## 2106. Maximum Fruits Harvested After at Most K Steps

Binary Search Prefix Sum Sliding Window

**Problem Description** 

<u>Array</u>

Hard

and amount\_i. position\_i represents a unique position on an infinite x-axis where fruits are located, and amount\_i is the number of fruits available at that position. Additionally, we have a starting position on the axis, startPos, and an integer k which represents the maximum number of steps we can take in total. The goal is to find the maximum total number of fruits one can harvest. One can walk either left or right on the x-axis, and upon visiting a position, all fruits there are harvested and thus removed from

In this problem, we are given a sorted 2D integer array fruits where each entry fruits[i] consists of two integers, position\_i

that position. Walking one unit on the x-axis counts as one step, and the total number of steps cannot exceed k. The task is to devise a strategy for harvesting the most fruits under these constraints. Intuition

#### To arrive at the solution, we first need to understand that walking too far in one direction might prevent collecting fruits in the

and then potentially changing direction to collect more if the steps allow. The intuition behind the solution is to use a sliding window technique to keep track of the total amount of fruits that can be collected within k steps from startPos. We slide the window across the sorted positions while ensuring the total number of steps including the return to startPos does not exceed k. If at any point the total distance of our sliding window exceeds k, we

other direction because of the step limit k. To maximize the harvest, we should consider walking in one direction to collect fruits

move the starting point of our sliding window to the right to shrink the harvesting range back within the allowed steps. We calculate the maximum number of fruits that can be collected within this range and update our answer accordingly. At each step, we consider reaching the farthest point in our window from startPos and then returning to the nearest point. The heart of this approach relies on the optimal substructure of the problem—maximizing fruits within a range naturally contributes to

maximizing the overall harvest. Solution Approach The solution implements a sliding window approach to maintain a range of positions we can visit to collect fruits within k steps.

The fruits array gives us the positions and the number of fruits at each position, sorted by the positions. The implementation

harvest, i to denote the start of the sliding window, s to keep the sum of fruits within the window, and j to iterate over the fruit

iterates over this array, expanding and shrinking the window to find the optimal total amount of fruits that can be harvested.

### The maxTotalFruits function starts by initializing important variables: ans to store the maximum number of fruits we can

positions. Here is a step-by-step explanation of the algorithm:

2. Check if the current window [i, j] exceeds k steps. This is done by calculating the distance pj - fruits[i][0] (the width of the window) and adding the smaller distance from startPos to either end of the window. 3. If the window is too wide (exceeds k steps), we shrink the window from the left by incrementing i, effectively removing fruits at fruits[i][1] from the sum s.

The core concept here leverages the fact that, since the fruit positions are sorted and unique, we can efficiently determine the

range of positions to harvest by only considering the endpoints of the current window.

4. At each iteration, after adjusting the window, update ans with the maximum of itself or the current sum s.

1. Iterate through the fruit array using the index j and for each position pj with fruit count fj, add fj to the sum s.

The algorithm uses constant space for variables and O(n) time where n is the number of positions since it scans through the positions once.

while i <= j and pj - fruits[i][0] + min(abs(startPos - fruits[i][0]), abs(startPos - pj)) > k:

Let's consider a small example to illustrate the solution approach. Suppose we have the following input:

s -= fruits[i][1] i += 1 ans = max(ans, s)

The loop adds fruits to s and potentially contracts the window by advancing i if the condition (pj - fruits[i][0] +

min(abs(startPos - fruits[i][0]), abs(startPos - pj))) > k is met, ensuring the total steps do not exceed k. The answer

We want to maximize the number of fruits we can pick within those 4 steps.

Here's how the solution approach would work on this example:

Here is a portion of the code explaining the critical part of the algorithm:

5. Continue this process until all positions have been checked.

for j, (pj, fj) in enumerate(fruits):

• fruits array: [[0,3], [2,1], [5,2]]

pj, which still does not exceed k.

Solution Implementation

s += fj

startPos:1

```
ans is updated to the maximum sum s found within the constraints.
  This elegant approach effectively balances expanding and contracting the harvesting range on the fly to maximize fruit collection
  within the step limit.
Example Walkthrough
```

• k:4 Here, we can move a maximum of 4 steps from the starting position, and positions [0, 2, 5] have [3, 1, 2] fruits respectively.

1. Initialize ans as 0 (maximum number of fruits harvested), i as 0 (start of the sliding window), and s as 0 (sum of fruits within the window).

