Algorithms and Data Structures (ADS) - COMP1819

Develop and optimise solutions in Python with ADS and provide complexity analysis.

Group Name: **18_4**
Team members:

Member	Name	ID	Contribution %
1	Peddada, Venkata Ramachandra Madhav Peddada	001335072	100%
2	Khan, Hamza Ali	001345840	100%
3	Blyznyuk Petrunyak, Vanessa	001320521	85%
4	Ahmed Farah	001313774	100%
5	Ahmed, Mohammed Jahidul	001299114	90%

Contents

1. Create unique solutions!

Student 1:

The Coursework specification instructs us to come up with a code that checks if a number is prime and palindrome at the same time within a given range, if it is both then we print the first three and last three in the range (i.e. Special numbers). From this we get that that the most basic code would be containing three functions 1) That checks primality, 2) That checks if they are palindrome, 3) That combines both the functions and prints the special numbers. I had written a basic code for this which upon testing was taking extremely long time to run for bigger test cases.

1) is_prime : -

In this function we use the mathematical logic where N can be factored into two parts, a and b, so that n = a * b, if n is not a prime number. The product of a and b, a * b, would be more than n if both were greater than the square root of n. Likewise, their product would be less than n if both were smaller than the square root of n. It is therefore not required to look for factors larger than the square root of n when determining primality because at least one of the factors must be smaller than or equal to the square root of n. using this we check if the number is prime or not.

2) THE_ONE_THAT_GENERATES_PALINDROMES : -

There is also a function which generates palindrome which will reduce the numbers in the range by a lot and in turn will make the code much faster. Up to a specific number of digits, indicated by {`digit_limit'}, palindrome numbers are generated by the function {`THE_ONE_THAT_GENERATES_PALINDROMES}. First, it produces palindromes with just one digit (1 through 9). It first computes the numbers that make up the palindrome's first half to create palindromes with more than one digit. After that, it mirrors this half to create the entire palindrome, changing how it does so depend on whether there is odd or even numbers in total. The first half is replicated in its entirety for even digit counts. To prevent duplication in the middle, the last digit of the first half is not mirrored for odd counts. Without having to examine each number for palindrome characteristics, this method effectively generates palindrome numbers in a succession.

3) THE_ONE_THAT_FINDS_THE_SPECIALS : -

The function effectively looks for numbers in a user-defined range {[m, n]} that are prime and palindromes. To do this, it first determines the maximum length of a number's digit inside the range, establishing a limit for the creation of palindromes. After that, it precomputes a list of prime numbers up to {n} squared, which improves the effectiveness of more prime checks. Up to the designated digit limit, it generates palindrome numbers by utilizing the `THE_ONE_THAT_GENERATES_PALINDROMES` function. It checks if a given integer is in the range {[m, n]} for each palindrome and uses the precomputed prime list to certify the number's primality. The outcome is a set of numbers that, within the specified range, only satisfy the palindrome and prime conditions once, demonstrating an efficient method for finding such exceptional numbers.

4) TEST_CLASS : -

The purpose of the `TEST_CLASS` function is to assess how well the `THE_ONE_THAT_FINDS_THE_SPECIALS` function performs when looking for prime palindrome numbers in different numerical ranges. It accomplishes this by measuring the duration of the search for each given range and recording start and end times using Python's `time` module. {TEST_CLASS} publishes the full list of these numbers or, if the list is extensive, prints only the first and last three items plus the total

count, depending on how many special numbers are found. The algorithm's performance is shown on several scales by repeating this for several predetermined ranges. The range tested, a sample or full list of the prime palindromes discovered, their total count, and the time required for each are all included in the output.

Student 1's Results:

	Input	Output	Running Time (in sec)
1	1 to 2000	First three: [2,3, 5] .... Last three: [797, 919, 929] (Total: 19)	0.0 seconds
2	100 to 10000	First three: [101, 131, 151] .... Last three: [797, 919, 929] (Total: 15)	0.0 seconds
3	20000 to 80000	First three: [30103, 30203, 30403] .... Last three: [79397, 79697, 79997] (Total: 48)	0.0010001659393310547 seconds
4	100000 to 2000000	First three: [1003001, 1008001, 1022201] .... Last three: [1993991, 1995991, 1998991] (Total: 190)	0.007578611373901367 seconds
5	2000000 to 9000000	First three: [3001003, 3002003, 3007003] .... Last three: [7985897, 7987897, 7996997] (Total: 327)	0.014361381530761719 seconds
6	10000000 to 100000000	Special Numbers: [] (Total: 0)	0.04705667495727539 seconds
7	100000000 to 400000000	First three: [100030001, 100050001, 100060001] .... Last three: [399737993, 399767993, 399878993] (Total: 2704)	0.3815629482269287 seconds
8	1100000000 to 15000000000	First three: [10000500001, 10000900001, 10001610001] .... Last three: [14998289941, 14998589941, 14998689941] (Total: 5474)	5.342708110809326 seconds
9	15000000000 to 100000000000	First three: [15001010051, 15002120051, 15002320051] .... Last three: [99998189999, 99998989999, 99999199999] (Total: 36568)	63.34208345413208 seconds
10	1 to 1000000000000	First three: [2,3, 5] .... Last three: [99998189999, 99998989999, 99999199999] (Total: 47994)	73.77152490615845 seconds

Student 2:

Understanding the problem: -
The task according to the CW specification is to write a Python program that identifies, counts and displays special numbers within a specified range. Where according to CW specification special numbers are those numbers which are both prime (divisible by only 1 and themselves) and palindrome (read the same from forwards and backwards). The program should accept two positive integers m and n (where m < n) and then returns the number of special numbers present between m and n, along with displaying these special numbers if they are less than or equal to 6. Otherwise it should display the first three smallest and the last three largest special numbers present between m and n. And we had to make sure that the program does not take more than 1 hour of runtime for large values such as trillion.

Approach for solving the problem: -

1. Defining suitable functions: I have defined three functions in my code which are: getting_prime_numbers(num), getting_palindrome_numbers(num) and getting_special_numbers(limit).
2. Input Managing: to get the input values for m and n from the user.
3. Getting Special Numbers: to get the special numbers inside the specified range by using the defined functions.
4. Displaying Output: if the number of special numbers is less than or equal to 6, print the count and show it, if not then display the first three smallest and final three largest special numbers between m and n.
5. Measure the runtime: record the time taken to produce or to get the results.

Short description of the code: -
My code is mainly made up of three functions which are - getting_prime_numbers(num), getting_palindrome_numbers(num) and getting_special_numbers(limit). The program takes two positive user input values name m and n (where m is smaller than n) and then the getting_prime_numbers(num) function finds the prime numbers present between the specified range, then the getting_palindrome_numbers(num) function converts the numbers into string and checks whether the numbers are spell same from forwards and backwards, then the third and last function getting_special_numbers(limit) finds or filters the numbers which are both prime and palindrome within the specified range and if six or less than six are six special numbers are found its prints all of them else it displays the first three and the last three special numbers along with the time taken to get the results.

Brief explanation of the working of my code:

1. getting_prime_numbers(num) function: this function checks whether the number is prime or not. It first eliminates the numbers which are smaller than 2 or divisible by 2 or 3 or 5 or 7 or 10 to make my program more efficient. And then it iterates from 2 to the square root of the number (using the "sqrt function" of "math library") and returns true if the number is exactly divisible by 1 and itself, else it returns false.

2. getting_palindrome_numbers(num) function: this function checks whether a function is palindrome or not. It first converts the number into string and then compare it with its reverse and returns true if they are equal, else it returns false.

3. getting_special_numbers(limit) function: now this is the core function of my code it generates palindromic numbers within the given limit for both even and odd lengths and then filters them based on whether they are prime or not.)" i.e. this function iterates through a range from 1 to 10^((limit + 1) // 2) inclusive to generate palindromic numbers within a given limit. It creates palindromes with odd and even lengths within this range. It first determines whether each potential palindrome falls into the input range (m to n), then uses the "getting_palindrome_numbers(num)" function to determine if it is a palindrome and last, the "getting_prime_numbers(num)" function to determine if it is prime. Then the palindrome is added to a set of special numbers if all the

requirements are satisfied.
Its working is broken down below.

- Nested loop: the function has a nested loop structure which iterates over potential palindromes considering both odd and even length of numbers to make ensure that all possible palindromic numbers are evaluated.
- Palindromic number generation: each time the nested loop iterates, a base number and its reverse are joined to produce a palindromic number. It is assured that the output of this technique will read the same in both directions.
- Filtering criteria: as now we know the palindromic numbers, so those numbers are filtered which are both prime and palindromic.
- Now they are filtered in a set and then finally returned as a list of special numbers.

4. Input Handling: to get the input values for m and n from the user.

5. Get Special Numbers: now, call the get_special_numbers(limit) function to get the special numbers between the specified range.

6. Measure Runtime: at last, record the time taken to produce or to get the result.

Student 2's Results:

	Input	Output	Running Time (in sec)
1	m = 1 n = 2_000	First three and last three special numbers: [2, 3, 5, 797, 919, 929]	0.0005
2	m = 100 n = 10_000	First three and last three special numbers: [101, 131, 151, 797, 919, 929]	0.0022
3	m = 20_000 n = 80_000	First three and last three special numbers: [30103, 30203, 30403, 79397, 79697, 79997]	0.0031
4	m = 100_000 n = 2_000_000	First three and last three special numbers: [1003001, 1008001, 1022201, 1993991, 1995991, 1998991]	0.0357
5	m = 2_000_000 n = 9_000_000	First three and last three special numbers: [3001003, 3002003, 3007003, 7985897, 7987897, 7996997]	0.0620
6	m = 10_000_000 n = 100_000_000	Special numbers: []	0.0994
7	m = 100_000_000 n = 400_000_000	First three and last three special numbers: [100030001, 100050001, 100060001, 399737993, 399878993, 399878993]	1.5073
8	m = 1_100_000_000 n = 15_000_000_000	First three and last three special numbers: [10000500001, 10000900001, 10001610001, 14998289941, 14998589941, 14998689941]	21.7191
9	m = 15_000_000_000 n = 100_000_000_000	First three and last three special numbers: [15001010051, 15002120051, 15002320051, 99998189999, 99998989999, 99999199999]	300.7609

10	m = 1 n = 1_000_000_000_000	First three and last three special numbers: [2, 3, 5, 99998189999, 99998989999, 99999199999]	333.1485

Student 3:

The code shown below defines three main functions, each with a different job. One is to search for palindromes, the second one is to search for prime numbers, and the third one will search for numbers that satisfy the other two functions, the special numbers. It will search within a range that can be provided by the user, and it will iterate through each number and test for both properties, so when it satisfies both it will be added to a list. If the special numbers found are 5 or less, it will print all of them but if it is larger, it will only print the three smallest and largest special numbers. Additionally, it will also measure the runtime of the process and print it after the special numbers.

