818. Race Car

Hard · Dynamic Programming · Leetcode Link

Problem Description

In this problem, we have an automated car that starts at position 0 on an infinite number line and can move in both positive and negative directions. The position is determined based on a sequence of instructions, consisting of 'A' (accelerate) and 'R' (reverse). When the car receives an 'A' instruction, it moves forward by its current speed and then doubles its speed. When the car receives an 'R' instruction, it changes the direction of its speed without moving, effectively setting its speed to -1 if it was positive, or to 1 if it was negative. The goal is to determine the minimum number of instructions required for the car to reach a specified target position on the number line.

For example, if our sequence is "AAR", the car will execute the following steps:

1. Accelerate: position goes from 0 to 1 (+1), speed doubles from 1 to 2.
2. Accelerate: position goes from 1 to 3 (+2), speed doubles from 2 to 4.
3. Reverse: speed changes from 4 to -1, position remains at 3.

The problem asks for the shortest sequence of instructions to reach the target position.

Intuition

To find the minimum number of instructions (`A` and `R`) to reach a target position, we can use dynamic programming. The intuition behind the system is to progressively build up from position 0 to the target by finding the optimal sequence for each position in between.

Dynamic programming can help us to avoid recalculating the same subproblems repeatedly, as the number of instructions required to reach position `i` might be reused when computing the instructions needed for position `i+1` or others.

We maintain an array `dp` where `dp[i]` contains the minimum number of instructions to reach position `i`. We iterate over each position from `1` to `target`, and for each one:

- If the position is a power of 2 minus 1 (e.g., 1, 3, 7, 15, ...), we can reach it with only accelerations: the number of instructions is equal to the number of bits needed to represent the number in binary (`k`).

- If the position is not a power of 2 minus 1, we have two options. We can either overshoot the target and then reverse (which gives us the equation `dp[i] = dp[2**k - 1 - i] + k + 1`), or we can approach the target by reversing earlier and then moving toward it (which gives the recursive case with `min(dp[i], dp[i - (2 ** (k - 1) - 2**j)] + k - 1 + j + 2)`).

The `min()` function is used to find the smallest number of instructions among the various paths we could take to reach any given position. The goal is to calculate the least amount of instructions that lead us to reach or pass the target, and then possibly reverse to get back to it when we overshoot.

By progressively filling in the `dp` array using these rules, we can find the minimum number of instructions needed to reach the target position.

Solution Approach

The solution employs a dynamic programming approach where we assume that we have already calculated the minimum number of instructions needed to reach all positions up to `target`. To achieve this, we initialize a list `dp` of length `target + 1` with zeroes, which will store the minimum number of instructions required to reach every position.

Here are the steps we take to fill in the `dp` array:

1. Iterate through all positions from `1` to `target` using a `for` loop.
2. For each position `i`:
   - Calculate `k`, which is the number of bits required to represent `i` in binary. This can be done using the `.bit_length()` method in Python.
   - Check if the current `i` is equal to `2**k - 1`, which means it's one less than a power of 2. If it is, we can reach this position by accelerating `k` times with no need for reversal. We set `dp[i]` to `k`.
   - If `i` is not equal to `2**k - 1`, calculate the minimum number of instructions to reach `i` by considering overshooting and then reversing:
     - `dp[i] = dp[2**k - 1 - i] + k + 1`, where we overshoot the target to the next power of 2 minus 1 and then reverse to reach `i`.
     - Another possibility is that we reverse before reaching the power of 2 minus 1, and then drive toward `i` after the reverse. This is done using another loop over `j` which runs from `0` to `k - 1`.
     - For each potential reverse point defined by the `j` iterator, calculate the number of steps as `dp[i - (2 ** (k - 1) - 2**j)] + k - 1 + j + 2` and update `dp[i]` with the minimum between its current value and this new set of steps.
   - The reason to add `1` more step after reversing is that we need one 'R' instruction to change the direction of the speed.

The algorithm follows these procedures and fills up the `dp` array by considering both overshooting and the possibility of reversing early. Once all positions up to the `target` have been considered, the minimum number of instructions required to reach the `target` position is found in `dp[target]`.

In terms of data structures, we use a simple list (`dp`) in Python to hold our dynamic programming states. As for the algorithmic pattern, it is a classic example of dynamic programming where overlapping subproblems are solved just once and their solutions are stored for later use to optimize the overall computation.

Example Walkthrough

Let's illustrate the solution with a small example where our target position is `6`.

1. **Initialization**: We start by initializing an array `dp` of length `7` (as our target is `6`) with zeroes: `dp = [0, 0, 0, 0, 0, 0, 0]`.

