279. Perfect Squares

Medium · Breadth-First Search · Math · Dynamic Programming

Problem Description

The task is to find the smallest number of perfect square numbers which sum up to a given integer `n`. Perfect square numbers are integers that are the product of some integer with itself, such as `1 (1x1)`, `4 (2x2)`, `9 (3x3)`, and so on. The goal is to represent the number `n` as the sum of these perfect square numbers and to do so with the fewest components possible. It's akin to breaking down the number into its square number components, resembling a specialized form of factorization that limits factors to square numbers only.

Intuition

The intuition behind the solution for this problem can be framed through an example from the field of coin change problems where we aim to make change for an amount using the least number of coins. Similarly, we can replace the concept of coins with perfect squares and the amount with our target number `n`. We aim to express `n` using a combination of perfect squares (1, 4, 9, ..., m*m) that adds up to `n` with the minimal count of these squares.

Dynamic programming is the strategy employed here, which builds up a solution by combining solutions to smaller subproblems. By maintaining an array where the `i-th` element represents the least number of perfect square numbers that sum to `i`, this problem can be incrementally solved. We start from the smallest perfect square (1) and work our way up to `n`, updating the array with the minimum count of perfect squares needed to reach each intermediate sum leading up to `n`.

The key is to realize that for any number `j` that we're trying to solve for, if we can subtract a perfect square `i*i` from it, the subproblem now reduces to finding the answer for `j - i*i`. The answer to the original problem (for `j`) would then just be one more than the subproblem (because we've used one perfect square, namely `i*i`). Thus, we loop through all perfect squares less than or equal to `n` and update our array accordingly, always trying to find the smallest number of perfect squares needed for each subsequent value of `j`.

Solution Approach

The reference solution approach adopts a dynamic programming strategy to tackle the problem. The key data structure used is an array `f` of length `n + 1`. The array is initialized such that `f[0]` is `0` (since zero perfect squares sum to zero) and all other elements are set to `infinity` (as placeholders for the minimal value to be found).

Here's a breakdown of the algorithm:

1. Compute the square root of `n` and cast it to an integer (`m`) to find the largest perfect square number we'll need to consider (`m * m`).
2. Initialize our dynamic programming array `f`, with `f[0]` set to `0` and all other values set to `infinity` to indicate that they've not yet been computed.
3. Iterate over each perfect square number `i*i`, with `i` ranging from `1` to `m` (inclusive). For each `i`, update the dynamic programming array.
4. For each perfect square `i*i`, iterate through the array starting from the point `i*i` to `n` (inclusive). For each index `j` in this range, calculate `f[j]` as the minimum of its current value and `f[j - i * i] + 1`.
   - `f[j]` represents the least number of perfect squares that sum to `j`.
   - `f[j - i * i]` represents the least number of perfect squares that sum to the difference between `j` and the current perfect square `i*i`.
   - The `+ 1` accounts for the fact that we're considering including `i*i` as part of the sum for `j`.
5. After all iterations, `f[n]` contains the least number of perfect square numbers that sum to `n`, and this is what the function returns.

The reasoning behind updating `f[j]` with the value `f[j - i * i] + 1` is based on the idea that if some sum `j` can be reached by a minimum number of perfect squares, then adding one more square (in this case `i*i`) would reach a sum of `j + i*i` with just one more perfect square added to the count.

Essentially, this algorithm explores all combinations of perfect squares to find the least number of perfect squares that can sum to any number `k` between `1` and `n`. By caching these results in the array `f`, the solution can efficiently be built from the bottom up, reusing previously computed values for optimal substructures.

This type of dynamic programming is often referred to as bottom-up because the solutions to smaller instances of the problem are used to construct solutions to larger instances. This is also an example of space for time trade-off since we're using extra space (the array `f`) to store intermediate results, thus avoiding recomputation and saving time.

Example Walkthrough

Let's walk through a small example to solidify the understanding of the solution approach described above. Suppose we are given the number `n = 12`. Our goal is to find the smallest number of perfect square numbers that sum up to 12.

