1739. Building Boxes

Hard Greedy Math Binary Search

Problem Description

You are given a task to organize `n` unit-sized cubic boxes into a room that is also a cube with sides of length `n`. While you can place the boxes in any manner on the room's floor, there's a constraint for stacking them:

- If you stack one box on top of another, the box at the bottom (`y`) must have each of its four vertical sides either adjacent to other boxes or to the room's wall.

The goal is to determine the minimum number of boxes that must be placed directly on the floor to accommodate all `n` boxes under the given rules.

Intuition

To solve this problem, visualize the arrangement of boxes in layers, starting from the bottom. The key point is understanding that not all boxes need to go on the floor; they can be stacked on top of each other.

We can place the boxes in a triangular formation on each layer. The first layer will have one box, the second layer will have two additional boxes (making a triangle of 3), and this pattern continues, adding a triangular number of boxes with each new layer. Triangular numbers are given by the formula `k * (k + 1) / 2`, where `k` is the layer level.

First, we find out the maximum height (`k`) such that when we add up all the boxes in these triangular layers (`s`), it is less than or equal to `n`. We do this by iteratively increasing `k` and calculating the sum `s` until `s` plus the next triangular number would surpass `n`.

Once we've established the maximum height, we calculate the total boxes up to this height using the triangular number formula, which gives us the boxes that would be touching the floor if only complete triangular layers were used.

However, we may not yet have accounted for all `n` boxes—there could be some left over that don't form a complete triangular layer. To include these, we start adding them one at a time in the next layer (`k + 1`), increasing the number of floor-touching boxes by 1 each time (since each new box added to the layer needs to touch the floor to maintain stability, as per the given rules), until we reach `n` boxes.

The solution effectively combines these two steps: finding the maximum fully filled triangular layer, then incrementally adding the boxes that don't fit into a full triangular layer, ensuring stability and adherence to the rules specified.

Solution Approach

The solution uses a simple but careful counting approach and capitalizes on the properties of triangular numbers to solve this problem efficiently.

Step 1: Maximizing the Base Layer

Initially, the variables `s` and `k` are set to 0. Variable `s` represents the sum of boxes used so far, and `k` represents the hypothetical height of the stack if it were to be composed of complete triangular layers.

In the `while` loop, we check the sum `s` along with the addition of the next triangular number `k * (k + 1) / 2` to ensure it doesn't exceed the total number `n` of boxes we have. If the condition is satisfied, it means we can place another complete triangular layer on top of the existing stack.

The line `s += k * (k + 1) // 2` accumulates the total number of boxes used in these complete layers, while `k += 1` moves us up to the next layer level. This loop continues until adding another triangular layer would result in `s` being greater than `n`.

Step 2: Completing after Maximizing

After maximizing the base with complete triangular layers, we decrement `k` by 1 because the loop increments `k` one time too many before exiting. At this point, `ans` is set to `k * (k + 1) // 2`, representing the boxes on the floor if only complete layers are used.

The next `while` loop is used for adding the leftover boxes that do not form a complete triangular layer. We start adding them one by one (by incrementing `s` by `k` each time) and for each box added, we increase `ans` by 1 since each box will touch the floor.

The line `s += k` adds a box to the next layer, while `ans += 1` counts the box as touching the floor. The variable `k` increment `k += 1` ensures that we are preparing the count for potentially adding another box on top of the previous ones in an incomplete layer.

This iteration continues until we have placed all `n` boxes (`s < n`), at which point we return `ans`, the accumulated count of how many boxes touch the floor.

No additional data structures are used in this solution, as we only need to keep track of integer counts and sums. The elegance of the solution stems from understanding the problem's geometric nature and how a careful count of the triangular layers converts directly into an algorithm that uses only basic arithmetic operations.

Example Walkthrough

Let's illustrate the solution approach with an example where `n = 10`, representing the total number of boxes we need to place inside the cubic room.

Step 1: Maximizing the Base Layer

We start with `s = 0` and `k = 0`, with `s` indicating the sum of boxes placed so far, and `k` representing the height of the stack with complete triangular layers. We need to iterate and increase `k` to place as many complete triangular layers as we can without exceeding the total number of boxes.