4. Move to the next item in the array, j = 1. The sum of fruits s becomes 3 + 1 = 4. The window [i, j] is now [0, 2], and the total steps

required are abs(startPos - fruits[i][0]) + (fruits[j][0] - fruits[i][0]) = abs(1 - 0) + (2 - 0) = 3 steps to go from startPos to

5. Move to the next item in the array, j = 2. Add fruit count at that position to s, so now s = 4 + 2 = 6. The window [i, j] is now [0, 5], and

#### we check the steps: abs(startPos - fruits[i][0]) + (fruits[j][0] - fruits[i][0]) = abs(1 - 0) + (5 - 0) = 6. This exceeds k, so we

need to shrink the window from the left by incrementing i.

max\_fruits = window\_start = total\_fruits = 0

total\_fruits += fruits\_at\_j

window\_start += 1

equal to k so we keep the window.

6. The new window [i, j] is now [2, 5]. We update s by subtracting the fruits at fruits[i] [1] and calculate steps again: s = 6 - fruits[i]

[1] = 6 - 3 = 3. The steps are abs(startPos - fruits[i][0]) + (fruits[j][0] - fruits[i][0]) = abs(1 - 2) + (5 - 2) = 4, which is

7. As i moves up, s decreases to 2, having removed the fruits at position 0. Since this is still less than k, we continue.

2. Start iterating through fruits array with index j. As we add fj to sum s, the sum of fruits becomes 3 when j = 0.

3. There is no need to shrink the window yet since we are within k steps range (the max steps we can move is 4).

8. Now, ans is updated to a maximum of itself or current sum s, so ans = max(0, 3) = 3. 9. No more positions left, we end with  $\frac{1}{2}$  and  $\frac{1}{2}$ , which is the amount of fruit we harvested.

Therefore, following this approach with our example, the maximum number of fruits that can be harvested is 3.

**Python** class Solution:

# Initialize variables for the answer, starting index of the window, and the sum of fruits within the window

# Iterate over the fruits list with index 'i' and unpack positions and fruits as 'position\_j' and 'fruits\_at\_j'

+ min(abs(startPos - fruits[window start][0]). abs(startPos - position\_j)) # Minimum distance to startPos from eithε

# If the window exceeds 'k' distance, subtract the fruits at 'window\_start' and move window start to the right

# Update the 'max fruits' if the current window's total fruits is greater than the previously recorded maximum

while ( window start <= window\_end</pre> and position i fruits[window start][0] # Distance between the current end of the window and its start

def maxTotalFruits(self, fruits: List[List[int]], startPos: int, k: int) -> int:

# Add the fruits at position 'j' to the running total within the window

# Shrink the window from the left as long as the condition is met

for window end, (position j, fruits at j) in enumerate(fruits):

> k # Check if the total distance is within 'k'

total fruits -= fruits[window\_start][1]

public int maxTotalFruits(int[][] fruits, int startPos, int k) {

// Use a sliding window defined by the indices [i, j].

int positionJ = fruits[j][0]; // Position of the j-th fruit tree

// If the distance from start to the current fruit is more than k

i++; // Move the start of the window to the right

\* The farmer starts at a given position and can move a maximum total distance k.

currentFruits -= fruitPositions[leftIndex++][1];

int fruitCountJ = fruits[i][1]; // Number of fruits at the i-th fruit tree

// after adjusting for the closest path, shrink the window from the left.

// Update maxFruits with the maximum of the current value and currentSum.

return maxFruits; // Return the maximum number of fruits that can be collected

currentSum += fruitCountJ; // Add the fruits from the j-th tree to the current sum

min(abs(startPos - fruits[i][0]), abs(startPos - positionJ)) > k) {

\* Calculates the maximum total number of fruits that can be collected from fruit trees lined up in a row.

\* @param {number[][]} fruitPositions - An array where each element is a pair [position, fruitCount];

currentSum -= fruits[i][1]; // Remove the fruits from the i-th tree from the current sum

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

while (i <= i && positionJ - fruits[i][0] +</pre>

maxFruits = max(maxFruits, currentSum);

int maxFruits = 0; // Stores the maximum number of fruits we can collect

int currentSum = 0; // Stores the current sum of fruits between two points

// Two pointers approach: i is for the start point and j is for the end point

max\_fruits = max(max\_fruits, total\_fruits)

