1984. Minimum Difference Between Highest and Lowest of K Scores

Problem Description

Array

Sorting

Sliding Window

elements and thus potential candidates for the minimum range.

and an integer k. The index of the array starts from 0, which means it is 0-indexed. Your objective is to select the scores of k students such that when you calculate the difference between the highest and the lowest scores amongst these k students, this difference is the smallest possible. The function should then return this minimum possible difference. The challenge can be likened to looking for a subsequence of length k within the array that has the smallest range (i.e., the

In this task, you are provided with an integer array nums, where each element nums [i] represents the score of the i-th student,

difference between the maximum and minimum elements of the subsequence is minimal).

Intuition

Given that we want to minimize the difference between the highest and lowest scores after picking k continuous scores, sorting

Easy

always be next to one another. After sorting, we slide a window of size k across the array and calculate the difference between the first and last elements in this

the array can simplify the problem. Once the array is sorted, the k scores that will have the minimum possible difference will

window (essentially the scores of the k students we've picked). The smallest difference found during this sliding window process will be the minimum difference we are seeking. Sorting makes sure that each such window will contain the k smallest consecutive

The Python code provided sorts the array and then iterates through the array to find the minimum difference between the scores at indices i + k - 1 and i, which represents the end and start of each window of size k. The min function is used on a generator expression to find and return the smallest difference among all these windows.

Solution Approach The solution approach for minimizing the difference between the highest and the lowest of the k scores selected from the array

Sort the Array: First, we sort the array in increasing order. This is important because we want to ensure that when we pick k

contiguous scores from the array, we are getting the smallest possible range (difference between the highest and the lowest

involves the following steps:

score) for that segment. Sliding Window: Once the array is sorted, instead of looking at every possible combination of k scores (which would be inefficient), we use a sliding window of size k. The sliding window starts at the beginning of the array and moves one position

to the right at each step, stopping when the end of the window reaches the end of the array.

- Calculate Differences: For each window, calculate the difference between the last and the first elements (nums [i + k 1] nums [i]). This difference represents the range of scores within that window of k students. Because the array is sorted, this range is also the difference between the highest and the lowest scores in the selection of k scores.
- Find the Minimum Difference: After calculating the differences for each window, we select the minimum difference obtained, which represents the solution to the problem. In Python, this is efficiently done using the min() function applied to a generator expression that iterates through the valid starting indices for the windows (\emptyset to len(nums) - k).

The data structure we principally use is the array (or list in Python), and the algorithm employs the sorting and sliding window

techniques. The sorting is typically an O(n log n) operation, and the sliding window traversal of the list is an O(n) operation,

making the overall time complexity of this algorithm be O(n log n). Here's part of the code that exemplifies the approach:

nums.sort() # Step 1: Sort the array # Step 2 and 3: [Sliding window](/problems/sliding_window_maximum) calculation of the minimum difference $min_diff = min(nums[i + k - 1] - nums[i] for i in range(len(nums) - k + 1))$ In the min_diff calculation, i represents the start index of the sliding window and i + k - 1 is the end index of the window. The

generator expression is used to dynamically create the range values for each window without consuming extra space to store

Example Walkthrough Let's illustrate the solution approach with a small example. Suppose the input integer array nums is [1, 5, 6, 3, 2, 8] and the integer k is 3. We want to find a subset of k students with the minimum possible difference between the highest and the lowest

Sliding Window: We then create a sliding window of size k to go through the sorted array. With k = 3, our sliding window will

Sort the Array: First, we sort nums in increasing order. After sorting, nums becomes [1, 2, 3, 5, 6, 8].

consider the following subsets as it moves from the beginning of the array to the end:

Window 2: [2, 3, 5] Window 3: [3, 5, 6]

Window 1: [1, 2, 3]

Window 4: [5, 6, 8]

these ranges.

scores.

Calculate Differences: We calculate the difference between the last and the first elements of each window, which represents the score range for that subset. ○ Difference for Window 1: 3 - 1 = 2

from the first window [1, 2, 3]. This means the smallest possible difference between the highest and the lowest of the k

○ Difference for Window 4: 8 - 5 = 3 Find the Minimum Difference: Among all the differences calculated, we find that the minimum difference is 2, which comes

The sliding window approach allows us to efficiently find the minimum range without having to explore every possible

combination of k elements. By sorting the array first, we guarantee that the elements within any given window are as close

together as possible, which is key to finding the subset that minimizes the range. Therefore, the minimum possible difference in this example is 2.

min_diff = float('inf')

for i in range(len(nums) - k + 1):

Initialize the minimum difference to a large number.

This will be updated as we find smaller differences.

we have at least 'k' elements remaining in the list.

range of the 'k' elements we are considering.

import java.util.Arrays; // Import Arrays class to use the sort method

int minimumDifference(vector<int>& nums, int k) {

for (int i = 0; i + k <= nums.size(); ++i) {</pre>

std::sort(nums.begin(), nums.end());

// values in this subsequence

// Return the smallest difference found

* @param {number[]} nums - An array of integers.

// Sort the array in non-decreasing order.

let minDiff = nums[numLength - 1] - nums[0];

for (let i = 0; i + k - 1 < numLength; i++) {

let currentDiff = nums[i + k - 1] - nums[i];

minDiff = Math.min(currentDiff, minDiff);

// Return the smallest difference found.

function minimumDifference(nums: number[], k: number): number {

// Iterate over the array, considering all k-sized windows.

