CSCI 3501 Sorting Competition

Group 4 Analysis Presentation

Fall 2023, CSCI 3501

Present by Group 4
Dongting Cai & Simon H-M

Results

Final Round 1	Final Round 2	Final Round 3
23 ms	6 ms	56 ms
23 ms	**6 ms**	**64 ms**
23 ms	5 ms	72 ms

Final Result

We took place **5**

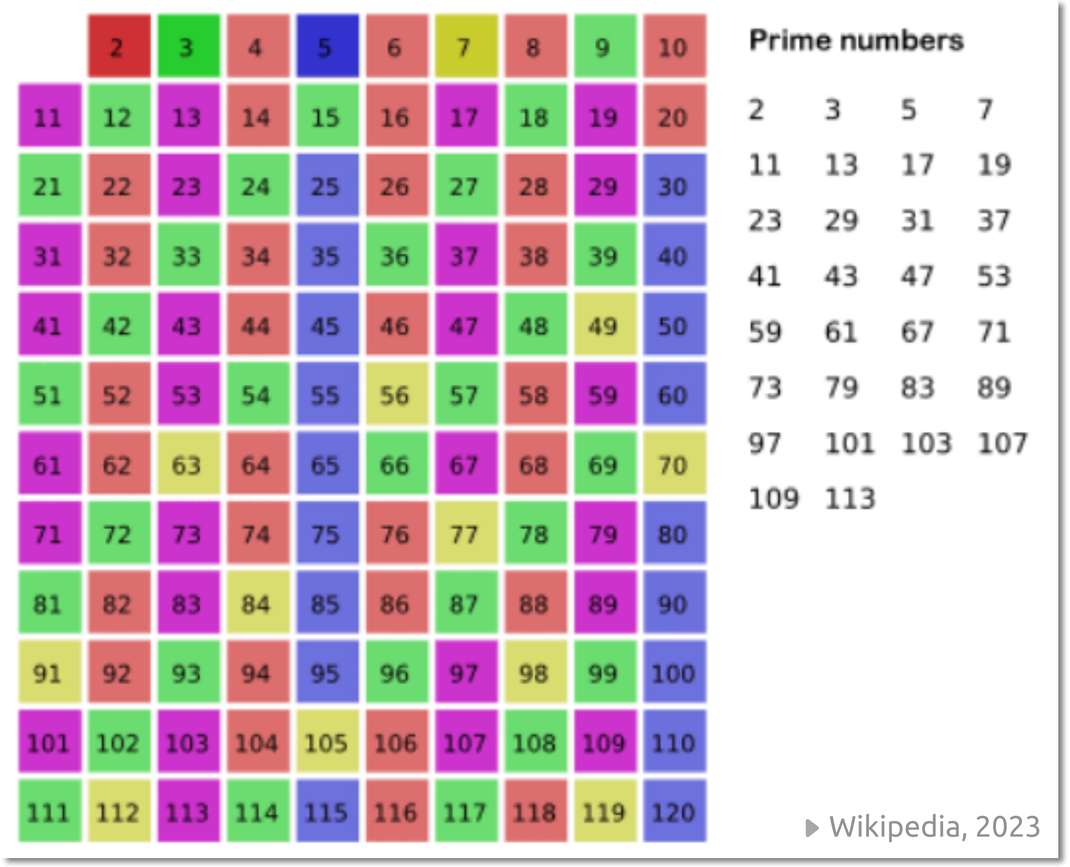
The sum of the places was **12**

The sum of the medians is **93.0**

▸ Github, 2023

Precompute Primes Method

- **Sieve of Eratosthenes algorithm:**
  The multiples of prime numbers are not prime themselves.
- **The algorithm simply eliminates them** from evaluation, which is **more efficient than an algorithm** that would test each integer individually.