First function: is_palindrome

By turning the given number into a string and then comparing it to its reverse, this function will return “true” when the numbers are the same, indicating that it is a palindrome, otherwise it will return “false”.

Second function: is_prime

This function will search for the prime numbers and there are 3 main steps.

1. If the number is less than 2 it will return false as these numbers are not prime, however, for numbers 2, and 3 it will return true directly because these numbers are prime.
2. For other numbers it will divide them by 2 and 3 separately because if they are divisible by this number, it is not prime. This also helps to skip unnecessary iterations and will improve the efficiency of the code.
3. After checking these cases it will use another method. By using “max_divisor”, it iterates from 5 to the square root of the number in steps of 6 checking for divisibility of the numbers in form 6x + 1, because prime numbers greater than 3 are represented in this form. This method reduces the number of divisors to be checked optimizing the test.

If the numbers are divisible by any of these it will return “false” as they are not prime.

Third function: find_special_numbers

This function initializes with an empty list that will store the special numbers when found, it iterates each number within the range and searches for both prime and palindromes using the “is_prime” and “is_palindrome” functions. When a number that satisfies both functions is found is added to the “special_numbers” list.

Student 3's Results:

	Input	Output	Running Time (in minutes)
1	1 - 2_000	Special numbers: [2, 3, 5] [797, 919, 929]	0.0 minutes
2	100 - 10_000	Special numbers: [101, 131, 151] [797, 919, 929]	0.0 minutes
3	20_000 - 80_000	Special numbers: [30103, 30203, 30403] [79397, 79697, 79997]	0.0 minutes
4	100_000 - 2_000_000	Special numbers: [1003001, 1008001, 1022201] [1993991, 1995991, 1998991]	0.01 minutes
5	2_000_000 - 9_000_000	Special numbers: [3001003, 3002003, 3007003] [7985897, 7987897, 7996997]	0.03 minutes
6	10_000_000 - 100_000_000	Special numbers: [	0.4 minutes
7	100_000_000 - 400_000_000	Special numbers: [100030001, 100050001, 100060001] [399737993, 399878993, 399878993]	1.6 minutes
8	1_100_000_000 - 15_000_000_000	Special numbers: [10000500001, 10000900001, 10001610001] [14998289941, 14998589941, 14998689941]	3.9 minutes
9	15_000_000_000 - 100_000_000_000	Special numbers: [15001010051, 15002120051, 15002320051] [99998189999, 99998989999, 99999199999]	10.3 minutes
10	1 - 1_000_000_000_000	Special numbers: [2, 3, 5] [99998189999, 99998989999, 99999199999]	33.7 minutes

Student 4:

CW Specification

In this CW specs I was told to use python program that can output time taken, list of unique numbers and total unique number. From reading the specifications i knew that i have to make sure that program can handle any range that it is given. Also, I had to input two positive numbers, m and n (where m is smaller than n), and the program should tell you how many unique numbers are between them (inclusively). The program should not only count these unique numbers but also show them to you. If there are less than 6 special numbers, it should display all of them. Otherwise, it should show you the first three smallest special numbers and the last three biggest special numbers. Special numbers are those that are both prime (only divisible by 1 and themselves) and palindromic (read the same backwards as forward).

The way my code functions:

My code is made up of three functions which is: **find_unique_numbers(Small, Big), display_running_it_back(Small, Big), this_is_my_prime_checkup_system(num)**. Starting with the first function "**find_unique_numbers(Small, Big)"** what it does is that it takes two arguments which is small and big which are integers, and then returns a long list of unique numbers which are within the range of small to big that must go through a prime checker. The second function is "**display_running_it_back(Small, Big)"** what this function does is that it generates all possible palindromic numbers that in the range from small to big and must return it as a set. The last function is "**this_is_my_prime_checkup_system(num)"** what this function does is that it checks if the given Num is a prime number, then it will either return it as true if it is a prime number, and if it isn't a prime number then it is false.

As a whole the code calculates the time taken to be able to complete and execute these operations and it will print out a full list of unique prime numbers found within a selected range, along with the total unique number.

Student 4's Results:

	Input	Output	Running Time (in minutes)
1	1, 2000	This is the full list of unique numbers = [2, 3, 5] [797, 919, 929] This is the full total number of unique number = 20	Time taken: 0.00 minutes
2	100, 10000	This is the full list of unique numbers = [101, 131, 151] [797, 919, 929] This is the full total number of unique number = 15	Time taken: 0.00 minutes
3	20000, 80000	This is the full list of unique numbers = [30103, 30203, 30403] [79397, 79697, 79997] This is the full total number of unique number = 48	Time taken: 0.00 minutes
4	100000, 2000000	This is the full list of unique numbers = [1003001, 1008001, 1022201] [1993991, 1995991, 1998991] This is the full total number of unique number = 190	Time taken: 0.00 minutes
5	2000000, 9000000	This is the full list of unique numbers = [3001003, 3002003, 3007003] [7985897, 7987897, 7996997]	Time taken: 0.00 minutes

		This is the full total number of unique number = 327	
6	10000000, 100000000	This is the full list of unique numbers = [] [] This is the full total number of unique number =	Time taken: 0.00 minutes
7	100000000, 400000000	This is the full list of unique numbers = [100030001, 100050001, 100060001] [399737993, 399767993, 399878993] This is the full total number of unique number = 2704	Time taken: 0.02 minutes
8	1100000000, 15000000000	This is the full list of unique numbers = [10000500001, 10000900001, 10001610001] [14998289941, 14998589941, 14998689941] This is the full total number of unique number = 5474	Time taken: 0.42 minutes
9	15000000000, 100000000000	This is the full list of unique numbers = [15001010051, 15002120051, 15002320051] [99998189999, 99998989999, 99999199999] This is the full total number of unique number = 36568	Time taken: 5.83 minutes
10	1, 1000000000000	This is the full list of unique numbers = [2, 3, 5] [99998189999, 99998989999, 99999199999] This is the full total number of unique number = 47995	Time taken: 6.28 minutes

Student 5:

A Brief Explanation of How I Understood The Problem:
According to the coursework specifications, I was tasked with developing a Python program to assist users in identifying special numbers. At first the end user would have to input two positive numbers, one identified as ‘m’ which is meant to be smaller than the other one which would be ‘n’. Then the programme will figure out the special numbers between them. When we refer to special numbers, we are talking about numbers which are prime (natural numbers bigger than 1 and have exactly two positive dividers 1 and the number itself), and palindromic (read the same forwards/backwards, e.g. 121). The program displays these numbers; if there are fewer than six, it shows them all; otherwise, it presents the first three smallest and the last three largest special numbers. I also was aware of the constraints which included not using more than five hard-coded values, avoiding external data files, and aiming for a maximum runtime of one hour for any test case. Additionally, the program was encouraged to rely solely on Python's built-in libraries to maintain understanding of how everything works.

My Approach to Solving It:
For my code I wanted the approach to be simple as for algorithms to be efficient I’ve learnt you need to make sure your code doesn’t include any unnecessary things which can slow down runtime as the programme will have to compute with all the things you’ve added. The first thing I done was define functions to use. These things would break down the complex task into smaller manageable components. I used two functions which were ‘is_prime(n)’ which checks whether a number given ‘n’ is a prime number or not. The other function was ‘generate_palindromic_primes(start, end)’ which generates a list of palindromic prime numbers within the specified range. Now that’s done I had to have two lines of code to get the inputs of m and n from the user. After that the programme receives the special numbers inside the range given. Finally, the program displayed these special numbers, showing all if fewer than six, and otherwise presenting the first three smallest and last three largest.

A Short Description of My Code:
With my code, it implements an algorithm to identify special numbers with a given range. These are identified using two main functions is_prime(n)', which determines I if numbers prime considering factors such as divisibility by 2 and 3 and more, and 'generate_palindromic_primes(start, end)', which generates palindromic primes within the specified range using a nested loop structure. Within this function, it initializes an empty list called special_numbers to store the palindromic prime numbers. Then, it iterates over numbers from 1 to 10^6 using a for loop. During each step, the code converts the current number to a string and creates palindromic numbers by adding the reverse of the string to itself. It then checks if the generated palindrome is within the range given and if it's a prime. If both conditions are met, the code adds the number to a list. Once all numbers have been processed, the list is sorted and returned. After asking the user for a range of numbers between "m" and "n," the programme sets a timer and determines the palindromic prime numbers that fall inside the range. Following calculation, the timer is stopped, the duration is chosen, and an output string is created. It contains the special numbers separated by commas (if there are six or less of them). The special numbers and the time taken are printed by the programme too.

An Explanation of How The Code Works:

1. I have the time module imported first; this allows for time measurement in the script. It’s part of Pythons built in libraries, so I’m allowed to use it.

2. The ‘is_prime’ function determines if a numbers prime or not. It does a bunch of checks. It checks if the number is less than 2, as primes must be bigger than 1, checks if its 2 or 3 as there prime, checks if the numbers divisible by 2 or 3 excluding those from being prime. Now the basic checks are out of the way we can do more advanced ones like a loop in which two variables, i and w, are initialized to 5 and 2 respectively to set up a loop for prime number checking. It continues until the square of i is greater than the number being checked. Within the loop, it checks if the number is divisible by i; if so, it returns False, indicating it's not prime. It updates the values of i and w for the next iteration. Finally, if the number passes all prime checks, the function returns True, indicating it's prime.

3. The function generate_palindromic_primes(start, end), creates a list of prime numbers that are also palindromes within a given range. First, it starts with an empty list to hold these special numbers. Then, it goes through numbers from 1 to 999999 (1 to 10^6), examining each one. For each number, it turns it into a string to work with it. Next, it constructs two palindromic numbers, one odd and one even, by adding the reverse of the number to itself. For example, 123 becomes 123321 and 12321. It then checks if these palindromic numbers are within the specified range (from 'start' to 'end') and if they are prime using a function called is_prime() – I mentioned this before. If they are within the range and prime, they are added to the list of special numbers. Finally, the function returns the list of special numbers, ensuring they are sorted in ascending order.

4. How Input is Taken in My Code: The user is asked to provide two numbers, 'm' and 'n', to define a range.

5. Recording the run time. The program starts measuring time before doing the main work it needs to do. It then calculates the special numbers (prime palindromes) within the given range. After the calculation, it stops measuring thee time. The elapsed time is found out by the difference between the start and end times. The program prepares an output showing the count of special numbers. If there are six or lower, they are listed in the output. Otherwise, the first three smallest and last three largest special numbers are listed. Finally, the program prints both the special numbers and the time taken for the calculation.

