1954. Minimum Garden Perimeter to Collect Enough Apples

Medium Math Binary Search

Problem Description In this problem, we are given an infinite 2D grid where an apple tree is planted at every integer coordinate point. The amount of

to find the smallest axis-aligned square plot centered at the origin (0, 0) that contains at least a certain number of apples, neededApples. The challenge lies in minimizing the perimeter of this square plot while ensuring it includes or exceeds the specified amount of apples. The problem requires us to return the minimum perimeter of this plot.

apples growing on a tree is determined by the sum of the absolute values of its x and y coordinates (i.e., |i| + |j|). Our goal is

Intuition

1. Since the trees are symmetrically placed with respect to both axes, we can focus on any one of the four quadrants and extrapolate for the

others.

integers.

2. The number of apples grows linearly as we move away from the origin along any straight line, but since we're considering a square area, the growth rate of total apples is quadratic.

formula that relates to the number of apples within a square plot.

To solve this problem, we can observe a few things about the distribution of apple trees and their yield:

- The key insight here is that the number of apples within a square plot with side length 2 * L (centered at the origin) can be
- computed in a formulaic way. This formula must take into account the count of apples for trees along the perimeter and inside the square. Due to the quadratic nature of the problem, the number of apples within a plot is related to the sum of a series of

Given these insights, we can calculate the number of apples within a plot of side length 2 * L without explicitly adding apples from each tree.

Our solution uses a binary search approach within a reasonable range (1 to r) to find the minimum side length such that the square plot defined by 2 * L has at least neededApples. Binary search is effective here as it avoids a linear scan of all possible side lengths and instead narrows down by halves, significantly reducing the number of checks needed. The width of the plot is 2 * L, and since the plot is a square, the perimeter will be 4 times the side length, which is 8 * L, giving

By conducting a binary search on the side lengths, it's possible to identify the side length where the number of apples meets or exceeds the neededApples, resulting in the smallest square plot that satisfies the problem constraints. **Solution Approach**

us our final answer. The formula 2 * mid * (mid + 1) * (2 * mid + 1) inside the binary search loop represents a mathematical

step: Initialize the search space with variables 1 (for left) and r (for right) representing the possible range of lengths for the sides

In the provided code solution, a binary search algorithm is used to efficiently find the smallest integer length 'L' such that a

square plot with this length satisfies the condition of containing at least neededApples. Here's how the solution works, step by

of the plot. Note that the range is set between 1 and 100000 which is an assumed limit that ensures our search space

Calculate the midpoint between 1 and r using $(1 + r) \gg 1$. The \gg operator is a bitwise shift that effectively divides by 2,

Perform a binary search within this range. A binary search starts with the midpoint of the current 1 and r and systematically

includes the solution.

finding the midpoint for the next iteration of our search. For the current midpoint, compute the number of apples in the square plot using the formula 2 * mid * (mid + 1) * (2 * mid + 1). This formula is derived from the problem's premise that calculates the number of apples based on the coordinates

of the trees, taking advantage of the symmetry and quadratic nature of apple growth in the grid.

1, focusing on the right half of our current search space for the next iteration.

* (2 * mid + 1) which simplifies to 2 * 2 * 3 * 5, giving us 60 apples.

scan, especially when the range of possible lengths is large.

reduces the search space in half based on whether the condition is satisfied (neededApples are within or on the plot).

Compare the number of apples calculated with the neededApples. If the computed number is greater than or equal to neededApples, it means that a square plot with the current midpoint length is sufficient, and thus we adjust the r variable to mid, effectively reducing our search space to the left half.