1. For `k = 1`, the number of boxes used would be `k * (k + 1) / 2 = 1 * (1 + 1) / 2 = 1`. Since `s + 1 <= 10`, we update `s` to be `1` and increment `k` to `2`.
2. For `k = 2`, the number of boxes for this layer would be `2 * (2 + 1) / 2 = 3`. Now `s + 3 < 1 + 3 = 4`, which is still less than `10`, so we update `s` to be `4` and increment `k` to `3`.
3. For `k = 3`, the third layer would add `3 * (3 + 1) / 2 = 6` boxes, making the total `s + 6 = 4 + 6 = 10`, which exactly matches our total number of boxes. We update `s` to `10`.

Since adding another triangular layer would exceed `n`, we stop the iterations here with a base layer maximized with `k-1` triangular layers since the last iteration equaled `n`.

The number of boxes on the floor is the sum of complete triangular layers, `s = 10` in this example.

Step 2: Completing after Maximizing

We've already placed all `10` boxes in this example with the step 1 iteration, so there are no leftover boxes to add in incomplete layers. The solution avoids this step because `s == n`, and we already have the required number of boxes on the floor.

Hence, the final answer, which is the number of boxes touching the floor with `n = 10`, is `10`.

In this example, we have demonstrated how the solution maximizes the use of complete triangular layers to get as close as possible to the total number of boxes and then (if necessary) adds the remaining few boxes in the most efficient way adhering to the stacking rules.

Solution Implementation

Python

```python
class Solution:
    def minimumBoxes(self, total: int) -> int:
        # Initialize the sum (s) of placed balls and the count (count_levels) of levels.
        sum_placed_balls, count_levels = 0, 1

        # Calculate the number of levels we can create while we have enough balls.
        # The formula (count_levels * (count_levels + 1) // 2) calculates the
        # number of balls that can be placed in a triangular level with
        # 'count_levels' levels.
        while sum_placed_balls + count_levels * (count_levels + 1) // 2 <= total:
            sum_placed_balls += count_levels * (count_levels + 1) // 2
            count_levels += 1

        # Adjust for the extra count since the last addition in the while loop was not needed.
        count_levels -= 1

        # Calculate the number of boxes used to build the triangular base.
        num_boxes_base = count_levels * (count_levels + 1) // 2

        # Start adding the minimum number of boxes needed on the top level,
        # one by one, until all balls are placed.
        count_top_level = 1
        while sum_placed_balls < total:
            num_boxes_base += 1  # Each time, one more box is added to the top level.
            sum_placed_balls += count_top_level  # Keep track of the total number of balls placed.
            count_top_level += 1  # Increment the top level counter (number of boxes you can add in the next step).

        # Return the total number of boxes needed to place all balls.
        return num_boxes_base
```

Java

```java
class Solution {
    public int minimumBoxes(int n) {
        // Initialize the sum of total balls (s) and the layer level (k)
        int totalBalls = 0;
        int layerLevel = 1;

        // Find the maximum layer level such that the number of balls used is less than or equal to n
        while (totalBalls + layerLevel * (layerLevel + 1) / 2 <= n) {
            totalBalls += layerLevel * (layerLevel + 1) / 2;
            layerLevel++;
        }

        // Decrement layer level since the last addition went over the limit
        layerLevel--;

        // Calculate the initial number of boxes needed to stack the pyramidal structure
        int boxesNeeded = layerLevel * (layerLevel + 1) / 2;

        // Reset layerLevel to start the flat stacking process to use up leftover balls
        layerLevel = 1;

        // While there are balls remaining, stack them flat, one per each layer level
        while (totalBalls < n) {
            boxesNeeded++; // Increment the number of boxes as we place a new ball
            totalBalls += layerLevel; // The number of balls increases by the current flat layer level
            layerLevel++; // Increment the flat layer level
        }

        // Return the total number of boxes needed
        return boxesNeeded;
    }
}
```