Student 5's Results:

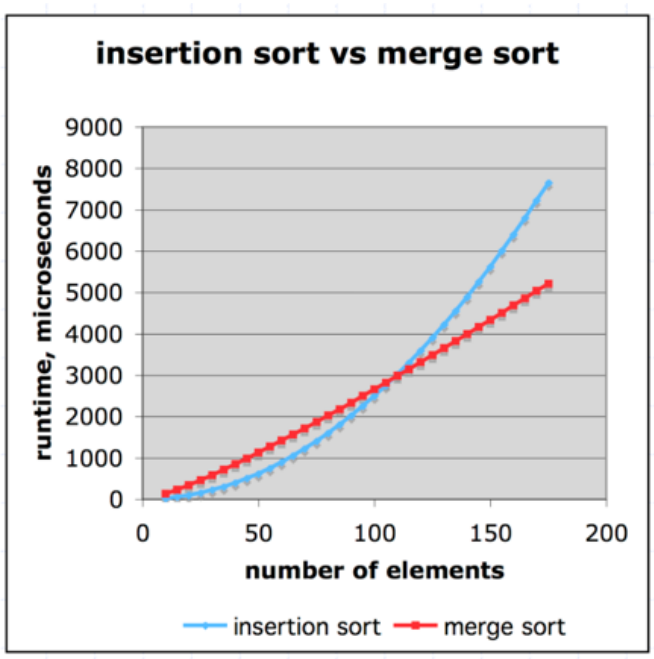
	Input	Output	Running Time (in sec)
1	1 and 2_000	20: 2, 3, 5, 797, 919, 929	0.000224 seconds
2	100 and 10_000	15: 101, 131, 151, 797, 919, 929	0.000217 seconds
3	20_000 and 80_000	48: 30103, 30203, 30403, 79397, 79697, 79997	0.001476 seconds
4	100_000 and 2_000_000	190: 1003001, 1008001, 1022201, 1993991, 1995991, 1998991	0.016825 seconds
5	2_000_000 and 9_000_000	327: 3001003, 3002003, 3007003, 7985897, 7987897, 7996997	0.041572 seconds
6	10_000_000 and 100_000_000	0	0.011914 seconds
7	100_000_000 and 400_000_000	2704: 100030001, 100050001, 100060001, 399737993, 399767993, 399878993	0.917796 seconds
8	1_100_000_000 and 15_000_000_000	5474: 10000500001, 10000900001, 10001610001, 14998289941, 14998589941, 14998689941	14.016485 seconds
9	15_000_000_000 and 100_000_000_000	36568: 15001010051, 15002120051, 15002320051, 99998189999, 99998989999, 99999199999	203 seconds
10	1 and 1_000_000_000_000	47995: 2, 3, 5, 99998189999, 99998989999, 99999199999	236 seconds

2. Test and analyse your solution!

Your test cases:

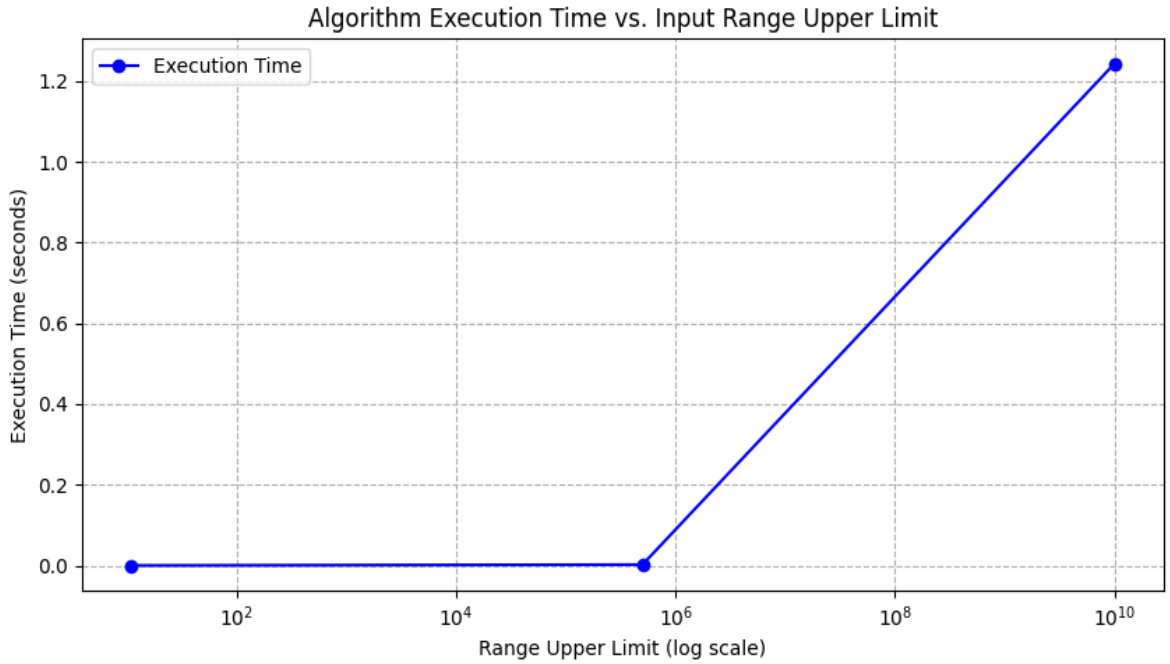
c	Input	Output	Justification	Student X results
1				
2				
3				
4				

Running time graphs



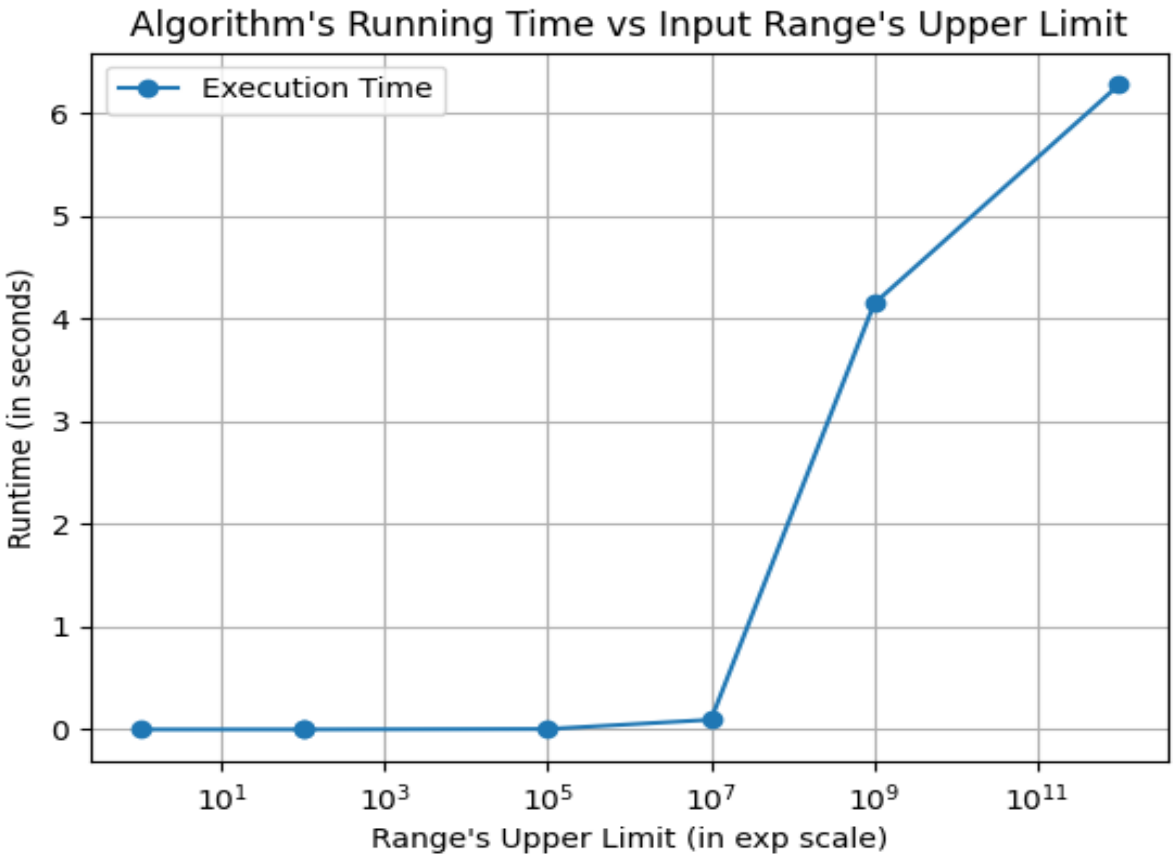
Student 1’s Test Cases and Running Time Graph:

c	Input	Output	Justification
1	(1,11)	[2, 3, 5, 7, 11] (Total: 5) Time taken: 0.0 seconds	This range evaluates how well the algorithm handles edge situations such as the lowest prime (2) and the first two-digit palindrome prime (11) and examines its ability to discover the smallest prime palindromes. It guarantees that the algorithm covers both fundamental functionality and edge situations, handling both very tiny integers and the change from single to double digits.
2	(1,500000)	[2, 3, 5] .... Last three: [97879, 98389, 98689] (Total: 113) Time taken: 0.002074718475341797 seconds	With an expected dense distribution of prime palindromes in the lower section, this scenario tests the algorithm's accuracy and efficiency over a broad range. It's a thorough test to evaluate how effectively the algorithm handles a broad collection of possible palindromes and scales with increasing range, especially in light of the variable density of prime palindromes.
3	(100000000,10000000000)	[100030001, 100050001, 100060001].... Last three: [999676999, 999686999, 999727999] Time taken: 1.2390239238739014 seconds	By aiming for a range with extremely high numbers, the algorithm's scalability and performance bounds are tested. It pushes the limits of computational complexity and assesses the algorithm's optimization for handling big inputs by looking at how well it can produce palindromes and check for primality while working with nine-digit values.



