Group 2 Sorting

Chenfei Peng and Brendan Conroy

What changes did we make? (1/2)

- We made most of our changes to the CalculatePrimeFactors method.
- We implemented the sieve of eratosthenes algorithm for returning all prime numbers up to n (we modified it in our algorithm to return all prime factors of n up to sqrt(n)
- We implemented the functionality of the isPrime boolean method within this.

```java
// implements modified sieve of eratosthenes algorithm
// to calculate the sum of the prime factors of a given number n.
private static int calculatePrimeFactors(int n) {
    // No prime factors for non-positive integers and 1
    if (n <= 1) {
        return 0;
    }
    int sum = 0;
    int sqrtN = (int) Math.sqrt(n);
    // initializes logical array as true,
    // which means that we assume all numbers
    // to be prime in the beginning.
    boolean[] isPrime = new boolean[sqrtN + 1];
    Arrays.fill(isPrime, val:true);
    // implements sieve of eratosthenes algorithm
    for (int p = 2; p * p <= sqrtN; p++) {
        if (isPrime[p]) {
            for (int i = p * p; i <= sqrtN; i += p) {
                isPrime[i] = false;
            }}}
    //Calculate the sum of the prime factors
    for (int p = 2; p <= sqrtN; p++) {
        if (isPrime[p] && n % p == 0) {
            sum += p;
            while (n % p == 0) {
                n /= p;
            }
        }
    }
    if (n > 1) {
        sum += n; // n is prime and we add it to the sum.
    }
    return sum;
}
```

Sieve of Eratosthenes Sources:

- Chat GPT
- https://www.geeksforgeeks.org/sieve-of-eratosthenes/

What changes did we make? (2/2)

- Caching prime sums
  - We rewrote the GetSumPrimeFactors method to have a cache.

```
private static Map<Integer, Integer> primeFactorSumCache = new HashMap<>();

private static int getSumPrimeFactorsCached(int n) {
    if (primeFactorSumCache.containsKey(n)) {
        return primeFactorSumCache.get(n);
    }

    // Calculate the sum of the prime factors here and store the result in the cache.
            // This can help us avoid redundant calculations when dealing with repetitive prime-sums.
    int sum = calculatePrimeFactors(n);
    primeFactorSumCache.put(n, sum);

    return sum;
}
```

Correctness?

- Unfortunately, the way that we implemented the cache resulted in our algorithm being incorrect.
- This is due to our cache hashmap storing hashes from the JVM warmup run before the actual runs. This gave our algorithm an unfair advantage at the start.

Score and times:

Round 1: 1st place

Round 2: Tied for 1st place with group 7

Round 3: 5th place

Overall placing: 2nd place (disqualified for incorrectness)

group 2 took place 2. The sum of places is 7, the sum of medians is 205.0

Run 1: 400 9 digit numbers (1st place)

Group 2:

1

1

1

Median: 1.0

Run 2: 1000 7 digit numbers (tied for 1st place)

Group 2:

1

1

1

Median: 1.0

Run 3: 100,000 4 digit numbers (5th place)

Group 2:

203

223

176

Median: 203.0

Takeaways from results

Our algorithm is more efficient for smaller datasets with larger length integers (runs 1 and 2) than larger datasets with smaller length integers (run 3).

Why?

An idea: for larger datasets with smaller length integers like run 3, the workload of calculating the sum of prime factors for one number is not very high. Thus a cache is less effective as it doesn’t save that much time.

Efficiency: The worst case running time and the expected case, in terms of:

- **n** - the number of elements in the array
- **L** - the length of the numbers
- **m** - the integer value to calculate the sum of prime factors

- Worst Case: $O(n \log n \cdot (L + m \cdot \log (\log \sqrt{m})))$
- Expected Case: $O(n \log n \cdot L)$

Questions?