964. Least Operators to Express Number

Hard Memoization Math Dynamic Programming Leetcode Link

Problem Description

The problem presents a situation where we need to find the minimum number of operations required to achieve a target number starting with a given positive integer `x`. The only allowed operations are addition (`+`), subtraction (`-`), multiplication (`*`), and division (`/`), and `x` can be used repeatedly in the expression. The challenge is to construct an expression using the fewest possible operations such that the result of the expression is equal to the target number provided.

The constraints are as follows:

- Division results in rational numbers.
- No parentheses allowed in the expression.
- Standard order of operations is followed (multiplication and division precede addition and subtraction).
- Unary negation is not permitted.

The goal is to return the least number of operators used in an expression that equals the target number.

Intuition

To solve this problem, a depth-first search (DFS) algorithm can be used with the help of dynamic programming to store intermediate results. This is because the problem has an overlapping subproblem structure and possesses optimal substructure properties, which are both hallmark traits that benefit from dynamic programming optimization.

We understand that larger numbers can be more efficiently created by multiplication, especially if `x` is large. However, as the number gets closer to the target, it might be more efficient to use addition or subtraction instead of multiplication or division.

The solution involves recursively trying to build up from smaller numbers to the target, using dynamic programming to avoid recalculating subproblems. For each step, the current value could be built by either adding `x`, subtracting `x`, multiplying by `x`, or dividing by `x`. However, the algorithm wisely narrows down the choices by considering that division is not optimal compared to other operations since it requires more operations to get integers unless explicitly needed.

The `dfs` function calculates the minimum number of operations needed to reach a number `v` by comparing the cost (in terms of the number of operations) of reaching `v` by multiplying `x` to itself some number of times then adding or subtracting `x` to this product. It handles the base case when `v` is less than `x`, and it uses recursion for larger values of `v` by finding the least number of operators required for `x**k` closest to but not smaller than `v`, and also for `x**(k-1)`.

The `@cache` decorator is a Python technique that automatically stores the results of each call to the `dfs` function to avoid redundant calculations during the search, thereby speeding up the entire process.

By considering these options, the algorithm minimizes the number of operations at each step and ultimately finds the least number of operations to reach the target number from `x`.

Solution Approach

The solution implements a recursive depth-first search (DFS) algorithm with a dynamic programming approach. It employs memoization to save the results of intermediate computations using Python's `@cache` decorator. The core of the problem is the `dfs` function, which uses recursion to determine the minimum number of operations required to express any intermediate value `v` that will eventually lead to the target.

Here's a step-by-step breakdown of the `dfs` function in the code:

- **Return base case for small `v`**: When the current value `v` that needs to be expressed is less than or equal to `x`, the base case logic calculates the number of operations needed by using `x` either through direct multiplication/addition or subtraction. This is done by the expressions `min(v * 2 - 1, 2 * (x - v))`, which represent the two possible ways to get to `v` using the fewest operations: either by adding `x` to itself `v` times (`v * 2 - 1` accounts for the operations) or combining `(x - v)` subtractions with additions to achieve a shorter path, which might be more optimal as `v` gets closer to `x`.
- **Determine power of `x` required**: If `v` is greater than `x`, the function computes the smallest exponent `k` such that `x**k` is the smallest power of `x` that is just greater than or equal to `v`. This step is necessary because creating large numbers by multiplying `x` by itself is more operationally efficient than adding `x` repeatedly.
- **Recursive calls to strategy**: Once `k` is found, there are two main scenarios catered to by the recursive calls:
  1. If `x**k - v` is less than `v`, this implies that it may be more efficient to reach `x**k` first and then to subtract the difference to `v`. The recursive call `dfs(x**k - v)` calculates the minimal operations to reach from `x**k` to `v`, while `k + dfs(x**k - v)` includes the operations to get to `x**k`. This case also contemplates an alternative, which is not getting entirely to `x**k` but to `x**(k-1)` and then add up from there (`k - 1 + dfs(v - x ** (k - 1))`).
  2. If `x**k - v` is greater than or equal to `v`, the algorithm only considers using the alternative approach of reaching `x**(k-1)` first (`k - 1 + dfs(v - x ** (k - 1))`).
- **Memoization**: The `@cache` decorator above the `dfs` function is responsible for memorizing the results of recursive calls. This ensures that if a particular value of `v` is reached multiple times during the recursive exploration, its minimum number of operations is calculated only once and reused, which drastically reduces computation time.

Finally, the `dfs` function is called with the initial target, and the result represents the least number of operators required to reach the target from `x`.

This algorithm leverages recursion, memoization, and mathematical power operations to effectively explore and calculate the minimum number of steps required to reach the desired result.

Example Walkthrough

Let's use a small example to illustrate the solution approach, considering the problem's statement and intuition. Suppose our starting number `x` is `3`, and our target number is `19`. The goal is to find out the least number of operations needed to reach `19` from `3` using addition, subtraction, multiplication, and division only. Let's walk through how the `dfs` function would work for this example.