Student 2's Test Cases and Running Time Graph:

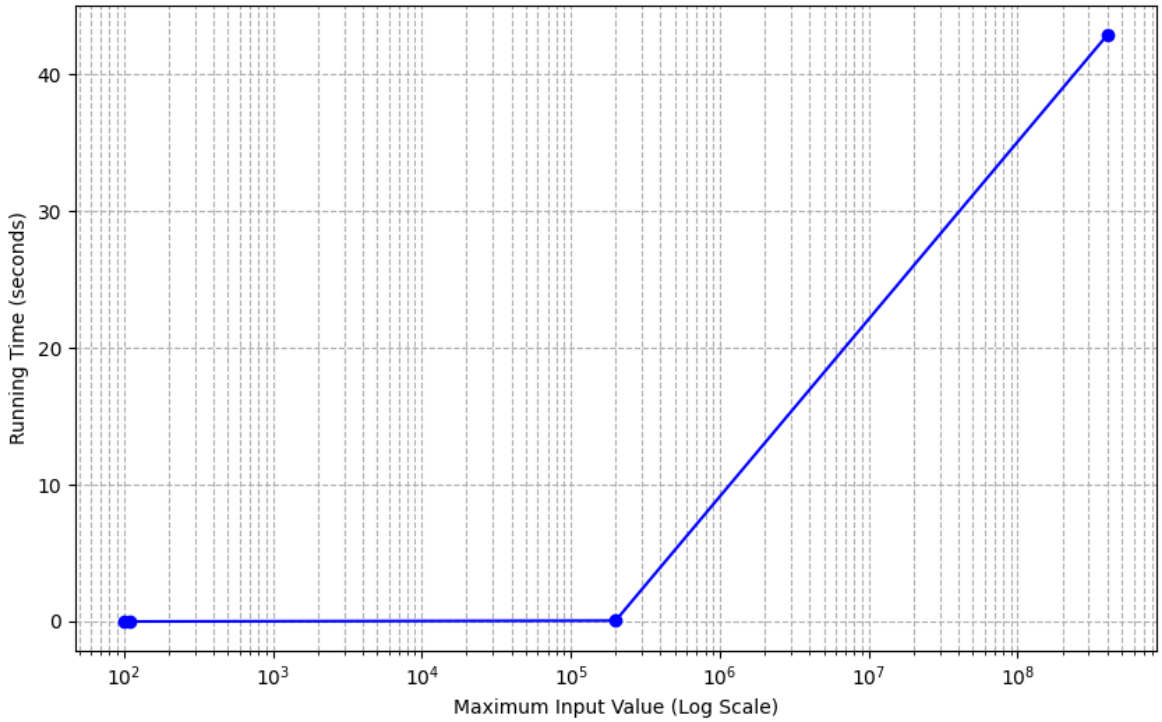
c	Input	Output	Justification
1	m = 1 n = 2	Special numbers: [2] Time taken to produce the result: 0.0001 seconds	This test case examines the smallest possible edge case where special numbers can be present. It verifies that the program handles the edge cases correctly or not.
2	m = 101 n = 101	Special numbers: [101] Time taken to produce the result: 0.0003 seconds	This test case focuses on the range where there exist only one special number. It verifies that the program handles the test cases with ranges having only one special numbers between them correctly or not.
3	m = 50 n = 80	Special numbers: [] Time taken to produce the result: 0.0001 seconds	This test case examines the range where there is no special number present. It verifies that the program handles the test cases where the special number do not exist and outputs the count as 0 correctly or not.
4	m = 10000 n = 100000	First three and last three special numbers: [10301, 10501, 10601, 97879, 98389, 98689] Time taken to produce the result: 0.0045 seconds	This test cases examines the range where more than 6 special numbers exist. It verifies that the program correctly prints the first three and the last three special numbers present within the specified range.
5	m = 1000000 n = 10000000	First three and last three special numbers: [1003001, 1008001, 1022201, 9980899, 9981899, 9989899] Time taken to produce the result: 0.0947 seconds	This test case examines the range spanning from one million to ten million. It verifies that the program handles the test cases with large datasets correctly and efficiently. And by this we get to know that the program effectively processes significant amount of data.
6	m = 1 n = 1000000000	First three and last three special numbers: [2, 3, 5, 999676999, 999686999, 999727999] Time taken to produce the result: 4.1445 seconds	This test case examines the very-very large range spanning from one to one billion. It evaluates how reliable the programme is with massive numbers and having massive range. And by this we can assess the algorithm's optimization for handling big inputs and the robustness and correctness of the program.
7	m = 500000000000 n = 1000000000000	Special numbers: [] Time taken to produce the result: 6.2735 seconds	This test case covers a span from 500 billion to 1 trillion. It assesses how reliable and effective the programme is when handling astronomical numerical values. We confirm that the solution is still efficient and dependable in situations with very big numbers by choosing such a vast range.



My code works well and processes a variety of input sizes in efficient manner. Its runtimes are consistently low and it handles small to moderately sized ranges quickly. Nevertheless, the runtime sharply rises for very large input ranges i.e. in the trillions. In summary my code manages to scale well while maintaining efficiency within the limitations.

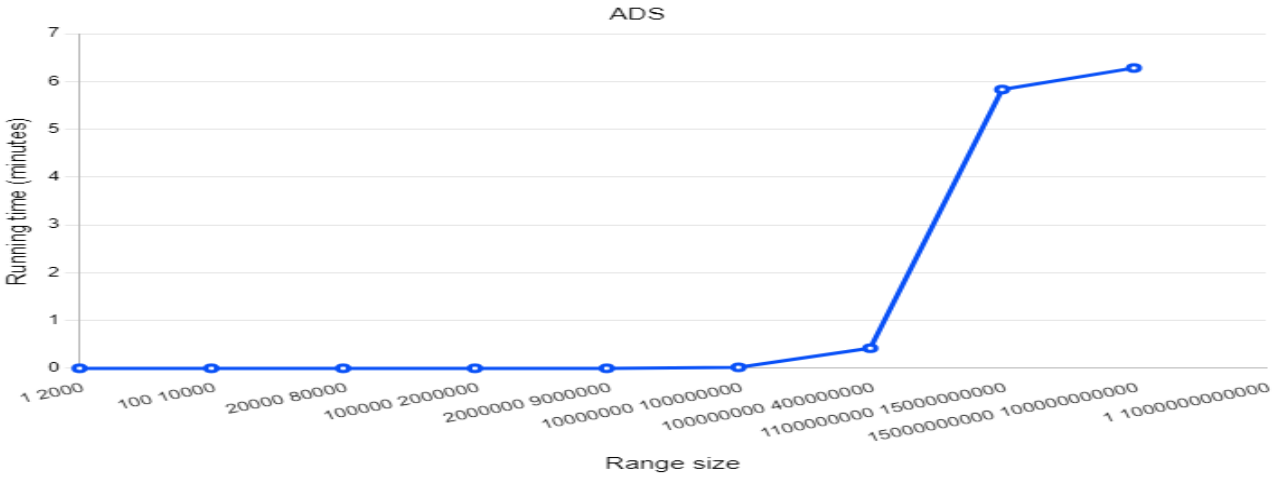
Student 3's Test Cases and Running Time Graph:

c	Input	Output	Justification
1	1, 101	Special numbers: [2, 3, 5] [7, 11, 101] Runtime: 0.0 minutes	This test case covers a small range of numbers, this is to ensure that the code prints the solutions correctly by identifying all special numbers.
2	100, 110	Special numbers: [101] Runtime: 0.0 minutes	This test case examines a range where there is only one special number. It verifies that the program is still function even when there is only one special number
3	100_000, 200_000	Special numbers: [] Runtime: 0.0 minutes	This test case is to show how even when the range is large there are cases with no special numbers at all. The result is just an empty list proving that the program still works.
4	40_000_000, 400_000_000	Special numbers: [100030001, 100050001, 100060001] [399727993, 399767993, 399878993] Runtime: 0.7 minutes	This test case verifies that the program successfully handles scenarios with large ranges.



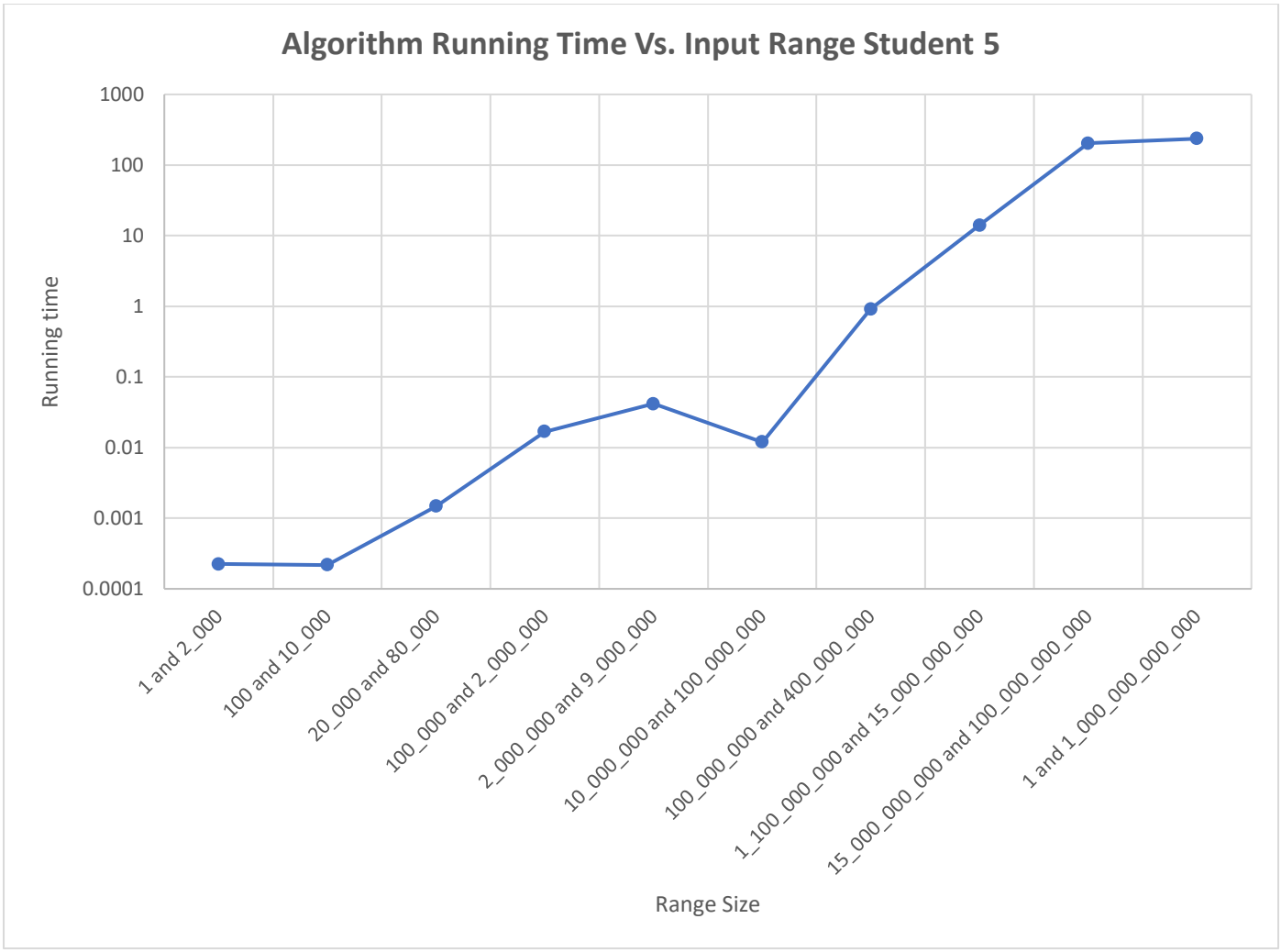
Student 4's Test Cases and Running Time Graph:

c	Input	Output	Justification
1	1, 15	Time taken: 0.00 minutes This is the list of unique numbers = [2, 3, 5] [5, 7, 11] This is the full total number of unique number = 5	In this section code what it does is that it inspects the range of my small number and sees what it can handle by currently inputing these test cases.You can also see if it gives you back the right information such as the list of unique numbers.
2	100, 105	Time taken: 0.00 minutes This is the full list of unique numbers = [101] This is the full total number of unique number = 1	In this part I wanted to see if my code could give me a single unique number when given a test case. This proves that the program works properly and that it will give out one unique number.
3	5, 400000	Time taken: 0.00 minutes This is the full list of unique numbers = [5, 7, 11] [97879, 98389, 98689] This is the full total number of unique number = 111	In this section I used bigger numbers compared to the previous test cases, the reason why I used this big number test case is to see if my code algorithm can handle it and still provide me with the time taken, correct list of unique number and the full total number. Once the output is checked and it is correct then you will know that your code is structured in a way so that it will be able to handle any number input (test cases).
4	43, 73	Time taken: 0.00 minutes This is the full list of unique numbers = [] [] This is the full total number of unique number = 0	In this part I used this test case to inspect and see that when there is no list of unique numbers, if my code would still give out an output. However, this proves that my program works correctly and that when there is no list of unique number it should give nothing as an output.
5	1000000, 1000000000000	Time taken: 6.16 minutes This is the full list of unique numbers = [1003001, 1008001, 1022201] [99998189999, 99998989999, 99999199999] This is the full total number of unique number = 47882	In this section I used a very big test case to see if my programme will be able to handle the range from one million to a trillion. And my programme was able to handle it this shows prove on how strong and independent this code really is. Also, this means that programme will be able to handle multiple large data sets at any range and still be able to complete them correctly.



Student 5's Test Cases and Running Time Graph:

c:	Input:	Output:	Justification:
1	m = 10 n = 20	Special Numbers are: 1: 11 Time taken: 0.000079 seconds	This test will test how the programme can handle a range of small numbers close to each other. It ensures that the programme can identify special numbers where there is a narrow interval which helps validate the functionality of the programme. Its essentially testing if it can handle simple cases as if it can't handle the basic ones there's no point trying the larger ones.
2	m = 2 n = 400	Special Numbers are: 14: 2, 3, 5, 353, 373, 383 Time taken: 0.000185 seconds	Since part of the specification is that if there's more than 6 special numbers it should print the first three smallest and last three biggest, this test case will ensure the correctness of this requirement.
3	m = 1000 n = 1100	Special Numbers are: 0: Time taken: 0.000208 seconds	With this test I can check how well the program handles a range where both numbers end with zero, i.e. multiplies of 10. This helps ensure the program remains accurate whilst also being efficient when dealing with numbers that share a common factor, such as multiples of 10.
4	m = 1 n = 80000	Special Numbers are: 94: 2, 3, 5, 79397, 79697, 79997 Time taken: 0.001430 seconds	To test from inputs spanning from the smallest number possible to a larger number we can do a test which will test the program's scalability and efficiency. This will evaluate how well the program performs when dealing with a significant amount of data, ensuring it remains accurate and gives a response even with large input ranges.
5	m = 17 n = 31	Special Numbers are: 0: Time taken: 0.000061 seconds	The programme does two things, it checks if a numbers prime, but it also must be palindromic. With this test I used a range of only prime numbers to make sure that even though those numbers are prime they cannot be outputted as they are not palindromic (read the same front/back). This tests the accuracy. With my inputs of 17-31 we know the prime numbers between this are 17, 19, 23, 29 and 31 but none of these are palindromic.
6	m = 1000000000 n = 1000000000000	Special Numbers are: 42042: 10000500001, 10000900001, 10001610001, 99998189999, 99998989999, 99999199999 Time taken: 212.189598 seconds	Testing with large data, i.e. billion to trillion range. This test evaluates the program's performance when dealing with an extremely large range. It is easier to evaluate the program's scalability and effectiveness in handling large tasks when such a wide range is tested. It ensures that unusual numbers in this wide range can be recognised and shown by the programme. By going from one magnitude to another we can easily test large data sets instead of using random numbers.


Algorithm Running Time Vs. Input Range Student 5
Running time
1000
100
10
1
0.1
0.01
0.001
0.0001
1 and 2_000
100 and 10_000
20_000 and 80_000
100_000 and 2_000_000
2_000_000 and 9_000_000
10_000_000 and 100_000_000
100_000_000 and 400_000_000
1_100_000_000 and 15_000_000_000
15_000_000_000 and 100_000_000_000
1 and 1_000_000_000_000
Range Size

Complexity analysis

Student 1's Complexity Analysis:

1)Finding Prime Numbers: -Using is_prime(n) to find primes: As it may iterate up to sqrt(n) to discover divisors, especially if no precomputed primes are utilized to optimize the process, this has a worst-case time complexity of O(sqrt(n)).
2)Generating Palindromes :- (digit_limit) / THE_PERSON_ WHO_GENERATES_PALINDROMES): Because it creates palindromes by looping over more than half of the integers required to construct a palindrome, the complexity is O(10^(digit_limit/2)).
3)Finding Special Numbers: - (THE_ONE_THAT_FINDS_THE_SPECIALS (m, n)). In this process, palindromes are generated, and their primality is verified. Presuming that precomputed primes up to n are utilized, the first sieve operation to precompute primes is O (n loglog n), after which palindrome creation and primality checking take center stage. Because precomputed primes provide efficiency, the estimated upper bound for the total procedure may be expressed as O(10^(digit_limit/2) * sqrt(n)).

Student 2's Complexity Analysis:

- Here in my code, the first function – the "getting_prime_numbers(num)" function iterates from 2 to the square root of the number i.e. upto int(sqrt(num))+1. So the time complexity for this operation is **"0(sqrt(n))"** where n is input number.

- The second function – the "(getting_palindrome_numbers(num)" function first converts the number into string and then checks whether it is same from backwards and forwards or not. So the time complexity for this operation will be **"O(log10(n))"** where n is input number.

- Now, coming to the third/last and core or important function of my program i.e., "getting_special_numbers(limit)". This function iterates through a range from 1 to 10^((limit + 1) // 2) inclusive to generate palindromic numbers within a given limit. It creates palindromes with odd and even lengths within this range. It first determines whether each potential palindrome falls into the input range (m to n), then uses the "getting_palindrome_numbers(num)" function to determine if it is a palindrome and last, the "getting_prime_numbers(num)" function to determine if it is prime. Then the palindrome is added to a set of special numbers if all the requirements are satisfied. The function's temporal complexity is dependent on the range (m to n) and the specified limit. The time complexity is about "**O(10^(limit/2))",** since iterating up to 10^((limit + 1) // 2). Due to which the overall/final time complexity of my code is determined by the range of the input value (m to n) and the number of special numbers that are found within the range.

- Hence, for my code in worst case with a big range and many special numbers would result in a time complexity which is about "**O(10(limit/2) * sqrt(n)**)".

Student 3's Complexity Analysis:

The 'is_palindrome' function will convert the numbers into a string and check if its equal to its reverse, compering whether the given number is a palindrome or not. The conversion complexity takes 0(d) time, where d is the number of digits in the number. The comparison of the string to its reverse also takes 0(d) time. Therefore, the complexity of this function is 0(d).

The 'is_prime' function handles some special cases in constant time, then it iterates up to the square root of the number and it will perform a constant-time operations such as modulo calculations, this to determine whether the given number is prime or not. The complexity of the time is 0(sqrt(n)), where n is the number being checked for primality.

The 'find_special_numbers' function is crucial for the solution. This function will iterate through a range of numbers, which can be provided by the user, and it will check each number whether they are palindromes and prime at the same time. Of within the range given all numbers appear to be special numbers, the function will iterate through the entire range. The palindrome check takes 0(d) time, and the primality check takes 0(sqrt(n)) time. Therefore, the overall time complexity of this function is 0((n*sqrt(n))*d), approximately, where n is the size of the range, sqrt(n) is the number of iterations in the is_prime function, and d is the maximum number of digits in the range.

Student 4's Complexity Analysis:

In my code I used the test case example, which is down below, I had to use the inputs to check if my code would end up giving the correct output. Doing this it allowed me to keep on attempting and analysing my code until it would work and release the right output and display it in the same way as the comments on image did.

Examples:

Input (m n)	Output (Total: Special Numbers)	Comments
1 3	2: 2, 3	List of special numbers = 2,3 Total number of special numbers = 2
1 20	5: 2, 3, 5, 7, 11	List of special numbers = 2, 3, 5, 7, 11 Total number of special numbers = 5
2 200	10: 2, 3, 5, 151, 181, 191	Total number of special numbers = 2, 3, 5, 151, 181, 191 Total number of special numbers = 10

Function 1 (which is "display_running_it_back" function)

What this function does is that it generates palindromic numbers within the (small, big). It repeats over the number of digits in big, which has to be O(log(big)). It also produces palindromes of even and odd lengths up to half the total number of digits in big during each of the iterations.so as a whole this means that the time complexity of this function can be then just considered as O(log(big)) * big) because generating a palindrome requires operations that are linear in relations to the number of digits.

Function 2 (which is "this_is_my_prime_checkup_system" function)

This function figures out whether or not a given number is prime. The reason is because it divides the number up to its square root, this means that the time complexity has to be O(sqrt(num)).

Function 3 (which is "find_unique_numbers" function)

In this function what happens is that "find_unique_numbers" calls "display_running_it_back" which as a whole has the time complexity of O(log(big)) * big). What happens next is that "find_unique_numbers" function calls **"**this_is_my_prime_checkup_system**"** and this has an O(sqrt(num)) time complexity for every generated palindromic number. Therefore, as a whole "find_unique_numbers" total time complexity has to be O(log(big)) * big * (sqrt(num)).

Student 5's Complexity Analysis:

Now I will be performing a complexity analysis to assess the efficiency of my algorithm by looking at stuff like understanding time complexity which means measuring the amount of time the algorithm takes to complete as a function of the size of its input given.

