<https://leetcode.com/problems/minimum-absolute-difference-between-elements-with-constraint/description/>

You are given a 0-indexed integer array nums and an integer x.

Find the minimum absolute difference between two elements in the array that are at least x indices apart.

In other words, find two indices i and j such that abs(i - j) >= x and abs(nums[i] - nums[j]) is minimized.

Return an integer denoting the minimum absolute difference between two elements that are at least x indices apart.

**Example 1:**

**Input:** nums = [4,3,2,4], x = 2

**Output:** 0

**Explanation:** We can select nums[0] = 4 and nums[3] = 4. They are at least 2 indices apart, and their absolute difference is the minimum, 0. It can be shown that 0 is the optimal answer.

**Example 2:**

**Input:** nums = [5,3,2,10,15], x = 1

**Output:** 1

**Explanation:** We can select nums[1] = 3 and nums[2] = 2.They are at least 1 index apart, and their absolute difference is the minimum, 1.It can be shown that 1 is the optimal answer.

**Example 3:**

**Input:** nums = [1,2,3,4], x = 3

**Output:** 3

**Explanation:** We can select nums[0] = 1 and nums[3] = 4.They are at least 3 indices apart, and their absolute difference is the minimum, 3.It can be shown that 3 is the optimal answer.

**Constraints:**

1 <= nums.length <= 10^5

1 <= nums[i] <= 10^9

0 <= x < nums.length

**Attempt 1: 2023-03-02**

**Solution 1: Brute Force (10 min, TLE 2010/2013)**

class Solution {

    public int minAbsoluteDifference(List<Integer> nums, int x) {

        int result = Integer.MAX\_VALUE;

        int n = nums.size();

        for(int i = 0; i < n; i++) {

            for(int j = i + x; j < n; j++) {

                result = Math.min(result, Math.abs(nums.get(i) - nums.get(j)));

            }

        }

        return result;

    }

}

**Solution 2: Sliding Window + Hash Table (30 min)**

class Solution {

    public int minAbsoluteDifference(List<Integer> nums, int x) {

        int result = Integer.MAX\_VALUE;

        int n = nums.size();

        // TreeMap to store the occurrences of the numbers in the current window

        TreeMap<Integer, Integer> map = new TreeMap<>();

        // Start from index 'x', as we want to have a complete window nums[0, x - 1]

        for(int i = x; i < n; i++) {

            // Add or update the frequency of the starting element of the current window

            map.put(nums.get(i - x), map.getOrDefault(nums.get(i - x), 0) + 1);

            // Find the smallest number greater than or equal to the current number

            Integer higherOrEqual = map.ceilingKey(nums.get(i));

            if(higherOrEqual != null) {

                result = Math.min(result, higherOrEqual - nums.get(i));

            }

            // Find the greatest number less than or equal to the current number

            Integer lowerOrEqual = map.floorKey(nums.get(i));

            if(lowerOrEqual != null) {

                result = Math.min(result, nums.get(i) - lowerOrEqual);

            }

        }

        return result;

    }

}

Time Complexity: O(N\*logx)

Space Complexity: O(N)

**Refer to**

<https://algo.monster/liteproblems/2817>

**Problem Description**

In this problem, we're provided with an array of integers, nums, and a separate integer x. The objective is to find the smallest possible absolute difference between two elements of the array, given that these elements' indices are x or more positions apart. More formally, we need to find indices i and j (where i can be before or after j) in such a way that abs(i - j) >= x, while simultaneously minimizing the value of abs(nums[i] - nums[j]).

The problem defines an absolute difference function abs, which calculates the non-negative difference between two numbers. In this context, it's applied twice: first to ensure that the indices i and j are sufficiently separated by at least x indices; second, to measure the difference in value between the array entries at those indices.

**Intuition**

The main intuition behind the solution is that to find the minimum absolute difference between elements that are at least x indices apart, we don't need to compare each element with every other element that meets the index distance requirement. That would be inefficient, especially for large arrays. Instead, we can maintain a subset of the array elements that allows us to quickly find the closest elements in value to any given element in the array. We use a data structure that maintains the elements in sorted order and allows for efficient lookups - an ordered set.

