# Group 3 Presentation

Harry Lyu, Jaydon Stanislowski

# Group 3 results

**First prelim**

- Time: 1000021 ms (sum of medians w/ penalty, runtime error)
- Score: 4th place

**Second prelim**

- Time: 193 ms (sum of medians, no penalty)
- Score: 1st place

**Final**

- Time: 199 ms (sum of medians, no penalty)
- Score: 3rd place

**Overall**

1st place out of the class

No correctness issues were reported with our final algorithm.

# Evaluating sum of prime factors

```java
/**
 * Computes and stores the sum of prime factors for the given number 'n'.
 *
 * @param n    The number for which to calculate the sum of prime factors.
 * @param temp A HashMap to store the sum of prime factors for each number.
 */
private static void sumPrimeFactors(int n, HashMap<Integer, Integer> temp) {
    // Check if we already computed the sum of prime factors for this number
    if (temp.containsKey(n)) {
        return;
    }
    int sum = 0; // Variable to hold the sum of prime factors
    int originalN = n; // Store the original value of 'n'

    // Loop to find prime factors up to the square root of 'n'
    for (int i = 2; i <= Math.sqrt(n); i++) {
        if (n % i == 0) { // If 'i' is a factor of 'n'
            sum += i; // Add 'i' to the sum
            while (n % i == 0) { // Remove all instances of 'i' from 'n'
                n /= i;
            }
        }
    }

    // If 'n' is greater than 1, then it is a prime number and a factor of itself
    if (n > 1) {
        sum += n;
    }
          You, 23 seconds ago • Uncommitted changes
    // Store the computed sum in the HashMap
    temp.put(originalN, sum);
}
```

How the for-loop works:

1. Evaluate the first factor (between 1 and √N) of N. This is also the first prime factor, we'll add it to the sum and call it f.

2. Divide N by f until we can't do so anymore and set N to the last value in the chain.

(i.e., if N = 72, f = 2,

72 / 2 / 2 / 2 = 9 ... 9 % 2 ≠ 0, so set N = 9)

3. Evaluate the next potential, smallest factor of old N; this is a prime factor if it is divisible by new N. Repeat the process until we have gone through all potential prime factors of the old N.

This gives us the sum of prime factors of the old N.

As an example: N = 72

# Sorting by sum of prime factors

```java
/**
 * The sorting is done in ascending order of the sum of prime factors.
 * If two numbers have the same sum, they are sorted in descending order of their values.
 */
private static void sort(String[] toSort) {
    HashMap<Integer, Integer> temp = new HashMap<>(); // HashMap to hold the sum of prime factors

    // ████calculate the sum of prime factors for each number in the array
    for (String s : toSort) {
        int num = Integer.parseInt(s);
        sumPrimeFactors(num, temp);
    }

    Arrays.sort(toSort, (a, b) -> {
        int numA = Integer.parseInt(a);
        int numB = Integer.parseInt(b);

        // Compare based on the sum of prime factors
        int compare = temp.get(numA) - temp.get(numB);

        // If the sums are equal, compare based on the actual number values
        if (compare == 0) {
            return numB - numA;
        }

        return compare;
    });
}
```

1. Take each String in toSort as an Integer and call sumPrimeFactors() on it.

- This will place its value in the HashMap along with its sum of prime factors. We can look this up later in O(1) time and avoid repeating extra work.

2. Call Arrays.sort() (using Java's default implementation of Timsort, which is comparison-based)

- We pass in a lambda function specifying how each pair of values should be compared in the sorting operation.
- We do this by parsing both Strings to Integers again, then comparing their values in the HashMap (their sums of prime factors).
- If their sums are equal, make sure that the smaller number goes first.

# Complexity analysis: sumPrimeFactors()

```
        if (temp.containsKey(n)) {
            return;
        }
```

containsKey method usually is O(1) ignoring hash function collisions.

```
        for (int i = 2; i <= Math.sqrt(n); i++) {

            if (n % i == 0) { // If 'i' is a factor of 'n'
                sum += i; // Add 'i' to the sum
                while (n % i == 0) { // Remove all instances of 'i' from 'n'
                    n /= i;
                }
            }
        }
```

The for-loop runs in O(√k), where k is the number for which we are evaluating the sum of prime factors (called n in the method).

The while-loop runs in O(logk) because there are cases where we continue to divide n by i until n = 1.

So, the total complexity would be O(√k + logk), which is O(√k).

```
        if (n > 1) {
            sum += n;
        }

        // Store the computed sum in the HashMap
        temp.put(originalN, sum);
```

The if-statement runs in O(1).

temp.put(originalN, sum) is O(1), ignoring hash function collisions.

The time complexity of this method is O(√k).

# Complexity analysis: sort()

```
        for (String s : toSort) {
            int num = Integer.parseInt(s);
            sumPrimeFactors(num, temp);
        }
```

This is O(n*(√k+L)). We assume k is the average value of the dataset, n is the number of elements in the array and L is the length of the numbers

```
        Arrays.sort(toSort, (a, b) -> {
            int numA = Integer.parseInt(a);
            int numB = Integer.parseInt(b);

            // Compare based on the sum of prime factors
            int compare = temp.get(numA) - temp.get(numB);

            // If the sums are equal, compare based on the actual number values
            if (compare == 0) {
                return numB - numA;
            }

            return compare;
        });
```

The complexity of Arrays.sort() is O(n*logn) on average

So the complexity of our algorithm is O(n*(√k +L)+ L*n*logn).

# How we arrived at our algorithm

```
private static void sort(String[] toSort) {
    int N = Integer.MAX_VALUE / 5;
    int[] sumPrimeFactors = new int[N + 1];
    for (int i = 2; i <= N; i++) {
        if (sumPrimeFactors[i] == 0) {
            for (int j = i; j <= N; j += i) {
                sumPrimeFactors[j] += i;
            }
        }
    }
    int[] parsedNumbers = new int[toSort.length];
    for (int i = 0; i < toSort.length; i++) {
        parsedNumbers[i] = Integer.parseInt(toSort[i]);
    }

    Arrays.sort(toSort, (a, b) -> {
        int numA = Integer.parseInt(a);
        int numB = Integer.parseInt(b);

        int compare = sumPrimeFactors[numA] - sumPrimeFactors[numB];
        if (compare == 0) {
            return numB - numA;
        }
        return compare;
    });
}
```

At first, we tried to use an array to pre-compute the sum of prime factors of each number. However, we realized that we wasted lots of memory and it was not efficient in terms of small size of data. Plus, we couldn’t determine the size of the array.

# How we could have done better

- While analyzing, we noticed that the for-loop in the sumPrimeFactors() method is not as efficient as it could be.
- When N = 1, it’s no longer necessary to continue evaluating prime factors, since 1 will not be divisible by any of the numbers we iterate through.
- This is extra computation done for no reason, though it makes the algorithm slightly easier to analyze.

```
// Loop to find prime factors up to the square root of 'n'
for (int i = 2; i <= Math.sqrt(n); i++) {
    if (n % i == 0) { // If 'i' is a factor of 'n'
        sum += i; // Add 'i' to the sum
        while (n % i == 0) { // Remove all instances of 'i' from 'n'
            n /= i;
        }
    }
}
```