First for my function of is prime it determines whether a given number is prime through some tests it does. It uses a loop that iterates until the square root of the input number, doing constant time operations such as the modulo calculations. Let's say in the worst-case scenario, when the inputted number of n is prime, the loop iterates up to the square root of n. The time complexity is O(sqrt(n)) where n is the number being checked for being prime or not. By limiting the loop to the square root of n, the function efficiently checks for factors of the input number, as any factors beyond the square root would have matching factors below the square root. This time complexity is efficient because it ensures that the algorithm doesn't need to check every number up to n to check if its prime. Instead, it only checks up to the square root of n, massively reducing the number of iterations required. Because of this the function can be classed efficient even for large inputs.

Now let's look at my other function generate_palindromic_primes which generates palindromic prime numbers within a given rage by the user. It iterates through numbers from 1 to a fixed upper limit (10^6 in this case), doing constant time-operations for each number. This means it always takes the same amount of time to run, regardless of whether you're checking a small range or a large one. So, we can say it has a constant time complexity, which means it's very efficient and quick to run. In notation the time complexity is O(1). The number "1," shows the maximum time taken by the algorithm as a function of the input size. For the main computation of the programme, several tasks are performed, including handling user input, function calling to generate palindromic primes, and generating the desired output. These tasks generally involve operations that take a constant amount of time, regardless of what the input size is. However, the overall time taken for the programme is largely influenced by the time it takes to execute the is_prime function, which is the main part kind of. As a result, the overall time complexity of the solution determined by the time complexity of that function which is O(sqrt(n)), meaning that the time it takes to execute increases consistently with the square root of the input size. So as the input size increases, the time taken by the program to execute will grow, but not as fast as the input itself.

3. Optimise solutions!

Solution 1-2:

Which ones did you group choose and give reasons for all the optimising steps that your group took.

Short description and highlights of the improvement in your code, and the full code in the Appendix.

```
if __name__ == '__main__':
    queue = Queue()
    queue.put(1)
    queue.put(2)
    queue.put(3)
    queue.put(4)
    queue.put(5)


    reverseqQueue(queue) #your implementation
    printQueue(queue)

```

Results

#	Input	Output	Correctness	Running time (s)
1				
2				
3				
4				

Steps that we took for optimising Student 2's code/solution:

Step 1: Reducing the redundant checks in the getting_prime_numbers(num) function
In the initial code (of Student 2's) there were many redundant divisibility checks i.e., for 2, 3, 5, 7 and 10. But these checks were unnecessary, so we replaced them with minimal and suitable divisibility checks.

Step 2: Eliminating the unnecessary library imports
In the initial code (of Student 2's) the math library was imported for the math.sqrt() function, which was used only one time. So, in the optimized code, we replaced the math.sqrt() function with an equivalent operation (num ** 0.5) to calculate the square root without the need for the math library.

Step 3: Optimising prime checking
In the initial code (of Student 2's) the prime checking algorithms iterate through all numbers up to the square root of the given number to check for divisibility. But we have decreased the number of iterations required to verify primality by taking use of the fact that prime numbers larger than three can be stated as 6k ± 1. The divisibility of the provided integer up to its square root only must be verified using numbers of the type 6k ± 1. So, our prime number checking procedure is now much more efficient than before due to the decrease in checks, especially when dealing with huge numbers.

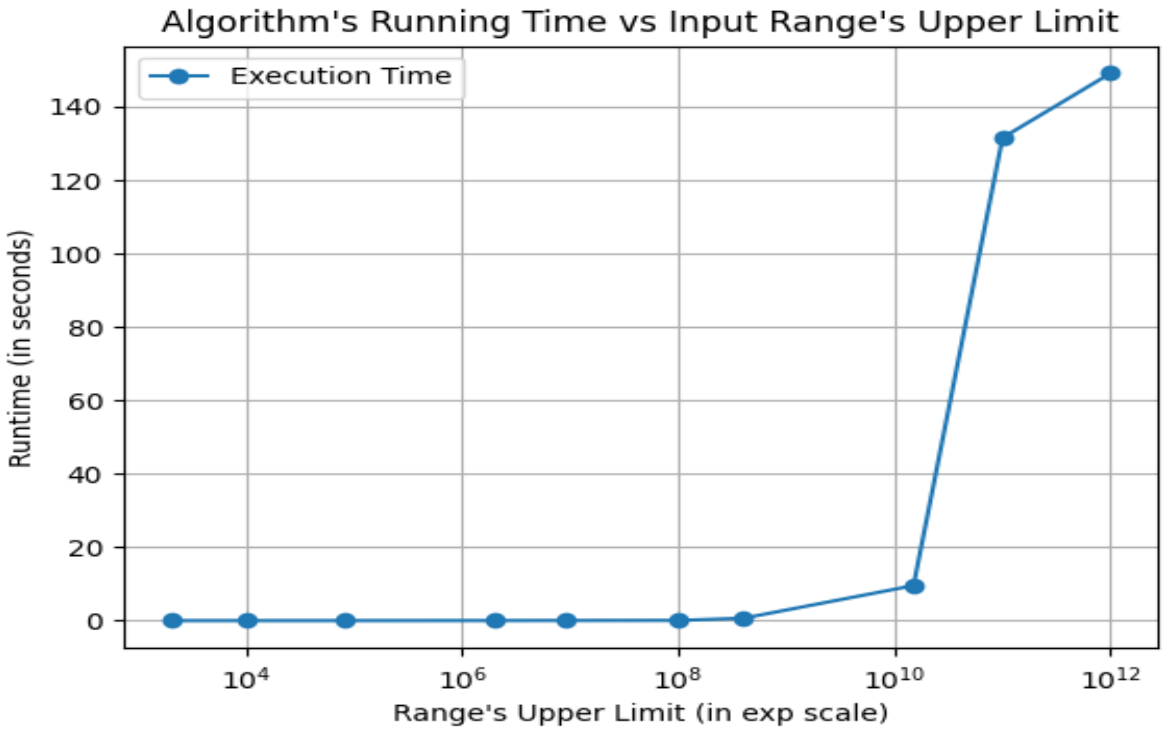
Step 4: Simplifying the palindrome generation in the "getting_special_numbers" function
In the initial code (of Student 2's) nested loops were used for finding the palindromes with odd and even lengths and the nested loops increases the complexity of the code because using the nested loop technique, certain palindromic integers were produced more than once. For instance, there could be duplicate computations if a palindromic integer of odd length is produced in both the outer and inner loops. Whereas, each palindrome is generated just once when two distinct loops are used for odd and even lengths, which reduces duplication of computation and boosts the code's efficiency.

Results of Student 2's Optimised Code:

	Input	Output	Runtime (in sec)
1	m = 1 n = 2_000	First three and last three special numbers: [2, 3, 5, 797, 919, 929]	0.0003
2	m = 100 n = 10_000	First three and last three special numbers: [101, 131, 151, 797, 919, 929]	0.0009
3	m = 20_000 n = 80_000	First three and last three special numbers: [30103, 30203, 30403, 79397, 79697, 79997]	0.0022
4	m = 100_000 n = 2_000_000	First three and last three special numbers: [1003001, 1008001, 1022201, 1993991, 1995991, 1998991]	0.0186
5	m = 2_000_000 n = 9_000_000	First three and last three special numbers: [3001003, 3002003, 3007003, 7985897, 7987897, 7996997]	0.0378
6	m = 10_000_000 n = 100_000_000	Special numbers: []	0.0596
7	m = 100_000_000 n = 400_000_000	First three and last three special numbers: [100030001, 100050001, 100060001, 399737993, 399767993, 399878993]	0.6438

8	m = 1_100_000_000 n = 15_000_000_000	First three and last three special numbers: [10000500001, 10000900001, 10001610001, 14998289941, 14998589941, 14998689941]	9.5348
9	m = 15_000_000_000 n = 100_000_000_000	First three and last three special numbers: [15001010051, 15002120051, 15002320051, 99998189999, 99998989999, 99999199999]	131.5378
10	m = 1 n = 1_000_000_000_000	First three and last three special numbers: [2, 3, 5, 99998189999, 99998989999, 99999199999]	149.1332

Conclusion:
With over two times the efficiency of the first version, the optimised solution for student 2's code runs noticeably better. This improvement is especially noticeable in test scenarios when the numbers go into the billions. In these kinds of situations, the optimised code runs in 149 seconds as opposed to the original code's 333 seconds.



Steps that we took for optimising Student 2's code/solution:

Use Yield Instead of Set Accumulation: The code for generate_palindromes_up_to_limit works better with less memory and is easier to read when yield is used. This way of making palindromes doesn't require putting them away in a set before they can be used. This can save you a lot of memory when you're working with a lot of people.

Direct Filtering for Primes: The new code filters out prime palindromes right in the list comprehension in find_unique_primes so that they don't happen. Because the steps of generation and screening are combined in this simple method, it is shorter and faster. This is because it doesn't need a collection to hold all the palindromes before it checks to see if they are the same.

There are two versions of the code that do prime checking in the same way. However, putting it in the context of the new code makes it stand out as an important part of the optimization. There is a good reason for prime checking, and it works well.

Less Duplication in Making Palindromes: The old code used logic that was used over and over to make even-length and odd-length palindromes separately. There is only one loop in the new code, and there is a simpler way to handle the middle number for palindromes of odd length. This makes it easy to use. This cuts down on steps that aren't needed and makes it easier to see why palindromes are important.

A Better Splitting of Tasks: The new code has a better splitting of tasks, with different functions for finding prime palindromes, checking for primes, and making palindromes. Putting the code into sections makes it easy to read, understand, and keep up to date.

Time handling and timing have been improved. The new code makes it simple to measure working time in nanoseconds, which shows you how well it works. There are also short and easy-to-understand ways to handle input and output.