2. **Iterate over positions**: We will iterate over each position `i` from `1` to `6` inclusive.

   a. For `i = 1`:

   - The binary representation is `1`, which requires `1` bit, so `k = 1`.
   - Since `1` is `2**1 - 1`, we can reach here with one `A`. So `dp[1] = 1`.

   b. For `i = 2`:

   - The binary representation is `10`, so `k = 2`.
   - We cannot reach exactly `2` with just accelerations since `2` is not `2**2 - 1`. So we check for overshooting:
     - If we overshoot to `3` and reverse, `dp[2] = dp[2**2 - 1 - 2] + k + 1` which is `dp[1] + 2 + 1`, so `dp[2] = 4`.

   c. For `i = 3`:

   - `3` is `2**2 - 1`, so we reach here by two accelerations. `dp[3] = 2`.

   d. For `i = 4`:

   - The binary representation is `100`, so `k = 3`.
   - Since `4` is not `2**3 - 1`, we calculate the overshoot reversal: `dp[4] = dp[7 - 4] + 3 + 1`, which is not yet calculated, so we proceed with the reverse-before-overshoot logic.
   - We try `j` ranging from `0` to `k - 1` which is `0` to `2`. For `j = 0`, we would be reversing when we have moved `1` step (at position `1`), then advancing `3` (`2**j`) more to position `4`, this gives us a possible `dp[4]` value of `dp[0] + 3 + 0 + 2 = 5`.
   - Similarly, we try `j = 1`, giving us the possibility of reversing at position `2` with speed `2`, then adding `1` more step to get to 4. This would result in `dp[2] + 2 + 1 + 2 = 4 + 5 = 9` steps, which is not better.
   - With `j = 2`, we would be reversing at `4` immediately, this will give infinity since we haven't reached `4` yet, so it tells us not to choose this step.
   - The minimum among these steps is `5`, so `dp[4] = 5`.

   e. Continue this process up to `i = 6`:

   - For `i = 5`, the number of instructions required can be calculated by considering various possible reversal paths like before.
   - For `i = 6`, we use the same methodology.

3. **Final output**: After filling in the `dp` array, we'd find `dp[6]` to get the minimum number of instructions to reach position `6`.

Through such iteration and checks, we build an optimal solution for smaller targets, which then helps us get the optimal solution for our actual target through a series of comparisons and dynamic updating of the `dp` array.

Python Solution

```python
class Solution:
    def racecar(self, target: int) -> int:
        # Initialize the dynamic programming array with the number of operations
        # for each position from 0 to target as zero.
        num_operations = [0] * (target + 1)

        # Iterate over each position from 1 to the target.
        for position in range(1, target + 1):
            # Calculate the minimum number of bits needed
            # to represent the position as this will indicate
            # the minimum sequence of 'A's needed to reach or pass this position.
            num_bits = position.bit_length()

            # Check if the position is a power of 2 minus 1. If so, the number
            # of operations is equal to the number of bits.
            if position == 2 ** num_bits - 1:
                num_operations[position] = num_bits
                continue

            # If the position is not a power of 2 minus 1, calculate the number
            # of operations by reversing before we reach the position.
            num_operations[position] = num_operations[2 ** num_bits - 1 - position] + num_bits + 1

            # Check all possible points where we can reverse before reaching
            # the position to see if there is a more optimal number of operations.
            for reverse_bit in range(num_bits - 1):
                distance_after_reverse = position - (2 ** (num_bits - 1) - 2 ** reverse_bit)
                num_operations[position] = min(
                    num_operations[position],
                    num_operations[distance_after_reverse] + num_bits - 1 + reverse_bit + 2
                )

        # Return the number of operations needed to reach the target position.
        return num_operations[target]

