Project 0 - Warming Up

---

1. **Write a program that reads n (say n=100) integers from keyboard and outputs the integers in ascending order with 10 integers in each line. You can use any sorting algorithms you learned in CSI 2300.**

a. **A description of the components of your solution and their relationships. In the case that a component is a method, describe the algorithm that the method implements. You can use diagrams together with narration to describe your design.**

The solution contains one class called IntegerSorter with one main method that manages the flow of the program using a sequential algorithm. The algorithm starts with the creation of a Scanner object for reading the input from the keyboard. The algorithm reads the number of integers (n), creates an integer array of size n, and then uses a for loop to input n integers into the array. The array is then sorted using the Arrays.sort() method that uses an optimized dual-pivot quicksort. A for loop then goes through the sorted array to print the integer values followed by a space. The use of modular arithmetic helps the loop to print a newline character when the position (i + 1) is divisible by 10.

b. **A justification of your solution. For instance, explain why you have chosen the solution among a few alternatives.**

The reason behind choosing this solution is its simplicity and ability to follow the best practices of the Java programming language. The next best option was to write a custom sorting algorithm such as bubble sort, selection sort, or insertion sort. However, the choice of Arrays.sort() came because it has a time complexity of O(n log n), which is much faster than the $O(n^2)$ time complexity of basic sorting algorithms. The choice of Scanner over BufferedReader came because of its convenience and ability to automatically parse the next token according to its type. An array was preferred over ArrayList because its size is known in advance and is memory efficient. The choice of the format solution based on modular arithmetic (i + 1) % 10 == 0 came because it is the cleanest solution to the problem and does not require the use of a counter or nested loops. The choice of a single class design came because it is the best way to ensure that the code is not over-engineered and requires only the necessary code.

---

2. **Write a method that displays on screen all prime numbers that are small than or equal to 100000. Make your program as fast as possible. Hint: This is a fast algorithm: You start with a sequence of numbers from 2 to 100000. You then output the smallest number in the sequence and remove all of its multiples from the sequence. You repeat the process until the sequence contains no numbers.**

a. **A description of the components of your solution and their relationships. In the case that a component is a method, describe the algorithm that the method implements. You can use diagrams together with narration to describe your design.**

Data Flow Diagram:

Input: limit = 100000

↓

Create boolean array [0..100000]

↓

Initialize array [2..100000] = true

↓

Sieve: Mark multiples of primes as false

↓

Output: Print indices where array = true

The solution consists of a PrimeGenerator class with two methods: main and displayPrimes. The main method provides the entry point and calls displayPrimes with the limit 100000. displayPrimes implements the Sieve of Eratosthenes algorithm using a boolean array isPrime to track which numbers are prime. The algorithm has three phases of work: initialization-clear all numbers from 2 to limit to potentially be prime, sieving-for numbers from 2 to sqrt(limit), mark all multiples of a current prime as composite, starting at prime squared, output-all numbers that remain cleared are prime. The boolean array is the central data structure, array index corresponds to number, and the boolean value indicates whether it is or is not prime.

b. **A justification of your solution. For instance, explain why you have chosen the solution among a few alternatives.**

The Sieve of Eratosthenes was selected because it provides the best possible efficiency in determining primes up to a certain limit. As it has a time complexity of O(n log log n) in comparison to O(n^1.5 / log n) for trial division, it directly picks primes out of evens and odd numbers. It makes use of a boolean array that has a minimal memory cost and takes a constant time to access. It has a number of optimization strategies that include starting marking at i-squared to avoid redundant computations, stopping the loop at the square root of the limit

since larger numbers would have been marked already, and not using dynamic data types that would introduce concepts of garbage collection.