Otherwise, if the computed number is less than neededApples, we need a larger plot. Thus, we adjust the 1 variable to mid +

The result is then multiplied by 8 to give the perimeter of the square plot (1 * 8), as there are four sides to the square and each side is 2 * L. The use of binary search is crucial in this solution as it provides a logarithmic time complexity, much more efficient than a linear

The binary search continues until 1 and r converge, meaning we have found the smallest L that meets the condition.

number of apples within or on the plot increases as well, allowing us to find the precise length needed efficiently. **Example Walkthrough**

Let's say we need to find a square plot with at least neededApples = 10 apples. Here's how we would apply the solution approach

We initialize our search space with l = 1 and r = 100000, though in practice, the right bound r could be anything

Note: In the code, 1 eventually holds the value of the minimum required length of the square plot's side that houses the

neededApples. It's important to remember that the binary search relies on the observation that as the side length increases, the

To start our binary search, we calculate the midpoint mid = (l + r) >> 1. Initially, it's ((l + 1000000) >> 1), so mid will be 50000. This is an exaggerated mid value for our small example, but for the purposes of our walk-through, we'll proceed with more reasonable numbers in the following steps.

Let's assume a more practical midpoint mid = 2. We'd calculate the number of apples using the formula 2 * mid * (mid + 1)

Since 60 is greater than 10, our needed number of apples, we adjust our r to be equal to mid, bringing r down from 100000

We won't update 1 because the number of apples at mid is already more than 10, so we need to find if there's an even smaller square that will satisfy the requirement.

both 1 and r are now 1.

the solution to our problem.

Solution Implementation

while left < right:</pre>

else:

return left * 8

} else {

mid = (left + right) // 2

right = mid

left = mid + 1

long low = 1, high = 100000;

public long minimumPerimeter(long neededApples) {

long mid = (low + high) >> 1;

left = mid + 1;

function minimumPerimeter(neededApples: number): number {

// Use binary search to find the smallest perimeter

const mid = Math.floor((left + right) / 2);

if (applesInSquare >= neededApples) {

// Formula: 2 * mid * (mid + 1) * (2 * mid + 1)

// Check if the current square has enough apples

def minimumPerimeter(self, neededApples: int) -> int:

mid = (left + right) // 2

right = mid

left = mid + 1

else:

// If it does, move the right bound to mid

// If it doesn't, move the left bound to mid + 1

// Return the perimeter, which is 8 times the side of the square

Calculate mid-point to partition the search range.

if 2 * mid * (mid + 1) * (2 * mid + 1) >= neededApples:

so we move the left boundary past the midpoint.

If we have enough apples, try to find a smaller square,

hence we move the right boundary to the current midpoint.

If we don't have enough apples, we need a larger square,

The formula calculates the total number of apples within a square with side length 'mid'.

// Calculate the middle point

// Initialize left and right bounds for binary search

return left * 8;

};

TypeScript

let left = 1;

let right = 100000;

while (left < right) {</pre>

right = mid;

left = mid + 1;

} else {

return 8 * left;

// Initialize low and high bounds for binary search

Python

Java

class Solution {

class Solution:

to 2.

described above:

sufficiently large.

Now we recalculate the mid: since we brought down r to 2, the new mid is ((1 + 2) >> 1), which is 1.5, but since we are dealing with integer values, we take mid = 1.

Once again, we see that 12 apples are greater than the 10 needed, so we again adjust r to mid, but since mid is already 1,

Calculate the number of apples with mid = 1 using the formula, which gives us 2 * 1 * 2 * 3 or 12 apples.

Since 1 and r have converged, we have found our minimum L = 1. The final step is to calculate the perimeter of the square plot 1 * 8, which gives us 8.

By condensing the range step by step, we were able to identify that a square plot of side length 2 centered at the origin (0, 0)

results in a square that covers enough apple trees to meet our required number of apples. The perimeter of this plot is 8, which is

def minimumPerimeter(self, neededApples: int) -> int: # Initialize the left and right boundaries for binary search. left, right = 1, 100000# Perform binary search to find the minimum distance from the tree to collect the needed apples.

The formula calculates the total number of apples within a square with side length 'mid'.

We multiply by 8 to get the perimeter of the square (distance from the tree is 'left').

// Apply binary search to find the minimum distance from the tree // at which the number of apples will meet or exceed the 'neededApples' while (low < high) {</pre> // Find the middle point between low and high

// Calculate the number of apples by the formula explained.

if $(2 * mid * (mid + 1) * (2 * mid + 1) >= neededApples) {$

// The formula is for the sum of the first 'mid' odd numbers

// and then multiplied by 4. because the garden has 4 guadrants.

