

Group 4 Sorting Competition

RJ Holman, Kyle Foss


Introduction

Our sorting algorithm got 10th place out of the 19 other algorithms in the competition.

The median sorting time for the first set of data was 9475.0

The median sorting time for the second set of data was 3205.0

Our code sorts correctly, when using the diff command and the final results from the github page do not include a time penalty, and the correction group did not find any issue with code correctness.



Our Algorithm

The bulk of the changes made are in the comparator class and involve finding the primes and their products.

The sorting method used is the java predefined Timsort method of the Array class.

No external storage was utilized for the algorithm.



Our Algorithm (in detail)

- 2 returnable numbers are defined, as 0, for evaluation as prime. The case of $n==1$ is checked otherwise we check if the number is even (using $\%2$).
- If the number is even, one of the primes is 2, and the bound is decreased by dividing the input number by 2, until it is no longer even. This is then used as the input for the next portion of the algorithm to find the second prime. This next portion also covers the case for odd input longs.

```
while (n%2 == 0){  
    if(p==0){  
        p = 2;  
    }  
    n = n/2;  
    counter++;  
}  
//if it's a perfect square, it will always be 2  
if(n1 == Math.pow(2, counter)){  
    return 2;  
}
```



Our Algorithm (cont.)

For the case of odd numbers (and finding the second prime number of even numbers). The input number, n , is checked for divisibility for every odd number following 3, until a match is found. This is done twice if “ n ” is initially odd. While this is all happening, n is being updated to be the quotient of n and the current odd number that is divisible by n .

```
For (int i = 3; i <= (long) Math.sqrt(n)+1;
i = i+2){
    if(n%i==0){
        //if p1 isn't set, set it
        if(p==0){
            p = i;
        } else {
            p2 = i;
            //can break, both factors set
            Break;
        }
    }
    while(n%i == 0) {
        n = n/i;
    }
}
```



Our Algorithm (cont.)

What if, there are still missing factors? Then this case applies

- Case 1: p2 is not set; in this case “n” is a prime number, because as explained earlier, the “n” input number is updated by dividing in the for loop, until no divisors exist, leaving n as a prime number, the second number for the product.

Finally the product of primes is calculated, and the case in which the result of the factors is zero, the input number “n” is returned.

```
//if p2 was never set, it was the product of  
2 primes so it will be set here
```

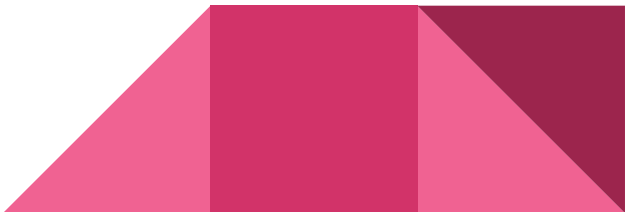
```
    if(p2 == 0){  
        p2 = n;  
    }
```

```
    long result = p2*p;
```

```
    //if one of the factors was zero it  
    was a prime number, so return it
```

```
    if(result == 0){  
        return n;  
    }
```

```
    //map.put(n, result);  
    return result;  
}
```




Efficiency

Tim Sort's worst case: $O(n \cdot \log(n))$, this is also the average case for Timsort.

The comparator method has a worst case of $O(\sqrt{n}/2)$, this however would only occur if n is a product of primes. This is our worst case because the for loop that checks for odd primes, skips even numbers until \sqrt{n} , or until a prime is discovered. Every other loop uses division which decrements the loop bounds significantly more than counting odds.

The practical worst case would be a situation in which all numbers are very large and prime or a product of primes.



Previous Attempts

- The first attempt was to calculate and store all primes up to \sqrt{n} , this could be up to around 13 digits. The algorithm that was implemented was the Sieve of Eratosthenes, however, the efficiency of this algorithm was $O(n \log(n))$. The runtime was slower than that of the original group0 code, just for finding the primes, so alternatives had to be implemented
- The next attempt was to store 1000 prime numbers in a global array. While this did speed up the process, and correctly sort, it was only by a marginal amount (2-5%).
- Another sorting algorithm was also tested, quicksort, this however was slower than the built in Timsort method, so it was abandoned.
- We finally decided that altering the trial division, but still maintaining the basic premise would be our solution. By avoiding stepping through every number and also checking for even primes, (setting $p_1 = 2$), the time was drastically improved.



What could ideally be implemented

In addition to the improvements made to the trial division code. We believe it could be improved more by adding an alternate sorting method to improve the runtime, and perhaps sorting for multiple cases for the different sets of data.

We looked into Pollards rho algorithm, for integer factorization. This algorithm has an efficiency of $\sqrt{\text{smallest-prime}}$ which would drastically reduce, or possibly eliminate, the chance of getting $O(n \log(n))$.

