# 1499. Max Value of Equation

Hard | Queue | Array | Sliding Window | Monotonic Queue | Heap (Priority Queue)

## Problem Description

You are given an array of `points`, each containing a pair of coordinates `[x, y]` that represent points on a 2D plane. These points are sorted according to their x-coordinate values, ensuring that `x[i] < x[j]` for any two points `points[i]` and `points[j]` where `i < j`. Along with this array, you're also given an integer `k`.

Your task is to find and return the maximum value obtained from the equation `y[i] + y[j] + |x[i] - x[j]|` for any two different points, provided that the distance between these points in x-direction (`|x[i] - x[j]|`) is less than or equal to `k`.

It's guaranteed that you'll be able to find at least one pair of points fulfilling the condition `|x[i] - x[j]| <= k`.

In essence, you need to find a pair of points not further away from each other than `k` units along the x-axis, such that the given equation yields the maximum result.

## Intuition

The core concept of the solution lies in understanding that we want to maximize the sum `y[i] + y[j] + |x[i] - x[j]|`, and that the absolute value `|x[i] - x[j]|` can simply be considered as `x[j] - x[i]` since the points are sorted by x-coordinate. Therefore, the equation simplifies to `y[i] + y[j] + x[j] - x[i]`. This can be rearranged to `(y[i] - x[i]) + (y[j] + x[j])`.

A crucial observation here is that for any point `j`, we want to find a point `i` with the maximum possible value of `y[i] - x[i]`. Such a point `i` should also be within the distance `k` from point `j` along the x-axis. To efficiently find this point `i` for each point `j`, we can use a queue to keep potential candidates of point `i` that are within the distance `k` from the current point `j`.

The goal is to maintain a monotonic queue where the value of `y[i] - x[i]` is in decreasing order (because if there is any `i` such that `y[i] - x[i]` is smaller than the last element in the queue, then it will never be the candidate to produce the maximum sum, as there will be another point with a larger `y - x` value and closer to the current `j`).

Let's breakdown the algorithm implemented in the solution code:

1. Initialize `ans` with the smallest number possible (`-inf`), to store the maximum value of the equation.
2. Initialize an empty deque `q`, where we will maintain our candidates for the maximum `(y[i] - x[i])`.
3. Iterate through each point `(x, y)` in the `points` list:
   - While the deque is not empty and the x-distance from the current point to the front of the deque is greater than `k`, remove the front of the deque. This is because the point at the front of the deque is too far away to consider for the current point `j`.
   - If there is still any point left in the deque after the previous step, update the `ans` with the maximum of `ans` and the sum of the current y, x, and the value at the front of the deque (which represents the maximum `(y[i] - x[i])` for a point within distance `k`).
   - While the deque is not empty and the value of `(y - x)` for the current point is greater than or equal to the value at the back of the deque, pop the back. This is because the new point is a better candidate since it provides a larger or equal value of `y - x` and is closer to future points `j`.
   - Finally, add the current point `(x, y)` to the deque as a candidate for future points.
4. After iterating through all points, return `ans` as the result.

The solution efficiently uses a deque to keep the best candidate points for the maximum sum, updating the possible maximum with each point it examines and maintaining the deque according to the constraints given.

## Solution Approach

The solution utilizes a deque, which is a double-ended queue that supports addition and removal of elements from both ends in O(1) time complexity. This data structure is perfect for our needs because it allows us to maintain the candidates for points `i` that will potentially maximize our equation, while also enabling us to efficiently add new candidates and remove the old ones that are no longer in contention.

Here's the implementation explained step by step:

1. Initialize `ans` with `-inf`, which acts as a placeholder for the maximum value of our equation as we go through each point.
2. Initialize an empty deque `q` to maintain the candidate points `i`.

The deque `q` will store tuples `(x, y)` representing the points and will maintain them in a way where the value of `y - x` is in decreasing order.

3. Begin a loop to go through each point `(x, y)` in the sorted `points` list:
   - First, we remove points from the front of the deque that are farther away than `k` from our current point `x`. This is done with a `while` loop that checks if the deque is not empty and the current `x` minus the `x` of the point at the front of the deque is greater than `k`, and if so, it removes the front point.
   - Next, if there are still points left in the deque after the cleanup, we calculate the value of the equation for the point `j` and the point `i` located at the front of the deque, which has the maximized value of `y[i] - x[i]`. We update `ans` with the maximum value between the existing `ans` and the newly calculated sum.
   - Then, we need to insert the current point `(x, y)` into the deque. Before doing that, we remove all points from the back of the deque that are worse candidates than the current one. A worse candidate is a point whose `y - x` is less than or equal to the `y - x` of the current point. This is because the current point is either equidistant or closer to all future points `j`, making the deque points with a smaller `y - x` irrelevant.
   - Finally, we add the current point `(x, y)` to the back of the deque, as it is now a candidate for future points.
4. After the loop concludes, all pairs of points that could potentially satisfy our constraints have been considered, and `ans` contains the maximum value found.