```

Java Solution

```java
class Solution {
    public int racecar(int target) {
        // dp array to store the minimum number of instructions to reach each position
        int[] dp = new int[target + 1];

        // Iterate through all positions up to the target
        for (int position = 1; position <= target; ++position) {
            // Calculate the number of bits needed to represent 'position',
            // effectively representing the minimum sequence of Accelerations
            int k = 32 - Integer.numberOfLeadingZeros(position);

            // If the position is 2^k - 1, exactly k accelerations are needed;
            // this is the best case, no reversals needed
            if (position == (1 << k) - 1) {
                dp[position] = k;
                continue;
            }

            // Overshoot position then reverse. Compute the steps taken to reverse
            // from the next power of 2 position, add k accelerations and 1 reverse instructions
            dp[position] = dp[(1 << k) - 1 - position] + k + 1;

            // Try all possible positions that can be reached by reversing earlier
            // and check whether it would yield a smaller result
            for (int j = 0; j < k; ++j) {
                // The number of steps required includes:
                // - k - 1 accelerations before the reversal
                // - the instructions to reach the remaining distance after the reversal
                // - another reverse (1 instruction)
                // - the extra accelerations taken after the first reverse (j instructions)
                int stepsBeforeReverse = k - 1;
                int extraAccelerations = j + 2; // includes reverse after acceleration sequence
                int remainingDistance = position - (1 << (k - 1)) + (1 << j);
                dp[position] = Math.min(dp[position], dp[remainingDistance] + stepsBeforeReverse + extraAccelerations);
            }
        }
        // Return the minimum number of instructions to reach the target position
        return dp[target];
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <algorithm>
using namespace std;

class Solution {
public:
    int racecar(int target) {
        // Initialize a dynamic programming array to store the minimum number of steps
        // to reach each position up to the target.
        vector<int> dp(target + 1);

        // Iterate over all positions from 1 to the target.
        for (int position = 1; position <= target; ++position) {
            // Calculate the number of bits needed to represent the position in binary,
            // which corresponds to the number of consecutive acceleration 'A' needed
            // if one deceleration 'R' is allowed to reach at or beyond the target.
            int bitLength = 32 - __builtin_clz(position);

            // If the position is exactly one less than a power of 2.
            // This means the car can reach the position with only accelerations
            // and without any deceleration.
            if (position == (1 << bitLength) - 1) {
                dp[position] = bitLength;
                continue;
            }

            // Case 1: Go past the target, then reverse and come back to target.
            dp[position] = dp[(1 << bitLength) - 1 - position] + bitLength + 1; // "+ 1" for reverse

            // Case 2: Stop before the position, reverse and drive to the target.
            for (int backSteps = 0; backSteps < bitLength; ++backSteps) {
                int distanceCovered = (1 << (bitLength - 1)) - (1 << backSteps);
                dp[position] = min(dp[position],
                                   dp[position - distanceCovered] + bitLength - 1 + backSteps + 2); // "+ 2" for two reverses
            }
        }

        // Return the minimum number of steps to reach the target.
        return dp[target];
    }
};

```

Typescript Solution

```typescript
function racecar(target: number): number {
  // Initialize an array to store the minimum number of steps to reach each position up to the target.
  let dp: number[] = new Array(target + 1);

  // Iterate over all positions from 1 to the target.
  for (let position = 1; position <= target; ++position) {
    // Calculate the number of bits required to represent the position in binary.
    // This corresponds to the number of consecutive accelerations 'A' needed.
    let bitLength: number =  Math.floor(Math.log2(position)) + 1;

    // If the position is exactly one less than a power of 2,
    // the car can reach the position with only accelerations and without any deceleration.
    if (position === (1 << bitLength) - 1) {
      dp[position] = bitLength;
      continue;
    }

    // Case 1: Go past the target, reverse and come back to the target.
    dp[position] = bitLength + 1 + dp[(1 << bitLength) - 1 - position]; // "+ 1" for the reverse operation

    // Case 2: Stop before the target, reverse and drive towards the target.
    for (let backSteps = 0; backSteps < bitLength; ++backSteps) {
      let distanceCovered: number = (1 << (bitLength - 1)) - (1 << backSteps);
      dp[position] = Math.min(dp[position],
                              dp[position - distanceCovered] + bitLength - 1 + backSteps + 2); // "+ 2" for two reverse operations
    }
  }

  // Return the minimum number of steps required to reach the target.
  return dp[target];
}

```

Time and Space Complexity

The provided code is a dynamic programming solution for the "Race Car" problem. Here's the analysis of its time and space complexity:

Time Complexity

1. The outer loop runs from `1` to `target`, hence the first part of the time complexity is `O(target)`.

2. In the calculation of `dp[i]`, `i.bit_length()` takes `O(log(i))` time, since it's equivalent to the number of bits required to represent `i`.

3. Within the outer loop, besides direct assignments and one call to the bit_length method, there are two loops:

   - The first loop checks if `i` is one less than a power of two and assigns a value to `dp[i]` directly. This check is done in constant time, but as it's inside the outer loop it doesn't add a dimension to the complexity.
   - The second, inner loop runs from `0` to `k - 1`, where `k` correlates to `i.bit_length()`, thus `k` is at most `log(i)`. Within this loop, `k` iterations may occur at most. Given that the highest possible value for `i` is `target`, the maximum value for `k` is `O(log(target))`.

4. Inside the inner loop, the operations can be considered constant except for the recursive state lookups, which are `O(1)` each due to direct indexing into an array.

5. Therefore, the time complexity inside the inner loop is `O(log(target))` for each loop iteration, with up to `log(target)` iterations.

6. Combining the above, the total time complexity of the code is `O(target * log(target)^2)`, because for each value of `i` (up to `target`) there is a nested loop that can iterate `log(target)` times, and within it another loop can iterate up to `log(target)` times.

Space Complexity

1. The space complexity is dominated by the size of the `dp` array, which has `target + 1` elements.

2. Hence, the space complexity is `O(target)` because it is proportional to the size of the input, `target`.