- **Base Case**: When the function is called with `v` which is less than or equal to `x`, it returns the minimum steps to reach `v` using either multiple additions or a near equal number of subtractions and additions, whichever is fewer. But in our example, `19` is bigger than `3`, so we do not consider this case initially.
- **Determining the Power of `x`**: Since `19` is more significant than `3`, we look for the smallest power of `x` (`3` in this case) that is just greater than or equal to `19`. We find out that `3` to the power of `3` (`3**3` or `27`) is the smallest power of `3` greater than `19`. So, `k=3` in this scenario.
- **Recursive Strategy**:
  1. **Subtraction First**: We compute `3**k - v`, which results in `27 - 19 = 8`. Because `8` is less than `19`, we can reach `27` by multiplying `3` by itself `3` times and then subtract `8`. To reach `8`, we would call `dfs(8)`. Here, `8` is twice `3` (our `x`) plus `2` (`3*2 + 2 = 8`). So, to get to `8`, we would need `4` operations (two multiplications and two additions). Adding `3` for the previous multiplications to get to `27`, we have `7` operations in total so far to get from `3` to `19` this way.
  2. **Addition First**: We also consider reaching `3^(k-1)`, which is `3**2` or `9`, and then add up from there. To get from `9` to `19`, we need to perform `10` operations (`9 + 3 + 3 + 3 + 3`). Including the two multiplications to get to `9`, we have `12` operations, which is more than the `7` operations from the subtraction approach.
- **Memoization**: The `@cache` decorator ensures results of `dfs(v)` are stored. For example, when `dfs(8)` is calculated, if `8` is required at another point, it won't need to be recalculated, thus, saving computation time.

Given these two strategies and considering memoization that reduces redundant calculations, the algorithm will decide that for `x = 3` and target `19`, the least number of operations to get the target from `x` would be `7`: `3 * 3 * 3 - 3 - 3 - 3 - 2`.

In summary, starting with `3`, to get to `19` with the fewest operators, it's optimal to first hit a power of `x` closest but above the target (`27`), then use subtraction to reduce to the target. The algorithm effectively computes this using recursion and dynamic programming techniques.

Python Solution

```python
from functools import lru_cache

class Solution:
    def leastOpsExpressTarget(self, x: int, target: int) -> int:
        # Use caching to avoid recomputing results for the same value
        @lru_cache(maxsize=None)  # Using LRUCache instead of `cache` (Python 3.9+ feature)
        def dfs(current_value: int) -> int:
            # If x is greater than or equal to the current_value, find the minimal operations required
            if x >= current_value:
                # Calculate the number of operations when adding x or subtracting from x
                return min(current_value * 2 - 1, (x - current_value) * 2)

            # Initialize multiplier k to 2 as x**1 has been checked above
            k = 2
            # Find k such that x**k is just greater than current_value
            while x ** k < current_value:
                k += 1

            # If the power raised to k is very close to current_value, decide to add or subtract
            if x ** k - current_value < current_value:
                # Take the minimum of either subtracting the next power or adding the previous power
                return min(k + dfs(x ** k - current_value),
                           k - 1 + dfs(current_value - x ** (k - 1)))

            # Otherwise, always subtract current_value from the previous power
            return k - 1 + dfs(current_value - x ** (k - 1))

        # Start the recursive function with the initial target value
        return dfs(target)

```

Java Solution

```java
class Solution {
    private int base;
    private Map<Integer, Integer> memo = new HashMap<>();

    public int leastOpsExpressTarget(int x, int target) {
        this.base = x;
        return dfs(target);
    }

    private int dfs(int value) {
        // If the base is greater or equal to the value, we have two expressions:
        // 1) Using '+' base 'value' times: (x/x + x/x + ... + x/x)
        // 2) Using '-' once and '+' (base-value) times: (x - x/x - x/x - ...)
        if (base >= value) {
            return Math.min(value * 2 - 1, 2 * (base - value));
        }
        // If we have already computed the minimum operations for this value, return it
        if (memo.containsKey(value)) {
            return memo.get(value);
        }
        // Start with exponent 'k' at 2 because we have x^1 = x already
        int exponent = 2;
        // Calculate x^exponent while it's less than the 'value'
        long power = (long) base * base;
        while (power < value) {
            power *= base;
            ++exponent;
        }
        // The result is either by subtracting the value from the power of x found
        // Or by adding more to reach the power of x and then subtracting the target value from it
        int operations = exponent - 1 + dfs(value - (int) (power / base));
        // Check if we should also consider the case where the power exceeds the value
        if (power - value < value) {
            operations = Math.min(operations, exponent + dfs((int) power - value));
        }
        // Store the result in memoization map for future reference
        memo.put(value, operations);
        return operations;
    }
}