// 2 * mid * (mid + 1) * (2 * mid + 1) directly gives the sum of

// Check if the number of apples within this perimeter is sufficient

// apples in one quadrant, so we compare that sum with 'neededApples'

low = mid + 1; // Adjust the low bound of the search to 'mid' + 1

// The final result is the side length times 8, which is the perimeter of the square.

// left now holds the value of the minimum required side length

// Calculate the number of apples within a (mid x mid) square

const applesInSquare = 2 * mid * (mid + 1) * (2 * mid + 1);

high = mid; // Adjust the high bound of the search to 'mid'

Calculate mid-point to partition the search range.

if 2 * mid * (mid + 1) * (2 * mid + 1) >= neededApples:

so we move the left boundary past the midpoint.

If we have enough apples, try to find a smaller square,

hence we move the right boundary to the current midpoint.

If we don't have enough apples, we need a larger square,

Once binary search is complete, left will hold the minimum distance.

```
// Once we find the closest perimeter that meets the requirement,
        // we multiply it by 8 to get the minimum perimeter of the square.
        // This is because the distance from the tree to the edge of the square
        // on one side forms the perimeter of the square after being multiplied by 4
        // (imagine extending the distance to all four sides of the square),
        // and the condition is checking for each half hence the multiplication by 2.
        return low * 8;
C++
class Solution {
public:
    // This function calculates the minimum perimeter of a square garden
    // from which one can collect at least 'neededApples' apples.
    long long minimumPerimeter(long long neededApples) {
        // Initialize the binary search boundaries
        long long left = 1, right = 100000;
        // Perform binary search to find the smallest side length of the square
        while (left < right) {</pre>
            // Find the middle point between current left and right
            long long mid = (left + right) >> 1;
            // Calculate the number of apples that can be collected from a square garden:
            // The formula is based on an arithmetic series that derives from the pattern
            // in which apples are distributed around the perimeter of the garden.
            long long apples = 2 * mid * (mid + 1) * (2 * mid + 1);
            // Check if the current garden size can meet or exceed the needed apples
            if (apples >= neededApples) {
                // If enough apples can be acquired, bring the right bound inwards
                right = mid;
            } else {
                // Otherwise, increase the left bound to enlarge the garden size
```

```
# Initialize the left and right boundaries for binary search.
left, right = 1, 100000
# Perform binary search to find the minimum distance from the tree to collect the needed apples.
while left < right:</pre>
```

class Solution:

Once binary search is complete, left will hold the minimum distance. # We multiply by 8 to get the perimeter of the square (distance from the tree is 'left'). return left * 8 Time and Space Complexity The given Python code is performing a binary search to find the minimum perimeter of a square that encloses at least neededApples apples. Here's the analysis of time and space complexity: **Time Complexity:**

The binary search algorithm divides the search interval in half each time, which creates a logarithmic time complexity.

Considering that the initial search range is defined between 1 and 100000, the maximum number of iterations this search will

perform is log2(100000) because it's effectively cutting the range in half with each iteration until it finds the perimeter that

The condition checked in the binary search involves a calculation based on mid: 2 * mid * (mid + 1) * (2 * mid + 1). The

evaluation of this arithmetic expression is done in constant time since it only depends on the current value of mid and doesn't

satisfies the condition. Therefore, the time complexity is O(log n) where n is the upper limit of the search range.

grow with the size of the input. Consequently, the overall time complexity of the function is $0(\log n)$, with n being the maximum number 100000.

Space complexity refers to the amount of space or memory taken by an algorithm to run as a function of the length of the

In this code, there are only a fixed number of integer variables (1, r, mid) used, and there's no use of any additional data

structures that grow with the size of the input. Thus, the space used by the algorithm does not depend on the input size. Therefore, the space complexity is constant, 0(1).

Space Complexity: input.