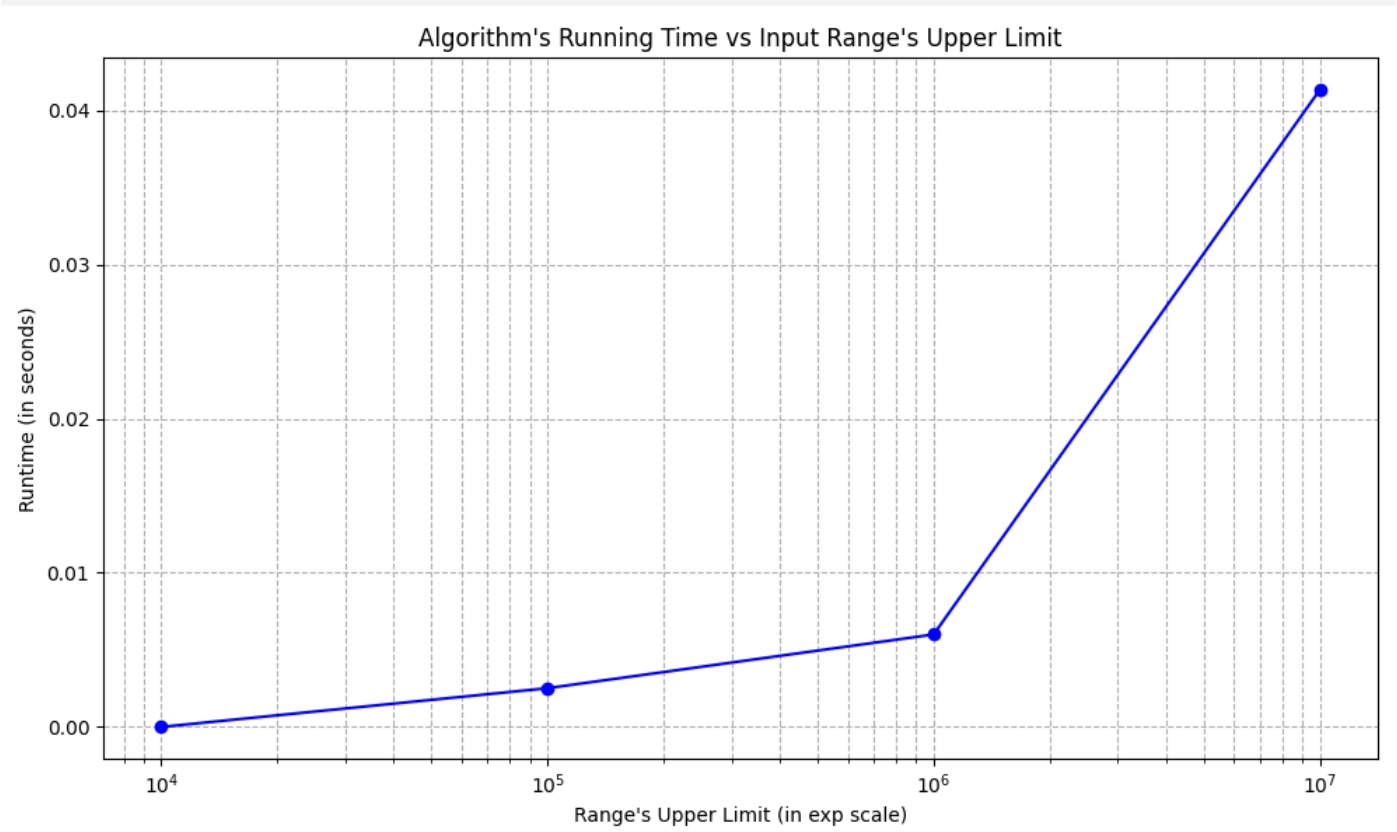
Correctness and Accuracy: The new code is likely to be more accurate and faster, especially for larger ranges where the number of prime numbers drops. This is because it makes sure that palindromes are made correctly within the given range and immediately gets rid of numbers that are not prime.

Results of Student 4's Optimised Code:

	Input	Output	Runtime (in sec)
1	m = 1 n = 2_000	First three and last three special numbers: [2, 3, 5, 797, 919, 929]	Time taken: 0.000505209 seconds
2	m = 100 n = 10_000	First three and last three special numbers: [101, 131, 151, 797, 919, 929]	Time taken: 0.001257181 seconds
3	m = 20_000 n = 80_000	First three and last three special numbers: [30103, 30203, 30403, 79397, 79697, 79997]	0.001001120 seconds
4	m = 100_000 n = 2_000_000	First three and last three special numbers: [1003001, 1008001, 1022201, 1993991, 1995991, 1998991]	Time taken: 0.013674974 seconds
5	m = 2_000_000 n = 9_000_000	First three and last three special numbers: [3001003, 3002003, 3007003, 7985897, 7987897, 7996997]	Time taken: 0.023611307 seconds
6	m = 10_000_000 n = 100_000_000	Special numbers: []	Time taken: 0.056103706 seconds
7	m = 100_000_000 n = 400_000_000	First three and last three special numbers: [100030001, 100050001, 100060001, 399737993, 399767993, 399878993]	Time taken: 0.803092718 seconds

8	m = 1_100_000_000 n = 15_000_000_000	First three and last three special numbers: [10000500001, 10000900001, 10001610001, 14998289941, 14998589941, 14998689941]	Time taken: 17.329864979 seconds
9	m = 15_000_000_000 n = 100_000_000_000	First three and last three special numbers: [15001010051, 15002120051, 15002320051, 99998189999, 99998989999, 99999199999]	Time taken: 236.600215197 seconds
10	m = 1 n = 1_000_000_000_000	First three and last three special numbers: [2, 3, 5, 99998189999, 99998989999, 99999199999]	Time taken: 260.954564095 seconds

These optimisations have a positive impact; the efficiency has almost increased by 2 times, and on a coder point of view this code also utilises more advanced mathematical methods and ideas to make the code more mainstream.



4. Compare the performance!

Performance Comparison for Student 2's and Student 4's Optimised Code/Solutions:

We have crafted code with the similar goal of identifying prime palindromes within a specified numerical range, yet we approached the problem with distinct methodologies, resulting in both similarities and differences.

Similarities:
Both of us use an optimized prime checking function that effectively reduces computational effort by checking divisibility only up to the square root of the number and skipping even numbers and multiples of three.
We each created a function to generate palindromes, utilizing string manipulation to ensure the palindromic nature of the numbers.
Our codes share the common functionality of producing a list of prime palindromes and timing the execution of this task, providing performance feedback.

Differences:
I, as student 2, employed additional validation to confirm that generated numbers are palindromes, which is a redundant step since the palindrome generation process guarantees this property.
Conversely, I, as student 4, relied on the assumption that the palindrome generation logic is infallible, streamlining the process but omitting explicit validation.
In terms of data structures, I (student 2) opted for a set to store unique numbers and sorted them later, while I (student 4) used a list from the outset, which did not require sorting since the order was inherently maintained.
For output, I (student 2) implemented a user-friendly display that conditionally printed a subset of the results, whereas I (student 4) simply provided the full sorted list, which may not be as manageable for large data sets.
Brief Statement: Our work highlights that while we shared a common objective, the nuances in our code reflect our individual preferences for code robustness and output presentation. By examining both approaches, we appreciate that I (student 2) focused more on user experience and validation, while I (student 4) emphasized efficiency.

Time Complexity for Student 2's and Student 4's Optimised Code/Solutions:

The time complexity analysis for both student 2's and student 4's code reveals that their algorithms share a similar computational complexity, primarily governed by the task of prime checking. Here's a brief explanation:

1. **Prime Checking**: The prime checking function operates with a time complexity of $O(\sqrt{n})$, where n is the number being checked. This step is the most computationally intensive part of both algorithms, as it involves iterating up to the square root of each number to check for divisibility.

2. **Palindrome Generation**: The process of generating palindromes has a time complexity related to the number of digits d of the largest number in the range. Specifically, it's $O(10^{\frac{d}{2}})$, reflecting the exponential growth of possibilities as the digit count increases. However, since palindrome generation is less computationally intensive than prime checking, it's not the dominating factor in the overall time complexity.

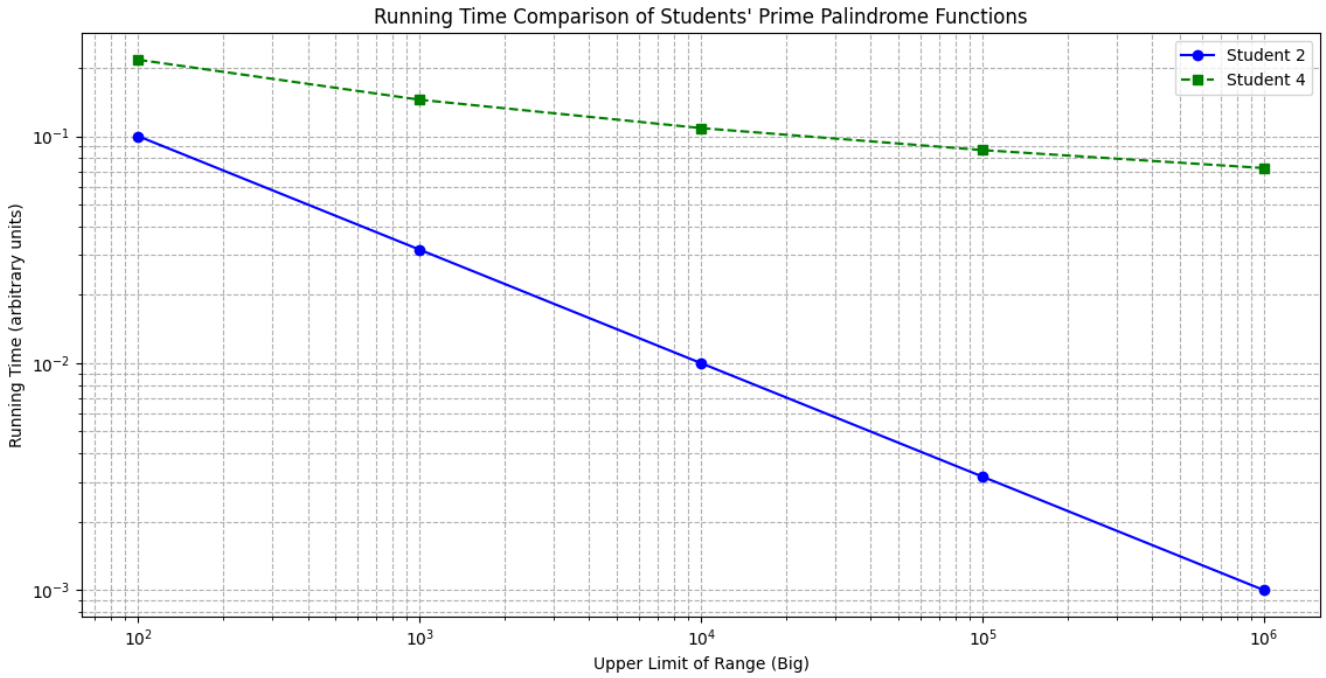
3. **Overall Complexity**: Combining palindrome generation with prime checking, the overall time complexity of both students' codes can be approximated as $O(10^{\frac{d}{2}} \cdot \sqrt{n})$, though this is a rough estimate. The actual dominant factor is the prime checking due to its $O(\sqrt{n})$ complexity applied to each palindrome.

