2616. Minimize the Maximum Difference of Pairs

## **Problem Description**

Medium

In this problem, you are given an array of integers nums and an integer p. Your task is to find p pairs of indices (let's denote each pair as (i, j)) in the array such that each index is used only once across all pairs and the maximum difference of the values at these indices is as small as possible. The difference for a pair (i, j) is defined as the absolute value of nums [i] - nums[j]. You need to determine the smallest maximum difference possible amongst all these p pairs. To summarize:

• The difference is calculated as the absolute difference of numbers at the given indices.

Indices should not be used more than once.

**Greedy Array Binary Search** 

The goal is to find the minimum of the maximum differences.

The maximum difference among the pairs should be minimized.

- The key to solving this problem lies in realizing that if a certain max difference x can satisfy the conditions (allowing for the

p pairs with differences less than or equal to x.

## formation of p pairs), any max difference larger than x would also satisfy the conditions. This gives us a hint that binary search can be used, as we're dealing with a monotonically non-decreasing condition. We aim to find the smallest x that allows us to form

To simplify the search, we first sort the array nums. Sorting allows us to consider pairs in a sequence where we can readily calculate differences between adjacent elements and check if the difference fits our current guess x. We then apply binary search over the potential differences, starting from 0 up to the maximum possible difference in the sorted array, which is nums[-1] - nums[0].

greedily form pairs with a difference less than or equal to x. If an adjacent pair fits the condition, we count it as one of the p pairs

and skip the next element since it's already paired. If the difference is too large, we move to the next element and try again. Our

check passes if we can form at least p pairs this way. By combining binary search to find the minimum possible x with the greedy pairing strategy, we efficiently arrive at the minimum maximum difference among all the p pairs.

We use a greedy approach to check if a certain maximum difference x is possible: we iterate through the sorted array and

that allows for the existence of p index pairs with that maximum difference. Here are the steps of the implementation:

**Sorting**: First, we sort the array nums in non-decreasing order. This is done to simplify the process of finding pairs of elements

**<u>Binary Search</u>**: We perform binary search on the range of possible differences, starting from 0 to nums [-1] - nums [0]. We use

The solution follows a binary search approach combined with a greedy strategy to determine the minimum maximum difference

## binary search because the possibility of finding p pairs is a monotonically non-decreasing function of the difference, x. If we can find p pairs for a given x, we can also find p pairs for any larger value of x. We aim to find the smallest such x that works.

**Solution Approach** 

**Greedy Pairing:** For a guess x in the binary search, we apply a greedy method to check if we can form p pairs with differences less than or equal to x. We iterate through the sorted nums and for the current index i, if nums[i + 1] - nums[i] is less than or equal to x, we count this as a valid pair (nums[i], nums[i + 1]) and increment our pairs count (cnt). We then skip the next

element by incrementing i by 2, because we can't reuse indices. If nums[i + 1] - nums[i] is greater than x, it doesn't form a valid pair, and we just increment i by 1 to try the next element with its adjacent one.

least p pairs, solving the problem.

difference of the values at these index pairs.

We start with the first index i = 0:

we could form the pairs with is 10.

solving such problems.

**Python** 

Solution Implementation

from bisect import bisect\_left

from typing import List

class Solution:

Now at i = 2:

returning the value of x that minimizes the maximum difference.

Let's consider a small example to illustrate the solution approach:

with the smallest possible difference.

Check Function: The check function is the core of this greedy application. It takes a difference diff as its argument and returns a boolean, indicating whether it is possible to find at least p pairs with a maximum difference of diff or less.

Binary Search with Custom Checker: Using bisect\_left from the bisect module, we find the smallest x that makes the

check function return True. The bisect\_left function is given a range and a custom key function, which is the aforementioned

check function. It effectively applies binary search to find the leftmost point in the range where the check condition is met,

**Result**: The result of the binary search gives us the minimum value of x, the maximum difference, that allows us to form at

Essentially, the binary search narrows down the search space for the maximum difference while the greedy approach checks its

- viability, ensuring an efficient and correct solution to the problem. **Example Walkthrough**
- Step 1: Sorting We sort the array, although nums is already sorted in this case: [1, 3, 6, 19, 20]. Step 2: Binary Search We will perform a binary search for the smallest maximum difference on the range from 0 to nums [-1] nums [0], which is 20 - 1 = 19.

Step 3: Greedy Pairing Using the Check Function (Example) Let's pick a mid-value from our binary search range, say x = 9.

• nums [i + 1] - nums [i] equals 3 - 1 = 2, which is less than x; we can form a pair (1, 3). We move to i = 2.

the value of x until we can find the minimum x that allows forming p = 2 pairs.