The ordered set is implemented using SortedList, which comes from the sortedcontainers Python library. This data structure supports logarithmic time complexity for insertion, deletion, and lookup operations—a key benefit when dealing with a potentially large number of elements.

To implement the solution, we iterate over the array starting from index x, for each element, adding the element that is exactly x indices before it to the ordered set. Now, for every nums[i] (where i >= x), we only need to compare it with the closest values that are already in the ordered set. These closest values are the ones that could potentially yield the minimum absolute difference.

The bisect\_left function from the Python's bisect module is used to find the position where nums[i] would fit in the sorted list. This gives us two candidates: the element right before and right after the position found by bisect\_left. Compare nums[i] with these two candidates, and the smaller difference is a potential answer. Update the answer if this difference is smaller than all previously seen differences.

This way, we efficiently narrow our search for the minimum absolute difference to just two comparisons per array element considered, using the ordered set to quickly find the best candidates for comparison.

**Solution Approach**

The solution is implemented in Python and involves the use of the SortedList data structure to maintain a dynamically ordered subset of the array elements for quick comparisons. This ordered set allows us to find the minimum difference efficiently.

Here's a step-by-step breakdown of how the algorithm works:

We initialize our ordered set sl by importing SortedList. The set is initially empty because we populate it as we iterate through the array.

We start with a variable ans initialized to inf, representing the smallest absolute difference found so far. inf is a placeholder for infinity, ensuring that any real absolute difference found will be smaller and thus will replace inf.

We enumerate the elements of nums starting from index x (i.e., the first element that can have another element in the array which is x indices apart is at index x).

As we visit each element nums[i] where i >= x, we add the element nums[i - x] to our ordered set sl. This is because nums[i - x] is exactly x indices before nums[i], satisfying the distance constraint.

We then use the bisect\_left function from the bisect module to find the position in the ordered set where nums[i] would fit based on its value.

Two adjacent positions in the ordered set are considered:

The position found by bisect\_left (sl[j]), which is the first element in the ordered set that is not less than nums[i].

The position immediately before (sl[j - 1]), which is the largest element in the ordered set that is less than nums[i] (if it exists).

If j is less than the length of the ordered set, it means there's an element on its right; we calculate the difference between this element and nums[i], and if it's smaller than ans, we update ans.

If j is not 0 (which means that there is an element in the set smaller than nums[i]), we find the difference with the left neighbor sl[j - 1] and update ans if this new difference is smaller.

After iterating through all elements from index x to the end of the array, the variable ans holds the minimum absolute difference, and we return this value.

Through the use of SortedList and the binary search function bisect\_left, this approach optimizes what could have been an O(n^2) brute force comparison process into an O(n log n) process, since each of the n - x iterations takes O(log n) due to operations on the ordered set.

**Example Walkthrough**

Let's walk through a small example to illustrate the solution approach.

Consider the array nums = [4, 2, 5, 1, 3] and x = 2. We want to find the smallest absolute difference between two elements where the indices are at least x positions apart.

We begin by initializing SortedList as sl and setting ans to inf.

Now, we start iterating over the array from the index x, which is 2 in this case. So we look at nums[2] which is 5.

Before we consider nums[2], we add nums[0] (4 in this case) to sl. Now sl = [4].

We then check for the closest elements in sl to 5. Since sl has only one value, we only need to check 4. The absolute difference abs(5 - 4) = 1, so we update ans from inf to 1.

We move to the next index, i = 3. We add nums[1] (2) to sl, so now sl = [2, 4].

nums[3] is 1. Using bisect\_left, we find that 1 would be inserted at the beginning of sl, so we compare it to the first element (2). The absolute difference abs(1 - 2) = 1. Our ans is already 1, so it remains unchanged.

Moving to i = 4, we add nums[2] (5) to sl. Now, sl becomes [2, 4, 5].

nums[4] is 3. Using bisect\_left, we determine 3 would fit between 2 and 4 in sl. Hence, we compare 3 with both sides: abs(3 - 2) = 1 and abs(3 - 4) = 1. ans remains 1.