Wikipedia, 2023

Getsum Primes Factors Method

- **Initializes a sum variable** to accumulate the prime factors.
- Iterates through a list of **precomputed** prime numbers.
- **Adds a prime to sum** if it's a factor of the input number n.
- Continues the process **until the prime number exceeds n**.
- **Utilizes a cache to store** and retrieve previously computed sums.
- The method ensures that **all prime factors are considered**.
- **Optimizes performance** by reducing redundant calculations.

```java
public static int getSumPrimeFactors(int number, List<Integer> primes,
HashMap<Integer, Integer> cache) {
    // If the sum of prime factors for the given number is already computed and
cached, return it
    if (cache.containsKey(number)) {
        return cache.get(number);
    }

    int sum = 0;  // Initialize the sum of prime factors
    int originalNumber = number;  // Store the original number to cache the result
later

    // Iterate through the precomputed prime numbers to find the prime factors of
the given number
    for (int prime : primes) {
        // If the current prime number is greater than the given number, break out
of the loop
        if (prime > number) {
            break;
        }

        // If the given number is divisible by the current prime number, add it to
the sum
        if (number % prime == 0) {
            sum += prime;

            // Divide the given number by the current prime number as long as it
remains divisible
            while (number % prime == 0) {
                number /= prime;
            }
        }
    }

    // If after the above process, the given number is greater than 1, it is a
prime number and added to the sum
    if (number > 1) {
        sum += number;
    }

    // Cache the computed sum for the given number
    cache.put(originalNumber, sum);

    // Return the computed sum
    return sum;
}
```

Correctness Issue

- ⚠️ **Correctness Issue:**

  Sorted Correctly: Input numbers were **equal** length - *e.g.* 01 and 11

  Sorted Incorrectly: Input numbers were of **different** length - *e.g.* 1 and 11

- **This issue is fixed by** altering the sort method to rewrite all numbers before evaluating them

src > Group4.java > Group4

```java
97      // Step 1: Initialize the map (buckets) for bucketing numbers based on their sum of prime factors
98      HashMap<Integer, List<Integer>> buckets = new HashMap<>();
99
100     // Step 2: Populate the buckets
101     for (String numberStr : data) {
102         int number = Integer.parseInt(numberStr);
103
104         // Compute the sum of prime factors for the current number
105         int sumPrimeFactors = getSumPrimeFactors(number, primes, cache);
106
107         // Add the number to the appropriate bucket in the 'buckets' map
108         buckets.computeIfAbsent(sumPrimeFactors, k -> new ArrayList<>()).add(number);
109     }
110
111     // Step 3: Sort each bucket in descending order based on actual integer values
112     for (List<Integer> bucket : buckets.values()) {
113         Collections.sort(bucket, Comparator.reverseOrder());
114     }
115
116     // Step 4: Merge sorted buckets into a single list
117     // S4-1. get the keys (sum of prime factors) and sort them in ascending order
118     Integer[] keysSorted = buckets.keySet().toArray(new Integer[0]);
119     Arrays.sort(keysSorted);
120
121     // S4-2. Initialize the array to store the final sorted data
122     String[] sortedData = new String[data.length];
123     int index = 0;
124
125     // S4-3. Populate the sortedData array by merging the sorted buckets
126     for (Integer key : keysSorted) {
127         for (Integer nu  int index - Group42.sort(String[])
128             sortedData[index++] = String.valueOf(number);
129         }
130     }
```

```java
98      // Step 1: Initialize the map (buckets) for bucketing numbers based on their sum of prime factors
99+     HashMap<Integer, List<String>> buckets = new HashMap<>();
100
101     // Step 2: Populate the buckets
102     for (String numberStr : data) {
103         int number = Integer.parseInt(numberStr);
104
105         // Compute the sum of prime factors for the current number
106         int sumPrimeFactors = getSumPrimeFactors(number, primes, cache);
107
108         // Add the number to the appropriate bucket in the 'buckets' map
109+         buckets.computeIfAbsent(sumPrimeFactors, k -> new ArrayList<>()).add(numberStr);
110     }
111
112+     // Step 3: Sort each bucket in descending order
113+     for (List<String> bucket : buckets.values()) {
114         Collections.sort(bucket, Comparator.reverseOrder());
115     }
116
117     // Step 4: Merge sorted buckets into a single list
118     // S4-1. get the keys (sum of prime factors) and sort them in ascending order
119     Integer[] keysSorted = buckets.keySet().toArray(new Integer[0]);
120     Arrays.sort(keysSorted);
121
122     // S4-2. Initialize the array to store the final sorted data
123     String[] sortedData = new String[data.length];
124     int index = 0;
125
126     // S4-3. Populate the sortedData array by merging the sorted buckets
127     for (Integer key : keysSorted) {
128+         for (String numberStr : buckets.get(key)) {
129+             sortedData[index++] = numberStr;
130         }
131     }
```

Algorithm Description & Complexity

- **Worst-case Running Time**: $O(n^2)$ ***[Sorting of buckets]***
- **Expected-case Running Time**: $O(n) \rightarrow (n+k)$
- **Data Storage**: Arrays and HashMaps for buckets
- **Additional Structures**: Cache for sum of prime factors

