Lucky Bus Ticket Numbers

Ori Yonay

September 5, 2023

1 Introduction

In Russia, bus tickets are assigned as 6-digit numbers. A ticket number is considered "lucky" if the sum of its first three digits equals the sum of its last three digits (e.g., 437905: $4 + 3 + 7 = 9 + 0 + 5$). We pose the following question: How many such "lucky" numbers exist? We present three algorithms for computing the total number of lucky bus ticket numbers and analyze their efficiency and scalability.

2 Idea 1: The Brute-Force Approach

2.1 Algorithm Description

The brute-force approach to solve this problem is straightforward. For each number in our desired range, we split it into two parts: the first three digits and the last three digits. We then compute the sum for both parts and compare them. If the sums are equal, the number is considered a lucky number.

Algorithm 1 Brute-Force Approach

```
procedure CountLuckyNumbers
    total ← 0
    for i ← 0 to 999999 do
        left ← first three digits of i
        right ← last three digits of i
        sum_left ← sum of digits in left
        sum_right ← sum of digits in right
        if sum_left = sum_right then
            total ← total + 1
        end if
    end for
    return total
end procedure
```

2.2 Intuition for Correctness

The brute-force algorithm directly checks every number within the desired range by splitting the number into its first three and last three digits, computing the sums, and comparing them. This approach guarantees correctness since it considers every possible six-digit number individually.

2.3 Performance

The brute-force approach goes through all numbers from 0 to 999999, implying it has a time complexity of $O(10^6)$, or more specifically $O(10^d)$ for d-digit bus numbers. For each number, the algorithm computes the sum of the digits twice (for both halves), adding a constant factor to each iteration. While it provides the correct result, it is inefficient for larger ranges.

3 Idea 2: The Divided Approach

3.1 Algorithm Description

Checking every number in such a brute-force manner is very redundant. For example, consider the thousand six-digit numbers that start with the digits 456. For each of these, the brute-force algorithm re-computes 4+5+6 for the left half even though it does not change. It might occur to the reader that using a nested loop is more efficient: it would allow us to compute the left sum once, which is about twice as fast. Taking this idea all the way to its logical conclusion, we notice that it would be far more efficient to *pair up three-digit numbers ourselves*. For example, if we know that there are 75 unique three-digit numbers that add up to 14, then the total number of lucky numbers where each half adds up to 14 is $75^2 = 5625$, i.e., every possible pairing of the two halves. Instead of checking every number directly, the divided approach optimizes the calculation by recognizing patterns in the sum of the digits.

Algorithm 2 Divided Approach

```
1: procedure DividedCountLuckyNumbers
2:     Initialize sums with keys from 0 to 27 and values as 0
3:     for i ← 0 to 999 do
4:         digit_sum ← sum of digits in i
5:         sums[digit_sum] ← sums[digit_sum] + 1
6:     end for
7:     total ← sum of squares of all values in sums
8:     return total
9: end procedure
```

3.2 Intuition for Correctness

The algorithm computes the number of times a sum can be achieved for three-digit numbers. By pairing every three-digit number's sum with itself, we ensure every possible combination is considered. Squaring the count for each sum gives us the number of lucky ticket numbers for that sum, ensuring correctness.

3.3 Performance

The algorithm improves efficiency by reducing the computations involved in summing digits. It iterates through 1000 three-digit numbers, computes their sums, and tallies the results. This approach achieves a time complexity of $O(10^3)$, or more generally $O(10^{\frac{d}{2}})$, a significant improvement over the brute-force approach.

4 Idea 3: Efficient Computation of Sums Array

4.1 Algorithm Description

Building upon the Divided Approach, the Efficient Computation optimizes the calculation of the 'sums' array by leveraging the Fast Fourier Transform (FFT). Instead of iterating through numbers and computing their sums, we can use convolution to simulate adding digits.

Figuring out the number of three-digit numbers that add up to a given sum can be thought of as the number of unique ways to roll three ten-sided dice and achieve that given sum. This distribution is well-known and follows a predictable pattern.

Consider each digit (0-9) as a possibility for each position in our three-digit number. The sum of all possibilities for a single position can be represented as a vector of ones (representing equal probability of being sampled, though we are not sampling these numbers) with a length of 10 (since there are ten possible digits). Repeatedly convolving this vector with itself essentially simulates "rolling another die" and results in an array whose indices represent possible sums and the values at those indices represent the count of ways to achieve those sums.

4.2 Intuition for Correctness

To prove the correctness of this method, we need to verify that the convolution approach represents the number of ways to achieve each sum for three digits.

Base Case: For a single digit, the number of ways to get each sum from 0 to 9 is 1, which is represented by our initial ones vector a.

Convolution: Convolution of two functions f and g is essentially a measure of how much f is influenced by a reversed and translated version of g. When we convolve our vector *sums* (which is initially equal to a) with a, we simulate adding another digit.

Algorithm 3 Efficient Computation of Sums Array using Convolution

```
1: procedure ComputeSumsArray
2:     Initialize vector a with ten ones
3:     Initialize sums with ten ones
4:     for i ← 1 to 2 do
5:         sums ← convolve(sums, a)
6:     end for
7:     total ← sum of squares of all values in sums
8:     return total
9: end procedure
```

To understand this, consider the specific case of finding the number of ways to achieve a sum of 7 with two digits. We can achieve this sum by $1+6, 2+5, 3+4, 4+3, 5+2$, or $6+1$. The convolution at index 7 for two vectors, both initialized as a, gives us this count.

More formally, the convolution value at a particular index i is:

$$(sums * a)[i] = \sum_{j=0}^{i} sums[j] \times a[i-j]$$

Each term in the sum corresponds to a unique way to add two numbers (from two separate instances of a) to get the sum i.

Thus, when we convolve the *sums* vector with a for the second time, we simulate adding a third digit. The final *sums* array then accurately represents the number of ways to achieve every possible sum from three digits.

Conclusion: The convolution method correctly computes the number of ways to get each sum for three digits, and thus, the algorithm is correct.

4.3 Performance

The convolution-based approach significantly reduces the complexity of computing the sums array. Convolution in the time domain would have a complexity of $O(d^2)$, but by leveraging the fast Fourier transform, we reduce this to $O(d \log d)$. Since we're dealing with a fixed length (the number of possible digits), the improvement in speed is evident, making this method much faster than the divided approach for computing the sums array.