Having checked all elements from index x to the end of nums, we conclude that the smallest absolute difference we found is 1, so we return ans which is 1.

Thus, the minimum absolute difference between elements at least x indices apart is 1. This example demonstrates the efficacy of the algorithm in finding the solution with fewer comparisons, thanks to the ordered set sl and binary search techniques.

**Java Solution**

class Solution {

    public int minAbsoluteDifference(List<Integer> nums, int windowSize) {

        // TreeMap to store the occurrences of the numbers in the current window

        TreeMap<Integer, Integer> frequencyMap = new TreeMap<>();

        // Initialize the answer with the maximum possible positive integer value

        int minimumDifference = Integer.MAX\_VALUE;

        // Start from index windowSize, as we want to have a complete window

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

            // Add or update the frequency of the starting element of the current window

            frequencyMap.merge(nums.get(i - windowSize), 1, Integer::sum);

            // Find the smallest number greater than or equal to the current number

            Integer higherOrEqual = frequencyMap.ceilingKey(nums.get(i));

            if (higherOrEqual != null) {

                // If found, update the minimum difference

                minimumDifference = Math.min(minimumDifference, higherOrEqual - nums.get(i));

            }

            // Find the greatest number less than or equal to the current number

            Integer lowerOrEqual = frequencyMap.floorKey(nums.get(i));

            if (lowerOrEqual != null) {

                // If found, update the minimum difference

                minimumDifference = Math.min(minimumDifference, nums.get(i) - lowerOrEqual);

            }

        }

        // Return the found minimum absolute difference

        return minimumDifference;

    }

}

**Time and Space Complexity**

The time complexity of the provided code is O(n \* log(x)) where n is the length of the array nums and x is the window size given as a parameter. This is due to the fact that we are iterating through n - x elements, and for each element, we perform operations like add and bisect\_left which are O(log(x)). Since x is less than or equal to n, in the worst case, the time complexity could be mistaken as O(n \* log(n)), but it is crucial to note that x dictates the actual size of the sorted list, not n.

The space complexity of the code is O(x) as the space is used only for maintaining the sorted list with a maximum of x elements at any time, rather than O(n) as inferred in the reference. If x were to be considered constant or a value that does not scale with n, the space complexity could be considered as O(1).

**Refer to**

<https://leetcode.com/problems/minimum-absolute-difference-between-elements-with-constraint/solutions/3901733/c-from-brute-force-to-optimization-sliding-window-binary-search/>

**Explanation**

Brute force is very intuitive

int minAbsoluteDifference(vector<int>& nums, int x) {

int n = nums.size(), res = 1e9;

for(int i = 0; i < n; i++){

for(int j = i + x; j < n; j++){

res = min(res, abs(nums[i] - nums[j]));

}

}

return res;

}

To optimize, we need to remove the inner for loop traversal. Since 1 <= nums.length <= 10^5, we can use Binary Search to find the closest element of each element from i = 0 to i = n - x - 1 in the search space i + x to n - 1.

Use a sliding window approach to consider the subarray in which we find the closest element.

Eg. nums = {1,2,3,4,5,6,7} and x = 3

Sliding window:

[1], 2, 3, 4,5,6,7 -> 1, [2], 3, 4,5,6,7 -> 1, 2, [3], 4, 5,6,7 ....

Here, for each element nums[i], we apply B.S on the window shown. Use multiset to store elements in sorted order for B.S

**Code**

class Solution {

public:

int minAbsoluteDifference(vector<int>& nums, int x) {

int n = nums.size(), res = 1e9;

multiset<int> st;

for (int i = x ; i < n; i++) st.insert(nums[i]);

for (int i = 0; i < n - x; i++) {

auto it = st.lower\_bound(nums[i]);

if (it != st.end()) {

res = min(res, abs(nums[i] - \*it));

}

if (it != st.begin()) {

--it;

res = min(res, abs(nums[i] - \*it));

}

st.erase(st.find(nums[i + x]));

}

return res;

}

};

**Complexity**

Time complexity: O(nlogn)

Space complexity: O(n)