C++

```cpp
class Solution {
public:
    int minimumBoxes(int totalCuboids) {
        // Initialize the total number of full floors and the variable to count layers
        int totalFullFloors = 0, layerCount = 1;

        // Keep adding layers until the number of cuboids for the full floors
        // surpasses the total number of cuboids we have
        while (totalFullFloors + layerCount * (layerCount + 1) / 2 <= totalCuboids) {
            totalFullFloors += layerCount * (layerCount + 1) / 2;
            ++layerCount;
        }

        // After finding the last full floor, we move one layer down
        --layerCount;

        // Calculate the total number of boxes (cuboids) used to build the full floors
        int minBoxes = layerCount * (layerCount + 1) / 2;

        // Now add the minimum number of boxes (cuboids) needed to reach the exact number
        // of total cuboids by constructing an incrementally growing pyramid
        layerCount = 1;
        while (totalFullFloors < totalCuboids) {
            minBoxes++;
            totalFullFloors += layerCount;
            ++layerCount;
        }

        // Return the total minimum number of boxes (cuboids) needed
        return minBoxes;
    }
};
```

TypeScript

```typescript
let totalFullFloors: number = 0;
let layerCount: number = 1;

function minimumBoxes(totalCuboids: number): number {
    // Reset the state for each call
    totalFullFloors = 0;
    layerCount = 1;

    // Continue adding layers of cuboids to create full floors until the total number of
    // cuboids for the full floors matches or exceeds the target number of cuboids
    while(totalFullFloors + layerCount * (layerCount + 1) / 2 <= totalCuboids) {
        totalFullFloors += layerCount * (layerCount + 1) / 2;
        layerCount++;
    }

    // After determining the highest full floor, move one layer down
    layerCount--;

    // Compute the total number of cuboids used to construct the full floors
    let minBoxes: number = layerCount * (layerCount + 1) / 2;

    // Add the minimum number of additional cuboids needed to reach the exact total
    // by constructing an incremental step-like structure (growing pyramid)
    layerCount = 1;
    while (totalFullFloors < totalCuboids) {
        minBoxes++;
        totalFullFloors += layerCount;
        layerCount++;
    }

    // Return the minimal number of cuboids needed to reach the total number of cuboids
    return minBoxes;
}
```

```python
class Solution:
    def minimumBoxes(self, total: int) -> int:
        # Initialize the sum (s) of placed balls and the count (count_levels) of levels.
        sum_placed_balls, count_levels = 0, 1

        # Calculate the number of levels we can create while we have enough balls.
        # The formula (count_levels * (count_levels + 1) // 2) calculates the
        # number of balls that can be placed in a triangular level with
        # 'count_levels' levels.
        while sum_placed_balls + count_levels * (count_levels + 1) // 2 <= total:
            sum_placed_balls += count_levels * (count_levels + 1) // 2
            count_levels += 1

        # Adjust for the extra count since the last addition in the while loop was not needed.
        count_levels -= 1

        # Calculate the number of boxes used to build the triangular base.
        num_boxes_base = count_levels * (count_levels + 1) // 2

        # Start adding the minimum number of boxes needed on the top level,
        # one by one, until all balls are placed.
        count_top_level = 1
        while sum_placed_balls < total:
            num_boxes_base += 1  # Each time, one more box is added to the top level.
            sum_placed_balls += count_top_level  # Keep track of the total number of balls placed.
            count_top_level += 1  # Increment the top level counter (number of boxes you can add in the next step).

        # Return the total number of boxes needed to place all balls.
        return num_boxes_base
```

Time and Space Complexity

Time Complexity

The given Python code consists of two while loops that execute sequentially. Let's analyze each part:

- The first while loop runs until the total number of boxes `s` plus the `k`th triangular number is less than or equal to `n`. The `k`th triangular number is computed using the formula `k * (k + 1) // 2`. Since `k` increments by 1 each iteration and `s` increases quadratically, the loop runs in `O(sqrt(n))` time because the number of layers that can be formed (each represented by `k`) is proportional to the square root of `n`.
- The second while loop starts after the first loop ends and increments `s` by `k` each iteration, which increments linearly starting from 1. This loop will run in `O(sqrt(n))` time as well, as it will only go up to the number of boxes required to reach `n`, which is at most what is needed to complete the current layer.

Hence, the total time complexity is `O(sqrt(n)) + O(sqrt(n))`, which simplifies to `O(sqrt(n))`.

Space Complexity

The space complexity of the code is `O(1)` because no additional space that scales with the size of the input `n` is used. Only a constant number of variables `s`, `k`, and `ans` are used to compute the result.