College of Engineering, Software Department
SE 2221 Algorithms

Laboratory 1
Implementing Pseudocode

Name: Rafael III Prudente Year: 2
Name: Paul Vincent Lerado Year: 2

General Instructions: For every item below, please submit one zip file named "<surname1>_<surname2>_<item number>.zip". For example, if Kaedahara and Sangonomiya were submitting for section number 3, they would name their zip file "Kaedahara_Sangonomiya_3.zip". Please also submit a filled out version of this laboratory named "Kaedahara_Sangonomiya_Lab.doc" or with any file extension you have .

Objectives:

- To be able to understand pseudocode and implement it in a language of choice.
- To be able to compare some algorithms that essentially do the same task but in different ways.

Section 1.
Earlier this week (1/24), we learned about the counting sort algorithm. In this section, you shall be implementing the counting sort algorithm yourselves in any language of your choosing. Shown below is its pseudocode:

```
CountingSort(A)
  // A[]-- Initial Array to Sort
  for i = 0 to k do
    c[i] = 0

  // Storing Count of each element
  for j = 0 to n do
    c[A[j]] = c[A[j]] + 1

  // Change C[i] such that it contains actual
  // position of these elements in output array
  for i = 1 to k do
    c[i] = c[i] + c[i-1]

  // Build Output array from C[i]
  for j = n-1 downto 0 do
```

```
    B[ c[A[j]]-1 ] = A[j]
    c[A[j]] = c[A[j]] - 1
end func
```

Make sure to add in some test cases that check whether the algorithm actually works. I **highly** recommend you do not check the internet for the code as this will be good practice. You can, however, look at references and docs.

Questions.

1. Was implementing the code difficult? If so, which parts were difficult and why? If it was not difficult, did any of your previous code knowledge help?

   Yes, even though we did tackle this during our lecture session, implementing wise during our laboratory session still gave us a hard time because there are parts that were not in the pseudo-code. We did have a hard time making the initial arrays because there are javascript syntaxes to apply like the "math.max' and "fill.' But we were able to come up with an idea on how to settle it.

Section 2.

In this section, your group shall be implementing two algorithms that will have the same output. Similar to section 1, you can use any programming language for the implementation as long as you use the same language for both algorithms. The following are their pseudocodes:

```
algorithm 1 (Sieve of Erasthothenes)
    input: an integer n > 1.
    output: all prime numbers from 2 through n.

    let A be an array of Boolean values, indexed by integers 2 to n,
    initially all set to true.
    for i = 2, 3, 4, ..., not exceeding sqrt(n) do
        if A[i] is true
            for j = $i^2$, $i^2 + i$, $i^2 + 2i$, $i^2 + 3i$, ..., not exceeding n do
                set A[j] := false

    return all i such that A[i] is true.
```

```
algorithm 2
    input: an integer n > 1.
    output: all prime numbers from 2 through n.

    k = (n - 2) // 2
    integers_list = [True] * (k + 1)
    for i in range(1, k + 1):
        j = i
        while i + j + 2 * i * j <= k:
            integers_list[i + j + 2 * i * j] = False
            j += 1;
    if n > 2:
        print(2, end=' ')
    for i in range(1, k + 1):
        if integers_list[i]:
            print(2 * i + 1, end=' ')
```

Feel free to ask me any questions if you do not understand the pseudocode.

Results:
Additionally, measure (in seconds) the time it takes for your algorithm to process with different values of n. You can easily do so with the following:

```
console.time('Execution Time');

await getPrimes1(1000);

console.timeEnd('Execution Time');
```

n	Algorithm 1	Algorithm 2
1000	0.159ms	0.206ms
100000	7.267ms	7.702ms
500000	31.328ms	19.635ms

Questions:

1. Was implementing the code difficult? If so, which parts were difficult and why? If it was not difficult, did any of your previous code knowledge help?

Yes, especially in algorithm 1 where on the part "set j=false,' we did make it to "j=== false' so the values in the A array became all true. Where if it is 'j=false' only, the value changes when you try to return the A array. Also, we did come up with a hard time returning the output array. For algorithm 2, it was not as hard as algorithm 1 but we did have an error with the odd numbers in the input. The way to answer that was through "Math.floor."

2. Which algorithm was faster on average? What contributed to that algorithm being faster than the other one?
   In terms of the results, we think that this may have changed if we put both algorithms into one file. But for our case, algorithm 1 is faster than algorithm 2 in n = '1000' and '100000.' On the other hand, algorithm 2 is faster in n = 500000. So to conclude, algorithm 2 is faster than algorithm 1 on average.
   Algorithm 2 is faster than algorithm 1 because it removes only odd composite numbers rather than both odd and even composite numbers. This leads to a smaller number of numbers that need to be marked as composite, reducing the overall computational time as we can see in variable 'k' and in the for a loop.

Conclusions.

For this laboratory, we did learn about pseudocode and how to implement it. Pseudocode is a high-level representation of an algorithm, written in a natural language-like syntax, that helps in understanding the logic of the algorithm before actually writing the code. In this laboratory in implementing it, we students need to translate the pseudocode into a programming language of our choice and in our case it is typescript, using the syntax and constructs of that language. So it requires us to have a good understanding of the algorithm, as well as the language being used.

Also, we did translate algorithms that essentially do the same task which is all about getting the prime numbers but in different ways. These are the Sieve of Sundaram and the Sieve of Eratosthenes. We conclude that the sieve of Sundram is much faster than the sieve of Eratosthenes.