We formed p = 2 pairs with a max difference x = 10, which are (1, 3) and (19, 20).

• For i = 0: The difference is 2 < x; pair (1, 3). Increment i by 2 to i = 2.

for forming p = 2 pairs given the array nums = [1, 3, 6, 19, 20].

def minimizeMax(self, nums: List[int], p: int) -> int:

if nums[i + 1] - nums[i] <= diff:</pre>

while i < len(nums) - 1:</pre>

return pairs\_count >= p

# difference in the sorted array.

// Number of elements in the array

// Initialize binary search bounds

int arrayLength = nums.length;

public int minimizeMax(int[] nums, int pairsToForm) {

// Sort the array to prepare for binary search

int minimizeMax(std::vector<int>& nums, int pairsToRemove) {

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

auto check = [&](int diff) -> bool {

for (int i = 0; i < numCount - 1; ++i) {

int numCount = nums.size();

// Store the number of elements in 'nums'

// First, sort the array to easily identify and remove pairs with minimal differences

// Lambda function to check if 'diff' is sufficient to remove 'pairsToRemove' pairs

// If the difference between a pair is less than or equal to 'diff'

// Initialize the binary search bounds for the minimal max difference

int pairsCount = 0; // Keep track of the number of pairs removed

// Iterate over the sorted numbers and attempt to remove pairs

int leftBound = 0, rightBound = nums[numCount - 1] - nums[0] + 1;

else:

# Loop through the numbers and determine if we can form enough pairs

pairs\_count += 1 # Acceptable pair, increment the count

i += 2 # Skip the next element as it has been paired up

# Use binary search to find the minimum possible 'diff' such that at least 'p' pairs

min\_possible\_diff = bisect\_left(range(nums[-1] - nums[0] + 1), True, key=is\_possible)

# have a difference of 'diff' or less. The search range is from 0 to the maximum

i += 1 # Move to the next element to find a pair

# Return True if we have enough pairs, otherwise False

# Sort the input array before running the binary search algorithm

# If the difference between the current pair is less than or equal to 'diff'

• nums[i + 1] - nums[i] equals 19 - 6 = 13, which is greater than x; the pair (6, 19) does not work. We move to i = 3.

Suppose we have an array nums = [1, 3, 6, 19, 20] and we need to find p = 2 pairs such that we minimize the maximum

At i = 3, we cannot pair 19 because there are no more elements to compare with that can form a valid pair with a difference less than or equal to x. We have only been able to form 1 pair.

Since we cannot form p = 2 pairs with x = 9, the maximum difference x must be higher. The binary search will continue adjusting

• For i = 2: The difference is 13 > x; we cannot pair (6, 19). Increment i by 1 to i = 3. • For i = 3: The difference is 20 - 19 = 1 < x; pair (19, 20). Increment i by 2 to i = 5.

Since we can form the required p pairs with x = 10 and we could not do so with x = 9, the smallest possible maximum difference

Step 5: Result Through the greedy and binary search methods described, we have found the smallest maximum difference is 10

Step 4: Binary Search Continues Let's try with a larger x, say x = 10. Now we perform the greedy pairing check again:

The binary search determines that the correct value x for the minimum maximum difference cannot be less than 10 if we are to form the required number of pairs, while the greedy strategy confirms the feasibility of x by actually attempting to form the pairs.

This example demonstrates both the complexity and the effectiveness of using binary search with a greedy pairing strategy in

# Helper function to check if it is possible to have 'diff' as the maximum difference # of at least 'p' pairs after removing some pairs from 'nums'. def is possible(diff: int) -> bool: pairs\_count = 0 # Count of valid pairs i = 0 # Index to iterate through 'nums'

return min\_possible\_diff Java

Arrays.sort(nums);

int left = 0;