In summary, by maintaining a list of candidate points in a deque and using a greedy approach to keep only the best candidates as we iterate through the points, the code ensures that we can find the maximum value of the equation `y[i] + y[j] + |x[i] - x[j]|` efficiently, where the absolute difference between `x[i]` and `x[j]` is at most `k`.

The approach effectively combines the monotonic deque pattern with the greedy algorithmic paradigm to tackle the problem.

### Example Walkthrough

Consider the array of points `points = [[1,3],[2,0],[5,10],[6,-10]]` and `k = 1`.

According to our problem, we are looking for maximum value obtained from the equation `y[i] + y[j] + |x[i] - x[j]|`, where `|x[i] - x[j]|` is less than or equal to `k`.

Let's process each point and follow the described approach using a deque (denoted as `q` here):

1. Our initial value of `ans` is `-infinity` because we haven't started, and `q` is empty.
2. We examine the first point:
   - `points[0] = [1,3]`
   - `q` is still empty, so we just add `(x[0], y[0] - x[0]) = (1, 2)` to `q`.
3. Moving to the second point:
   - `points[1] = [2,0]`
   - Before adding the point to `q`, remove points from `q` whose x-coordinate difference is more than `k`. Currently, point `[1,3]` is within `k`, so we keep it.
   - Consider the value for this point `[2,0]` using the point we have in `q`: `ans = max(ans, y[1] + q.front(y-x) + x[1])` which would be `0 + 2 + 2 = 4`.
   - Now, ensure current point's `y-x` is maintained in `q`. `0-2 = -2 < 2`, so `q` doesn't change.
   - The deque `q` now contains `[(1, 2), (2, -2)]`.
4. Now, for the third point:
   - `points[2] = [5,10]`
   - The difference between `x[2]` and `x[0]` is more than `k`, so we remove the first point from `q`.
   - The deque `q` is now `[(2, -2)]`.
   - Calculate `ans`: `ans = max(ans, y[2] + q.front(y-x) + x[2])` which would be `10 - 2 + 5 = 13`.
   - The current point `[5,10]` has `y-x` of `10 - 5 = 5` which is greater than `-2`, so we remove `(2, -2)` from the deque `q` and insert the current point.
   - The deque `q` updates to `[(5, 5)]`.
5. Finally, for the fourth point:
   - `points[3] = [6,-10]`
   - The difference between `x[3]` and `x[2]` is `1` which is within `k`. We don't remove anything from `q`.
   - Calculate `ans`: `ans = max(ans, y[3] + q.front(y-x) + x[3])` which would be `-10 + 5 + 6 = 1`. However, the `ans` is already 13 from the previous step which is greater.
   - Add the current point `[6,-10]` with value `y-x = (-10 - 6) = -16` to the deque `q`. Since `-16` is less than `5`, it's added but will not affect the `ans`.

We've now considered all points, and the maximum value found for `ans` is `13`.

The final `ans` we return is `13`, which is the maximum sum we calculated from the provided points, given the conditions of the problem.

## Python Solution

```python
from collections import deque
from math import inf

class Solution:
    def findMaxValueOfEquation(self, points: List[List[int]], k: int) -> int:
        # Initialize the answer to negative infinity to find the max value later
        max_value = -inf

        # Use a deque to keep track of potential candidates for maximum value
        candidates = deque()

        # Iterate through each point in the given list of points
        for x, y in points:
            # Remove any (x, y) from the deque where the current x
            # minus the candidate's x is greater than k (outside the window)
            while candidates and x - candidates[0][0] > k:
                candidates.popleft()

            # If the deque is not empty, update max_value with the maximum value found
            # considering the equation yi + yj + |xi - xj| with current point and
            # candidate at the front of the deque
            if candidates:
                max_value = max(max_value, x + y + candidates[0][1] - candidates[0][0])

            # Ensure that deque holds the candidates in decreasing order of their value for y - x
            # Pop out the last element if it's smaller than the incoming element
            while candidates and y - x >= candidates[-1][1] - candidates[-1][0]:
                candidates.pop()

            # Append the current point as a candidate for future computations
            candidates.append((x, y))

        # Return the computed max value of the equation
        return max_value

