine if we can select them
21
22
       for (let i = 0; i < capabilities.length; ++i) {</pre>
23
         // Check if the current element can be selected (has desired capability and has gap)
         if (capabilities[i] <= maxCapability && i - lastSelectedIndex > 1) {
24
           ++count; // Increase the selected count
            lastSelectedIndex = i; // Update the index of last selected element
26
27
28
29
       // If we have selected at least k elements, return true.
30
31
       return count >= k;
32
     };
33
34
     let left = 1;
35
     // Find the element with the maximum capability.
     let right = Math.max(...capabilities);
36
37
38
     // Binary search to find the minimum capability required.
     while (left < right) {</pre>
39
       // Take the middle of the current range as the candidate capability.
40
       const mid = (left + right) >> 1; // Equivalent to Math.floor((left + right) / 2)
41
42
43
       // Use the helper function to check if mid can be a solution.
       if (canSelectKElements(mid)) {
44
         // We can select k elements with the mid capability, try lower capabilities.
45
         right = mid;
46
       } else {
47
         // We cannot select k elements with the mid capability, try higher capabilities.
48
         left = mid + 1;
49
50
51
     // Return the lowest capability found by the binary search.
     return left;
55 }
56
```

Time and Space Complexity

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(Max)) iterations, where Max is the maximum element in nums.

maximum element in nums.

Time Complexity

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 0(n).

Combining these two, the overall time complexity will be O(n * log(Max)), where n is the number of elements in nums, and Max is the

The given Python function minCapability uses binary search through the bisect_left function to find the minimum capability

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 v use a constant amount of space.