class Solution {

nums.sort()

```
int right = nums[arrayLength - 1] - nums[0] + 1;
        // Perform binary search to find the minimum maximum difference
       while (left < right) {</pre>
            int mid = (left + right) >>> 1; // Mid-point using unsigned bit-shift to avoid overflow
           // If enough pairs can be formed with this difference, go to left half
            if (countPairsWithDifference(nums, mid) >= pairsToForm) {
                right = mid;
            } else { // Otherwise, go to right half
                left = mid + 1;
        // Minimum maximum difference when the correct number of pairs are formed
        return left;
    private int countPairsWithDifference(int[] nums, int maxDifference) {
        int pairCount = 0; // Count pairs with a difference less than or equal to maxDifference
        for (int i = 0; i < nums.length - 1; ++i) {
           // If a valid pair is found, increase count
            if (nums[i + 1] - nums[i] <= maxDifference) {</pre>
                pairCount++;
                i++; // Skip the next element as it's already paired
        return pairCount;
C++
#include <vector>
#include <algorithm> // to use sort function
class Solution {
public:
    // Function to minimize the maximum difference between pairs after 'p' pairs have been removed
```

```
if (nums[i + 1] - nums[i] <= diff) {</pre>
                    pairsCount++; // Increment the count of removed pairs
                    i++; // Skip the next element as it's part of a removed pair
            // True if enough pairs can be removed, false otherwise
            return pairsCount >= pairsToRemove;
        };
        // Perform binary search to find the minimal max difference
       while (leftBound < rightBound) {</pre>
            int mid = (leftBound + rightBound) >> 1; // Equivalent to average of the bounds
            if (check(mid)) {
                // If 'mid' allows removing enough pairs, look for a potentially lower difference
                rightBound = mid;
            } else {
                // Otherwise, increase 'mid' to allow for more pairs to be removed
                leftBound = mid + 1;
        // The left bound will be the minimal max difference after the binary search
        return leftBound;
};
TypeScript
// Define the function to minimize the maximum difference between pairs after 'pairsToRemove' pairs have been removed
function minimizeMax(nums: number[], pairsToRemove: number): number {
   // Sort the array to easily identify and remove pairs with minimal differences
    nums.sort((a, b) => a - b);
    // Store the number of elements in 'nums'
    let numCount: number = nums.length;
    // Initialize the binary search bounds for the minimal max difference
    let leftBound: number = 0, rightBound: number = nums[numCount - 1] - nums[0] + 1;
    // Define a checker function to check if 'diff' is sufficient to remove 'pairsToRemove' pairs
    const check = (diff: number): boolean => {
        let pairsCount: number = 0; // Keep track of the number of pairs removed
       // Iterate over the sorted numbers and attempt to remove pairs
        for (let i = 0; i < numCount - 1; ++i) {</pre>
            // If the difference between a pair is less than or equal to 'diff'
            if (nums[i + 1] - nums[i] <= diff) {</pre>
                pairsCount++; // Increment the count of removed pairs
                i++; // Skip the next element as it's part of a removed pair
       // Return true if enough pairs can be removed, false otherwise
        return pairsCount >= pairsToRemove;
    };
    // Perform binary search to find the minimal max difference
    while (leftBound < rightBound) {</pre>
        const mid: number = leftBound + Math.floor((rightBound - leftBound) / 2); // Compute the average of the bounds
        if (check(mid)) {
            // If 'mid' allows removing enough pairs, look for a potentially lower difference
            rightBound = mid;
        } else {
            // Otherwise, increase 'mid' for the possibility to remove more pairs
            leftBound = mid + 1;
```

// Return the left bound as the minimal max difference after the binary search completes

# Helper function to check if it is possible to have 'diff' as the maximum difference

# If the difference between the current pair is less than or equal to 'diff'

# Loop through the numbers and determine if we can form enough pairs

i += 1 # Move to the next element to find a pair

pairs\_count += 1 # Acceptable pair, increment the count

i += 2 # Skip the next element as it has been paired up

# Use binary search to find the minimum possible 'diff' such that at least 'p' pairs

min\_possible\_diff = bisect\_left(range(nums[-1] - nums[0] + 1), True, key=is\_possible)

# have a difference of 'diff' or less. The search range is from 0 to the maximum

## return min\_possible\_diff Time and Space Complexity

def minimizeMax(self, nums: List[int], p: int) -> int:

pairs\_count = 0 # Count of valid pairs

i = 0 # Index to iterate through 'nums'

if nums[i + 1] - nums[i] <= diff:</pre>

def is\_possible(diff: int) -> bool:

while i < len(nums) - 1:

return pairs\_count >= p

# difference in the sorted array.

else:

nums.sort()

# of at least 'p' pairs after removing some pairs from 'nums'.

# Return True if we have enough pairs, otherwise False

# Sort the input array before running the binary search algorithm

return leftBound;

from bisect import bisect\_left

from typing import List

class Solution:

performing certain operations. It uses a binary search mechanism on the range of possible differences and a greedy approach to check for the validity of a specific difference. The time complexity of the code is  $0(n * (\log n + \log m))$ , where n is the length of the array nums, and m is the difference

The given code snippet is designed to minimize the maximum difference between consecutive elements in the array after

between the maximum and minimum values in the array nums. The sorting operation contributes 0(n log n), whereas the binary search contributes 0(log m). During each step of the binary search, the check function is called, which takes 0(n). The space complexity of the code is 0(1), which indicates that the space required does not depend on the size of the input array nums. Aside from the input and the sorted array in place, the algorithm uses a fixed amount of additional space.