Following the steps of the algorithm:

1. We first calculate the largest perfect square number less than or equal to `n`. The square root of `12` is approximately `3.46`, so we round down to get `m = 3`. The perfect squares we will consider are `1*1`, `2*2`, and `3*3`, which are `1`, `4`, and `9` respectively.

2. Next, we initialize our dynamic programming array `f` of length `n + 1 = 13`. We fill it with `infinity` except for the first element: `f[0] = 0`.

3. We begin our iteration over each perfect square, so we'll look at `1`, `4`, and `9` in sequence.

4. For the perfect square `1*1 = 1`, we update `f[j]` for all `j` in the range from `1` to `12` by comparing `f[j]` and `f[j - 1] + 1`:
   - For `j = 1`, `f[1]` becomes `min(infinity, f[0] + 1)` which equals `1`.
   - For `j = 2`, `f[2]` becomes `min(infinity, f[1] + 1)` which equals `2`.
   - ...
   - And so on, until `f[12]` which becomes `min(infinity, f[11] + 1)` which equals `12`.
5. Now we use the perfect square `2*2 = 4`. We update `f[j]` for all `j` from `4` to `12`:
   - For `j = 4`, `f[4]` is updated to `min(f[4], f[0] + 1)` which equals `1` since `f[4]` was `4` and `f[0] + 1` is `1`.
   - For `j = 5`, `f[5]` is updated to `min(f[5], f[1] + 1)` which equals `2`.
   - ...
   - For `j = 12`, `f[12]` is updated to `min(f[12], f[8] + 1)` which equals `3` since `f[12]` was `12` and `f[8] + 1` is `3`.
6. Finally, we consider the perfect square `3*3 = 9`. We update `f[j]` for all `j` from `9` to `12`:
   - For `j = 9`, `f[9]` becomes `min(f[9], f[0] + 1)` which equals `1`.
   - For `j = 10`, `f[10]` becomes `min(f[10], f[1] + 1)` which equals `2`.
   - For `j = 11`, `f[11]` becomes `min(f[11], f[2] + 1)` which equals `3`.
   - For `j = 12`, `f[12]` is finally updated to `min(f[12], f[3] + 1)` which equals `3`. It was `3` from the last update and `f[3] + 1` is `4` which is not smaller than `3`.

After considering all perfect squares up to `3*3`, our array `f` shows that the smallest number of perfect squares summing up to `12` is `f[12]`, which is `3`. Therefore, `12` can be expressed as the sum of three perfect square numbers: `4 (2*2) + 4 (2*2) + 4 (2*2)` or alternatively as `9 (3*3) + 1 (1*1) + 1 (1*1) + 1 (1*1)`.

The dynamic programming solution effectively builds up the optimal solution one number at a time by combining the optimal solutions to smaller subproblems. By completing this example, the algorithm demonstrates how it leverages the computed solutions to previous numbers to find the least number of perfect squares that sum to `n`, step by step.

Python Solution

```python
from math import sqrt
from math import inf

class Solution:
    def numSquares(self, n: int) -> int:
        # Find the square root of n and floor it
        max_square_root = int(sqrt(n))

        # Initialize an array of size n+1 for dynamic programming, filled with infinity for all indices except the first one
        dp = [0] + [inf] * n

        # Loop through all possible square numbers up to the maximum square root
        for i in range(1, max_square_root + 1):
            square = i * i
            # Update the dp array with the minimum number of squares that sum up to each index j
            for j in range(square, n + 1):
                dp[j] = min(dp[j], dp[j - square] + 1)

        # Return the minimum number of squares that sum up to n
        return dp[n]

