# Group 7

(External submission)
Final time: 27.0ms (Place 1)

Presented by John Walbran

# Methodology

1. The algorithm first checks whether the length of the list is larger than 10^(digit_count), and will store the first 99 prime factor sums in a hashtable (which gives constant time membership checks). (If the list is shorter, it skips this step)
2. It then uses streams to sort the list, by creating a long for each number, putting the sum of the prime factors of a number i in the first 32 bits (making the prime factors the first sorting criteria) the last 32 bits are used to store -i, which provides the second sorting criteria. The whole list is sorted in reverse, giving the correct sorting order.
3. The getSumPrimeFactors() method works by checking odd numbers starting from 3, going up to the sqrt(n) dividing out that factor as far as possible, guaranteeing that any new factors found are prime. (Then adding the remainder to the sum if there's a significant remainder)
4. The output is printed with leading 0s returned.

# A comment on correctness

- There were a couple of concerns about not clearing data between JVM warmup runs. This turned out to not be the case. Given this, there are no serious concerns about correctness.

# Runtime Analysis

- The getSumPrimeFactors() method runs in O(sqrt(n)) time, regardless of width L. While the overall sorting uses java's built in sort() method on a stream of Longs, which is universally O(nlog(n))
- Combining these would give an expected complexity of O(n * sqrt(n) + nlog(n))
    - nlog(n) > n*sqrt(n) for n > 4, so the n*sqrt(n) is absorbed, leaving a total time complexity of O(nlogn)

# Possible further optimizations

- In the getSumPrimeFactors() method, instead of looping through all odd numbers starting from 3, you can instead check all numbers equivalent to 1 and 5 mod 6, starting from 5. (After dividing out 2 and 3 specifically), which reduces the number of comparisons in the loop by ⅓.
- Since there is an intermediary n as the prime factor loop divides out factors, if we store results as we find them, and then check n against the known table, we can further reduce the number of checks needed.
    - For example, if we want getSumPrimeFactors(56), and we know getSumPrimeFactors(18), we can see after dividing out 3, from 56, we have an temporary sum of 3, and we can add 3 to the given sum for 18, and we can update the table, and return more quickly.