Bucket Sort

$\Omega(n+k)$

$\theta(n+k)$

$O(n^2)$

```java
// Bucket Sort Implementation
public static String[] sort(String[] data) {
    // Find the maximum number in the data
    int maxNumberInData =
Arrays.stream(data).mapToInt(Integer::parseInt).max().getAsInt();
    // Calculate the square root of the maximum number, which will be used to
precompute primes
    int limit = (int) Math.sqrt(maxNumberInData);
    // Precompute prime numbers up to the calculated limit
    List<Integer> primes = precomputePrimes(limit);
    // Initialize a cache to store the sum of prime factors for numbers to speed
up computations
    HashMap<Integer, Integer> cache = new HashMap<>();
    // ... [Additional code]
}
```


▸ Bucket Sort Implementation

Development Insights & Acknowledgments

- **Caching Mechanism:** A Common Optimization Technique
- **Acknowledgments:** ChatGPT (GPT–4 Data Analysis mode), Wikipedia (Sieve of Eratosthenes)

Other Online Blogs

Wikipedia Page

Sieve of Eratosthenes

59 languages

Article Talk Read Edit View history Tools

From Wikipedia, the free encyclopedia

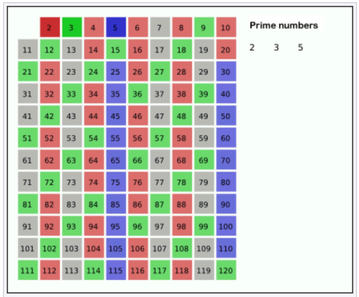
For the sculpture, see The Sieve of Eratosthenes (sculpture).

In mathematics, the **sieve of Eratosthenes** is an ancient algorithm for finding all prime numbers up to any given limit.

It does so by iteratively marking as composite (i.e., not prime) the multiples of each prime, starting with the first prime number, 2. The multiples of a given prime are generated as a sequence of numbers starting from that prime, with constant difference between them that is equal to that prime.[1] This is the sieve's key distinction from using trial division to sequentially test each candidate number for divisibility by each prime.[2] Once all the multiples of each discovered prime have been marked as composites, the remaining unmarked numbers are primes.

The earliest known reference to the sieve (Ancient Greek: κόσκινον Ἐρατοσθένους, *kóskinon Eratosthénous*) is in Nicomachus of Gerasa's *Introduction to Arithmetic*,[3] an early 2nd cent. CE book which attributes it to Eratosthenes of Cyrene, a 3rd cent. BCE Greek mathematician, though describing the sieving by odd numbers instead of by primes.[4]

One of a number of prime number sieves, it is one of the most efficient ways to find all of the smaller primes. It may be used to find primes in arithmetic progressions.[5]



Sieve of Eratosthenes: algorithm steps for primes below 121 (including optimization of starting from prime's square).

Overview [edit]

A prime number is a natural number that has exactly two distinct natural number divisors: the number 1 and itself.

To find all the prime numbers less than or equal to a given integer n by Eratosthenes' method:

1. Create a list of consecutive integers from 2 through n: $(2, 3, 4, \ldots, n)$.
2. Initially, let p equal 2, the smallest prime number.
3. Enumerate the multiples of p by counting in increments of p from $2p$ to n, and mark them in the list (these will be $2p$, $3p$, $4p$, ...; the p itself should not be marked).

Sift the Two's and Sift the Three's:
The Sieve of Eratosthenes.
When the multiples sublime,
The numbers that remain are Prime.

Anonymous[6]

Advanced Data Analysis

Group4.java
File

Hello! I am doing a sorting competition for one of my computer science classes which uses Java. And I finished the code. Before I submit it, Could you help check if my code ("Group4.java") meets the requirements that the professor required in the following description? Thanks!

Summary:

1. **Sorting Logic**: The sorting logic appears to correctly implement the required criteria.
2. **Setup for Sorting**: The setup for sorting, including the method signature, return type, data reading, cloning, and writing processes, aligns with the requirements.
3. **Global Variables**: There are no global variables, which is consistent with the requirements.
4. **Memory Allocation**: Memory allocation during the sorting process seems to be restricted to the duration of one round of sorting.
5. **Multithreading**: The code does not employ multithreading in the sorting logic.
6. **Output**: The program only prints the total time in milliseconds, as required.

Based on this review, it seems that the "Group4.java" code aligns well with the professor's requirements. However, before submitting, it would be advisable to test the code with various datasets to ensure its correctness and performance.

Correctness Analysis (via ChatGPT GPT-4 Data Analysis Mode)

Q & A Session

Thanks for your Listening!

Fall 2023, CSCI 3501
Sorting Competition Group 4

Present by Dongting Cai & Simon H-M