```

Java Solution

```java
class Solution {
    public int numSquares(int n) {
        int maxSquareRoot = (int) Math.sqrt(n); // Calculate the maximum square root for the given n.
        int[] dp = new int[n + 1]; // Create a dynamic programming array to store minimum squares for each number up to n.
        Arrays.fill(dp, Integer.MAX_VALUE); // Fill the array with maximum int value to simulate infinity.
        dp[0] = 0; // There are 0 squares that sum to 0.

        // Iterate through each square number up to the square root of n.
        for (int i = 1; i <= maxSquareRoot; ++i) {
            int square = i * i; // Calculate the current square number.
            // Calculate the minimum number of squares that sum to each number from the square up to n.
            for (int j = square; j <= n; ++j) {
                // Update the dp array with the minimum value between the current count and
                // the count of the current square plus the count required to sum to the remaining value (j - square).
                dp[j] = Math.min(dp[j], dp[j - square] + 1);
            }
        }
        return dp[n]; // Return the minimum number of squares that sum to n.
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <cmath>
#include <algorithm>
#include <cstring>

class Solution {
public:
    // Function to find the minimum number of perfect square numbers
    // that sum to 'n'.
    int numSquares(int n) {
        // Calculate the square root of 'n' to get the largest possible
        // square number that we will use.
        int maxSquareRoot = sqrt(n);

        // Create a dp (dynamic programming) vector to store the minimum
        // number of squares required for each number up to 'n'.
        // Initialize with a large value as a form of 'infinity'.
        std::vector<int> dp(n + 1, INT_MAX);

        // Base case: 0 can only be expressed as the sum of 0 squares.
        dp[0] = 0;

        // Start building up the solution from 1 to 'maxSquareRoot'.
        for (int i = 1; i <= maxSquareRoot; ++i) {
            int square = i * i; // The square of the current number i
            // For each value 'j' from the current square up to 'n',
            // update the dp array with the minimum value between its
            // current value and the number of squares required to reach
            // (j - current square) plus one (which represents the current square).
            for (int j = square; j <= n; ++j) {
                dp[j] = std::min(dp[j], dp[j - square] + 1);
            }
        }

        // The answer for 'n' will now be at dp[n], which was built up by the loop.
        return dp[n];
    }
};

```

Typescript Solution

```typescript
function numSquares(n: number): number {
    // Calculate the integer square root of 'n'
    const maxSquareRoot = Math.floor(Math.sqrt(n));

    // Initialize an n+1 length array with 'infinity' representing the minimum number of squares to achieve the index value
    const minSquares: number[] = Array(n + 1).fill(Infinity);

    // Base case: 0 can be composed by 0 squares (hence, minSquares[0] = 0)
    minSquares[0] = 0;

    // Loop through all possible perfect square numbers
    for (let squareRoot = 1; squareRoot <= maxSquareRoot; ++squareRoot) {
        const square = squareRoot * squareRoot; // Calculate the current square
        // Iterate through the array and update the minimum number of squares needed to compose 'j'
        for (let j = square; j <= n; ++j) {
            // Update the minSquares for the number 'j' if a lesser number of squares is found
            minSquares[j] = Math.min(minSquares[j], minSquares[j - square] + 1);
        }
    }

    // The last index of minSquares will hold the minimum number of perfect square numbers which sum up to 'n'
    return minSquares[n];
}

```

Time and Space Complexity

The provided code is a dynamic programming solution to find the least number of perfect square numbers which sum to a given number `n`.

Time Complexity:

The outer loop runs `m` times where `m` is the integer square root of `n`, which is calculated by `m = int(sqrt(n))`. This essentially means we are considering squares of all numbers from `1` to `m` (inclusive). The inner loop runs for each value from the current square `i * i` up to `n`. Essentially, the inner loop will run `n` times in the worst case for each `i`.

Hence, the total time complexity is O(`n * m`). Since `m` is the square root of `n`, we can express this as O(`n * sqrt(n)`).

Space Complexity:

The space complexity is determined by the array `f`, which is of size `n + 1`. Thus, the space complexity is O(`n`), as `f` is the only data structure that stores a number of elements linearly proportional to the input size.