```

## Java Solution

```java
class Solution {
    public int findMaxValueOfEquation(int[][] points, int k) {
        // Initialize answer to a very small number to ensure any valid equation will be larger.
        int answer = Integer.MIN_VALUE;

        // Use a deque to maintain potential candidates for (xi, yi) in a sliding window fashion.
        Deque<int[]> candidates = new ArrayDeque<>();

        // Iterate over all points.
        for (int[] point : points) {
            int x = point[0];
            int y = point[1];

            // Remove points from the start of the deque if their x values are out of the acceptable range (> k).
            while (!candidates.isEmpty() && x - candidates.peekFirst()[0] > k) {
                candidates.pollFirst();
            }

            // If the deque is not empty, compute the potential answer using the current point
            // and the front of the deque, then update the answer if necessary.
            if (!candidates.isEmpty()) {
                answer = Math.max(answer, x + y + candidates.peekFirst()[1] - candidates.peekFirst()[0]);
            }

            // Remove points from the end of the deque if they are no longer preferable.
            // If the current point has a better y - x value, then it is a better candidate.
            while (!candidates.isEmpty() && y - x >= candidates.peekLast()[1] - candidates.peekLast()[0]) {
                candidates.pollLast();
            }

            // Add the current point to the deque as it may be a candidate for future points.
            candidates.offerLast(point);
        }

        // Return the final computed answer.
        return answer;
    }
}

```

## C++ Solution

```cpp
#include <vector>
#include <deque>
using std::vector;
using std::deque;
using std::pair;
using std::max;

class Solution {
public:
    int findMaxValueOfEquation(vector<vector<int>>& points, int k) {
        int maxValue = INT_MIN; // Initialize maximum value to the smallest integer
        deque<pair<int, int>> window; // Deque to maintain the sliding window of valid points

        for (auto& point : points) {
            int x = point[0], y = point[1];

            // Remove points from the front of the deque that are out of the current x's range (distance greater than k)
            while (!window.empty() && x - window.front().first > k) {
                window.pop_front();
            }

            // If the deque is not empty, calculate the potential maximum value using the front element
            if (!window.empty()) {
                maxValue = max(maxValue, x + y + window.front().second - window.front().first);
            }

            // Maintain a monotone decrease in the value of y - x by popping from the back of the deque
            while (!window.empty() && y - x >= window.back().second - window.back().first) {
                window.pop_back();
            }

            // Add the current point to the deque
            window.emplace_back(x, y);
        }

        return maxValue; // Return the computed maximum value
    }
};

```

## Typescript Solution

```typescript
function findMaxValueOfEquation(points: number[][], k: number): number {
    // Initialize the answer to a very small number to start comparisons.
    let maxValue = Number.MIN_SAFE_INTEGER;

    // Queue to maintain the points within the sliding window constraint.
    const monoQueue: number[][] = [];

    for (const [xCurrent, yCurrent] of points) {
        // Remove points from the queue that are outside the sliding window of 'k'.
        while (monoQueue.length > 0 && xCurrent - monoQueue[0][0] > k) {
            monoQueue.shift();
        }

        // If the queue is not empty, calculate the potential answer.
        if (monoQueue.length > 0) {
            maxValue = Math.max(maxValue, xCurrent + yCurrent + monoQueue[0][1] - monoQueue[0][0]);
        }

        // Maintain the elements in the monoQueue so that their y - x value is in decreasing order.
        while (monoQueue.length > 0 && yCurrent - xCurrent > monoQueue[monoQueue.length - 1][1] - monoQueue[monoQueue.length - 1][0])
            monoQueue.pop();
        }

        // Add the current point to the queue.
        monoQueue.push([xCurrent, yCurrent]);
    }

    // Return the maximum value from all calculated values.
    return maxValue;
}

```

## Time and Space Complexity

The given code snippet finds the maximum value of the equation |xi - xj| + yi + yj specified by the problem statement, with the constraint that |xi - xj| <= k for a list of point coordinates. The solution uses a deque to maintain a list of points that are within the distance `k` of the current point being considered.

### Time Complexity:

- The outer `for` loop goes through each of the `n` points once, so it has a time complexity of `O(n)`.
- The `while` loop inside the `for` loop pops elements from the front of the deque until the condition `x - q[0][0] > k` is not met. Each element is added to the deque at most once and removed from the deque at most once. Therefore, even though it looks like a nested loop, the total number of operations this loop will perform over the entire course of the algorithm is at most `n`, hence it contributes to an `O(n)` complexity.
- The second `while` loop within the `for` loop is similar, as it also deals with each dequeued element at most once over the entire course of the algorithm. Thus, it too contributes to an `O(n)` complexity.

Since all of the operations are linear and the loops are working on disjoint operations (you do not have an `O(n)` operation for every other `O(n)` operation), these complexities add up in a linear fashion. Therefore, the overall time complexity of the algorithm is `O(n)`.

### Space Complexity:

- The deque `q` stores at most `n` points at any given time, where `n` is the number of points, in the worst case scenario when all points are within the distance `k` from each other.
- No other data structures or significant variables are used that scale with `n`.

The space occupied by the deque is the dominant factor, which gives us an overall space complexity of `O(n)`.