// Update minDiff with the smaller of the two differences.

int min_diff = INT_MAX;

// (i + k <= nums.size())

return min_diff;

TypeScript

/**

*/

// First ensure the array is sorted in non-decreasing order.

// Loop through the array, considering each subsequence of length 'k'

// Ensure that we can form a subsequence of length 'k' by checking the condition

// Update the minimum difference if the current difference is smaller

// Calculate the difference between the max (nums[i + k - 1]) and min (nums[i])

* This function finds the minimum difference between the largest and the smallest value in any k-sized subset of 'nums'.

// Initialize an answer variable with a large value.

int current_diff = nums[i + k - 1] - nums[i];

min_diff = std::min(min_diff, current_diff);

minimum difference, update the minimum difference.

current diff = nums[i + k - 1] - nums[i]

Iterate over the list, only going up to the point where

Calculate the difference between the number at the current index

If the current difference is smaller than the previously stored

and the number k-1 indices ahead. This difference is the

Solution Implementation

○ Difference for Window 2: 5 - 2 = 3

○ Difference for Window 3: 6 - 3 = 3

scores selected from the array is 2.

Python class Solution: def minimum_difference(self, nums: List[int], k: int) -> int: # First, we sort the list of numbers in ascending order. nums.sort()

min_diff = min(min_diff, current_diff) # After checking all possible groups of 'k' elements, # return the smallest difference found.

Java

return min_diff

```
class Solution {
    // Method to find the minimum difference between the max and min values in any subarray of k elements
    public int minimumDifference(int[] nums, int k) {
       // Sort the array in non-decreasing order
       Arrays.sort(nums);
       // Initialize the answer with a large value that is certain to be larger than any possible difference in the array
       int minDifference = Integer.MAX_VALUE;
       // Iterate through the array and consider each subarray of size k
        for (int i = 0; i < nums.length - k + 1; ++i) {</pre>
           // Calculate the difference between the last and the first element of the current subarray
            int currentDifference = nums[i + k - 1] - nums[i];
            // Update the minimum difference if the current difference is smaller
           minDifference = Math.min(minDifference, currentDifference);
       // Return the minimum difference found
        return minDifference;
C++
#include <vector>
#include <algorithm> // Include algorithm for std::sort and std::min functions
class Solution {
public:
   // Function to calculate the minimum difference between the maximum and minimum
   // elements of any subsequence of length 'k'.
```

```
nums.sort((a, b) \Rightarrow a - b);
// Get the length of the number array.
const numLength = nums.length;
```

* @param {number} k - An integer representing the size of the subset to consider.

* @returns {number} - The minimum difference found among all k-sized subsets of 'nums'.

// Initialize the answer with the largest possible difference in the sorted array.

// Calculate the difference between the end and start of the current window.

```
return minDiff;
class Solution:
   def minimum_difference(self, nums: List[int], k: int) -> int:
       # First, we sort the list of numbers in ascending order.
       nums.sort()
       # Initialize the minimum difference to a large number.
       # This will be updated as we find smaller differences.
       min_diff = float('inf')
       # Iterate over the list, only going up to the point where
       # we have at least 'k' elements remaining in the list.
        for i in range(len(nums) - k + 1):
```

Calculate the difference between the number at the current index

If the current difference is smaller than the previously stored

and the number k-1 indices ahead. This difference is the

range of the 'k' elements we are considering.

After checking all possible groups of 'k' elements,

minimum difference, update the minimum difference.

current_diff = nums[i + k - 1] - nums[i]

min diff = min(min diff, current diff)

return the smallest difference found.

```
a sorted array. The code performs a sort operation on the array and then iterates through the array to find the smallest difference
between k consecutive elements. The analysis of the time complexity and space complexity is as follows:
```

Time and Space Complexity

return min_diff

Time Complexity

minimum difference.

nums.sort(): Sorting an array of n elements generally has a time complexity of O(n log n). Python uses Timsort (a hybrid sorting algorithm derived from merge sort and insertion sort) for its sort function, which also has a worst-case time complexity of O(n log n).

min(nums[i + k - 1] - nums[i] for i in range(len(nums) - k + 1)): This part of the code involves a single for loop that

The time complexity of the code is dictated primarily by the sorting algorithm and the subsequent loop that calculates the

The given Python code aims to find the minimum difference between the maximum and minimum values within any k elements of

- iterates n k + 1 times, where n is the length of the input array nums. Since k is a constant (with respect to the size of the input array), the iteration is O(n). Combining these two parts, the overall time complexity is 0(n log n + n). In terms of big-O notation, the 0(n log n) term
- dominates, so the final time complexity is $0(n \log n)$. **Space Complexity**

The space complexity of the code is considered by analyzing the extra space required in addition to the input.

also 0(1).

nums.sort(): The sort operation occurs in-place and does not require additional space proportional to the input size (extra

constants space might be used, depending on the implementation details of the sort, but this does not scale with input size). Therefore, it does not change the space complexity and is 0(1). min(...): The min operation itself does not require additional space that scales with the input size since the result of nums[i +

k - 1] - nums [i] is calculated on the fly for each iteration and does not require storing all values. Hence, the space used is

Considering both operations, the overall space complexity of the code is 0(1), as no additional space that scales with the size of the input is used.