```

C++ Solution

```cpp
#include <functional>
#include <unordered_map>
using namespace std;

class Solution {
public:
    // Method to find the least number of operations to express the target using integers and the number x.
    int leastOpsExpressTarget(int x, int target) {
        // We use a map to memoize the results of subproblems.
        unordered_map<int, int> memo;

        // A recursive function, defined using a lambda, that does the work.
        // It calculates the least number of operations to express 'value'.
        function<int(int)> dfs = [&](int value) -> int {
            // If x is greater than or equal to the target value, we calculate the minimum number of operations.
            if (x >= value) {
                return min(value * 2 - 1, 2 * (x - value));
            }
            // Check if the result has been memoized; if so, return it.
            if (memo.count(value)) {
                return memo[value];
            }
            int opCount = 2; // The operation count starts at 2 (representing x/x).
            long long power = x * x; // Start with x squared.

            // Increase 'power' by multiplying with x until it just exceeds or equals 'value'.
            while (power < value) {
                power *= x;
                ++opCount; // Increment operation count each time we multiply with x.
            }

            // Compute the minimum operations if we use 'power' just less than 'value':
            int ans = opCount - 1 + dfs(value - power / x);
            // If the remaining value (power - value) is less than the original 'value',
            // it might be more optimal to also consider this path.
            if (power - value < value) {
                ans = min(ans, opCount + dfs(power - value));
            }

            // Memoize the answer for 'value'.
            memo[value] = ans;
            return ans;
        };
        // Start the recursive function with the target value.
        return dfs(target);
    }
};

```

Typescript Solution

```typescript
/**
 * A recursive function to compute the minimum number of operations required
 * to express the target number using the base number 'x', given the constraints of the problem.
 *
 * @param x The base number used for expressions.
 * @param target The target number to be expressed.
 * @return The minimum number of operations needed.
 */
function leastOpsExpressTarget(x: number, target: number): number {
    // A map that caches the minimum number of operations for given target values.
    const numOperationsCache: Map<number, number> = new Map();

    // The depth-first search function that calculates the number of operations recursively.
    const dfs = (currentTarget: number): number => {
        // If x is greater than the current target, calculate the minimum operations directly.
        if (x > currentTarget) {
            return Math.min(currentTarget * 2 - 1, (x - currentTarget) * 2);
        }
        // If the result for the current target is already in the cache, return it.
        if (numOperationsCache.has(currentTarget)) {
            return numOperationsCache.get(currentTarget)!;
        }
        // Initialize 'k' and 'y' for the power expression of 'x'.
        let powerIndex = 2;
        let poweredX = x * x;
        // Increase the power of 'x' until it is greater than or equal to the current target.
        while (poweredX < currentTarget) {
            poweredX *= x;
            powerIndex++;
        }
        // Calculate the minimum operations required when using one less power of 'x'.
        let minimumOperations = powerIndex - 1 + dfs(currentTarget - Math.floor(poweredX / x));
        // Consider the case where the remainder (y - v) is smaller than the target.
        if (poweredX - currentTarget < currentTarget) {
            minimumOperations = Math.min(minimumOperations, powerIndex + dfs(poweredX - currentTarget));
        }
        // Store the calculated result in the cache.
        numOperationsCache.set(currentTarget, minimumOperations);
        // Return the calculated minimum operations.
        return minimumOperations;
    };

    // Start the recursion with the target number.
    return dfs(target);
}

```

Time and Space Complexity

The provided Python code defines a function called `leastOpsExpressTarget` to find the least number of operations needed to express any number `target` using only integers, the operations `+`, `-` and multiplication by `x`, excluding any operation on numbers themselves to get a digit out of them.

Time Complexity:

The time complexity of this code is difficult to determine exactly without more information on the properties of the number `x` and the `target`, as the number of recursive calls to the `dfs` function depends on how these numbers relate to each other. However, we can discuss the broad factors affecting the time complexity:

1. We perform a recursive depth-first search (DFS) via the `dfs` function.
2. For each recursive call to `dfs`, we search for the smallest `k` such that `x**k >= v`, where `v` decreases with each recursion. In the worst case, this search could potentially be linear with respect to the logarithm of `v` base `x`, i.e., `O(log_x(v))`. It would be the height of the recursion.
3. At each recursive call, if `x**k != v`, we have up to two recursive calls: `dfs(x**k - v)` or `dfs(v - x ** (k - 1))`.
4. The use of caching (memoization via `@cache`) improves the time complexity significantly by avoiding repeated calculation for the same `v` values.

Considering the caching and the logarithmic height of the recursion tree, the overall time complexity is expected to be **O(log_x(target) * log_x(target))**. This is because there are at most `O(log_x(target))` levels and each level could potentially cause two recursive calls.

Space Complexity:

The space complexity of this algorithm is primarily determined by the cache and the call stack used for recursion.

1. The cache will store at most `O(log_x(target))` states, as it caches results for each distinct value of `v` that it encounters.
2. The recursion call stack will at most be `O(log_x(target))` deep since that's the maximum depth we can recurse before `v` becomes smaller than `x`.

Therefore, the space complexity is **O(log_x(target))**, dominated by the stack space required for recursion and the additional space for caching the results.