#### # Return the maximum number of fruits that can be collected within 'k' units of movement return max\_fruits

Java

**}**;

**TypeScript** 

class Solution {

```
for (int i = 0, j = 0; j < fruits.length; ++j) {</pre>
            int positionJ = fruits[j][0]; // The position of the j-th tree
            int fruitsAtJ = fruits[i][1]; // The number of fruits at the i-th tree
            currentSum += fruitsAtJ; // Add fruits at the j-th tree to the current sum
            // Adjust the starting point i to not exceed the maximum distance k
            while (i <= j &&
                   positionJ - fruits[i][0] +
                   Math.min(Math.abs(startPos - fruits[i][0]), Math.abs(startPos - positionJ))
                   > k) {
                // Subtract the number of fruits at the i-th tree as we move the start point forward
                currentSum -= fruits[i][1];
                i++; // Increment the start point
            // Update maxFruits with the maximum of current sum and previously calculated maxFruits
            maxFruits = Math.max(maxFruits, currentSum);
        return maxFruits; // Return the maximum number of fruits that can be collected
C++
#include <vector>
#include <algorithm>
class Solution {
public:
    // Function to calculate the maximum total number of fruits that can be collected
    // "fruits" vector holds pairs of positions and fruit counts, "startPos" is the starting position,
    // "k" is the maximum number of steps that can be taken.
    int maxTotalFruits(vector<vector<int>>& fruits, int startPos, int k) {
        int maxFruits = 0; // Store the maximum total fruits that can be collected.
        int currentSum = 0; // Store the current sum of fruits being considered.
```

```
position is the position of a tree, fruitCount is the number of
                                     fruits at that tree.
* @param {number} startPos - The starting position of the farmer.
* @param \{number\} k - The maximum total distance the farmer can move.
* @returns {number} The maximum total fruits that can be collected.
function maxTotalFruits(fruitPositions: number[][], startPos: number, k: number): number {
   let currentFruits = 0;  // The running total of fruits collected within the current window
   // Two-pointer approach to find the optimal range of trees to collect fruits from
   for (let leftIndex = 0, rightIndex = 0; rightIndex < fruitPositions.length; rightIndex++) {</pre>
       // Current fruit position and count
       const [currentPosition, fruitCount] = fruitPositions[rightIndex];
       currentFruits += fruitCount; // Add the fruits from the right pointer's current tree
       // Adjust the left pointer while the total distance required to collect fruits
       // from the range exceeds the maximum distance k
       while (
           leftIndex <= rightIndex &&</pre>
           currentPosition - fruitPositions[leftIndex][0] +
          Math.min(Math.abs(startPos - fruitPositions[leftIndex][0]), Math.abs(startPos - currentPosition)) > k
```

// Subtract the fruits from the left pointer's current tree and move the left pointer to the right

```
// Update the maximum fruits collected if the current total is greater
       maxFruits = Math.max(maxFruits, currentFruits);
   return maxFruits; // Return the maximum fruits collected after checking all ranges
class Solution:
   def maxTotalFruits(self, fruits: List[List[int]], startPos: int, k: int) -> int:
       # Initialize variables for the answer, starting index of the window, and the sum of fruits within the window
       max_fruits = window_start = total_fruits = 0
       # Iterate over the fruits list with index 'j' and unpack positions and fruits as 'position_j' and 'fruits_at_j'
       for window end. (position i. fruits at i) in enumerate(fruits):
           # Add the fruits at position 'j' to the running total within the window
            total_fruits += fruits_at_j
           # Shrink the window from the left as long as the condition is met
           while (
               window start <= window_end</pre>
               and position i
               - fruits[window start][0] # Distance between the current end of the window and its start
               + min(abs(startPos - fruits[window start][0]), abs(startPos - position_j)) # Minimum distance to startPos from eithe
               > k # Check if the total distance is within 'k'
               # If the window exceeds 'k' distance, subtract the fruits at 'window_start' and move window start to the right
               total fruits -= fruits[window_start][1]
               window_start += 1
           # Update the 'max fruits' if the current window's total fruits is greater than the previously recorded maximum
           max_fruits = max(max_fruits, total_fruits)
       # Return the maximum number of fruits that can be collected within 'k' units of movement
       return max fruits
```

# **Time Complexity**

Time and Space Complexity

features a nested loop, each fruit in the fruits list is processed only once due to the use of two-pointer technique. The inner while loop only advances the i pointer and does not perform excessive iterations for each j index in the outer loop. Therefore, every element is visited at most twice, keeping the overall time complexity linear with respect to the number of fruit positions. **Space Complexity** 

The time complexity of the given code is O(N), where N represents the number of pairs in the fruits list. Although the code

The space complexity of the given code is 0(1). The only extra space used is for variables to keep track of the current maximum fruit total (ans), the current sum of fruits (s), and the pointers (i, j) used for traversing the fruits list. This use of space does not scale with the size of the input, and as such, remains constant.