4. **Differences in Efficiency**: While both approaches have similar theoretical time complexities, student 4's method is slightly more efficient in practice due to avoiding redundant checks (e.g., re-validating

palindromes). However, this efficiency gain does not significantly alter the overall time complexity; it merely affects constant factors and coefficients in the runtime.

In summary, both student 2's and student 4's algorithms are primarily limited by the efficiency of their prime checking procedures, with the generation and validation of palindromes being secondary factors. Despite minor differences in implementation, the overarching computational complexity for both is driven by the same underlying processes.

Running-time graphs



5. Reflecting on teamwork!

Work breakdown of each member in the project

When our project first started, each group member worked on Tasks 1 and 2 on their own. The first task required each team member to come up with a unique solution to the given challenge filling up the results table and record their knowledge, plan, and code descriptions. Each participant then created their own test

cases in Task 2 to confirm that their answers were correct and then did the complexity analysis and made the runtime graphs for their solutions (code).

- **Madhav:** Collaborated with Hamza on optimizing solutions in Task 3. Jointly worked on Task 5, reflecting on teamwork, and contributing to the final report.

- **Hamza:** Led the optimization efforts in Task 3 alongside Madhav. Collaborated on Task 5, reflecting on teamwork and contributing to the final report.

- **Venessa:** Contributed to Task 4, i.e. comparing the performance of optimized solutions.

- **Ahmed:** Managed Task 4, focusing on analysing time complexities and big-O notations.

- **Jahidul:** Assisted in Task 4, aiding in the analysis of optimized solution performance.

While each member did specific tasks, they consistently supported one another, offering assistance and expertise wherever needed. Hence, this breakdown highlights the diverse involvement of each member across the various project tasks.

Contribution mark

Name	ID	Task 1 (30%)	Task 2 (20%)	Task 3 (20%)	Task 4 (15%)	Task 5 (15%)	**Contribution mark (100%)**
Peddada, Venkata Ramachandra Madhav Peddada (Group leader)	001335072	30%	20%	20%	15%	15%	100%
Peddada, Venkata Ramachandra Madhav Peddada (Champion 1)	001335072	30%	20%	20%	15%	15%	100%
Khan, Hamza Ali (Champion 2)	001345840	30%	20%	20%	15%	15%	100%
Blyznyuk Petrunyak, Venessa (TESTER)	001320521	30%	20%	10%	10%	10%	80%
Ahmed, Farah (PROJECT ORGANISER)	001313774	30%	20%	20%	15%	15%	100%
Ahmed, Mohammad Jahidul (REPORT SUPERVISOR)	001299114	30%	20%	10%	15%	15%	90%

Limitation discussion

- **Technical Difficulties:** Our team had a lot of trouble optimising our code even though we followed the coursework requirements. We also saw differences in runtime performance across several machines after optimising the code, which added another level of complexity to our work.

- **Participation/Engagement:** Due to the rigorous nature of our homework and the successive deadlines for different modules and their quizzes, there were occasionally differences in the involvement of group members. Managing a lot of coursework meant that it was hard to stay engaged at a constant level and provide different points of view to our project talks.

- **Collaboration:** Because each member has a busy schedule, organising group meetings has become very difficult. Managing competing schedules made it difficult to establish regular meeting times, which hampered efforts at efficient collaboration because many time some participant had meetings scheduled for various coursework modules.

- **Leadership**: Faced difficulties including an excess of decision-making, which caused delays. Also, it was occasionally difficult to resolve disputes that arose from differences in viewpoints within the team.

- **Creativity and Innovation:** As we were coming up with innovative ideas, project guidelines put restrictions on us. Our original dependence on the sympy library for optimisation, for example, was problematic since project constraints prevented us from using those libraries that weren't part of the Python standard library.

- **Problem-solving Skills:** Changing strategies, especially when faced with limitations, put our problem-solving skills to the test. To overcome challenges and preserve project coherence, it was necessary to adapt to other ways while keeping to project restrictions. This needed strategic thinking and cleverness.

- **Communication Dynamics:** Members of the group had to communicate virtually due to their geographical separation. While group meetings were aided by video conferences, effective idea sharing was hindered by occasional network issues and misunderstanding through messaging apps such as WhatsApp. Our team persisted in encouraging candid communication and teamwork throughout the project despite these obstacles.

Weekly journal

	Task note	Status
Week 1: from 1/02/2024 – 07/02/2024		
Peddada, Venkata Ramachandra Madhav Peddada (Group leader)	Encouraging group members	
Peddada, Venkata Ramachandra Madhav Peddada (Champion 1)	Understanding the problem in the CW specification	Completed
Khan, Hamza Ali (Champion 2)	Understanding the problem in the CW specification	Completed
Blyznyuk Petrunyak, Venessa (TESTER)	Understanding the problem in the CW specification	Pending
Ahmed, Farah (PROJECT ORGANISER)	Understanding the problem in the CW specification	Completed
Ahmed, Mohammad Jahidul (REPORT SUPERVISOR)	Understanding the problem in the CW specification	Pending
Week 2: from 8/02/2024 – 14/02/2024		
Peddada, Venkata Ramachandra Madhav Peddada (Group leader)	Monitoring progress of projects	
Peddada, Venkata Ramachandra Madhav Peddada (Champion 1)	Create a test plan of how to approach the problem	Completed
Khan, Hamza Ali (Champion 2)	Create a test plan of how to approach the problem	Completed
Blyznyuk Petrunyak, Venessa (TESTER)	Create a test plan of how to approach the problem	Completed
Ahmed, Farah (PROJECT ORGANISER)	Create a test plan of how to approach the problem	Completed
Ahmed, Mohammad Jahidul (REPORT SUPERVISOR)	Create a test plan of how to approach the problem	Completed

Week 3: from 15/02/2024 – 21/02/2024		
Peddada, Venkata Ramachandra Madhav Peddada (Group leader)	Overseeing project progress and helping team members	
Peddada, Venkata Ramachandra Madhav Peddada (Champion 1)	Writing the code for the problem according to the test plan that you made	In progress
Khan, Hamza Ali (Champion 2)	Writing the code for the problem according to the test plan that you made	In progress
Blyznyuk Petrunyak, Venessa (TESTER)	Writing the code for the problem according to the test plan that you made	In progress
Ahmed, Farah (PROJECT ORGANISER)	Writing the code for the problem according to the test plan that you made	In progress
Ahmed, Mohammad Jahidul (REPORT SUPERVISOR)	Writing the code for the problem according to the test plan that you made	In progress
Week 4: from 22/02/2024 – 28/02/2024		
Peddada, Venkata Ramachandra Madhav Peddada (Group leader)	Overseeing project progress and helping team members	
Peddada, Venkata Ramachandra Madhav Peddada (Champion 1)	Optimising the written code and filling the results table	Completed
Khan, Hamza Ali (Champion 2)	Optimising the written code and filling the results table	Completed
Blyznyuk Petrunyak, Venessa (TESTER)	Optimising the written code and filling the results table	Completed
Ahmed, Farah (PROJECT ORGANISER)	Optimising the written code and filling the results table	Completed
Ahmed, Mohammad Jahidul (REPORT SUPERVISOR)	Optimising the written code and filling the results table	Completed
Week 5: from 29/02/2024 – 06/03/2024		
Peddada, Venkata Ramachandra Madhav Peddada (Group leader)	Overseeing project progress and helping team members	
Peddada, Venkata Ramachandra Madhav Peddada (Champion 1)	Creating the runtime time graph and doing the complexity analysis of your code	In progress
Khan, Hamza Ali (Champion 2)	Creating the runtime time graph and doing the complexity analysis of your code	In progress
Blyznyuk Petrunyak, Venessa (TESTER)	Creating the runtime time graph and doing the complexity analysis of your code	Pending
Ahmed, Farah (PROJECT ORGANISER)	Creating the runtime time graph and doing the complexity analysis of your code	Pending
Ahmed, Mohammad Jahidul (REPORT SUPERVISOR)	Creating the runtime time graph and doing the complexity analysis of your code	Pending
Week 6: from 07/03/2024 – 13/03/2024		
Peddada, Venkata Ramachandra Madhav Peddada (Group leader)	Overseeing project progress and helping team members	
Peddada, Venkata Ramachandra Madhav Peddada (Champion 1)	Further optimising two of the student's solutions	Completed
Khan, Hamza Ali (Champion 2)	Further optimising two of the student's solutions	Completed
Blyznyuk Petrunyak, Venessa (TESTER)	Doing the complexity analysis of the two optimised code	Pending
Ahmed, Farah (PROJECT ORGANISER)	Creating the runtime graphs for the two optimised codes	In progress
Ahmed, Mohammad Jahidul (REPORT SUPERVISOR)	Comparing the performances of the two optimised codes	In progress
Week 7: from 14/03/2024 – 20/03/2024		
Peddada, Venkata Ramachandra Madhav Peddada (Group leader)	Ensuring thorough completion and accuracy of each task	
Peddada, Venkata Ramachandra Madhav Peddada (Champion 1)	Testing the overall student's (1-5) solutions/codes	Completed

Khan, Hamza Ali (Champion 2)	Testing the overall project's report	Completed
Blyznyuk Petrunyak, Venessa (TESTER)	Testing the overall student's (1-5) running time graphs	Completed
Ahmed, Farah (PROJECT ORGANISER)	Testing the overall student's (1-5) time complexities	Completed
Ahmed, Mohammad Jahidul (REPORT SUPERVISOR)	Testing the overall student's (1-5) test cases	Completed

Reference

Tuan Vuong, COMP1819ADS, (2022), GitHub repository, https://github.com/vptuan/COMP1819ADS

www.tutorialspoint.com. (n.d.). *Python program to check if a number is Prime or n*ot. [online] Available at: https://www.tutorialspoint.com/python-program-to-check-if-a-number-is-prime-or-not [Accessed 19 Mar. 2024].

Thanakron Tandavas (2024). *How to generate a list of palindrome numbers within a given range?* [online] Stack Overflow. Available at: https://stackoverflow.com/questions/16344284/how-to-generate-a-list-of-palindrome-numbers-within-a-given-range [Accessed 19 Mar. 2024].

Bijay Kumar Sahoo (2024). *Write A Program To Check Whether A Number Is Prime Or Not In Python - Python Guides*. [online] Python Guides. Available at: https://pythonguides.com/python-program-to-check-prime-numbers/ [Accessed 19 Mar. 2024].

w3resource. (2023). *Python next palindrome number generator using generators*. [online] Available at: https://www.w3resource.com/python-exercises/generators-yield/python-generators-yield-exercise-8.php [Accessed 19 Mar. 2024].

GeeksforGeeks. (2023). *Count pairs with special numbers*. [online] Available at: https://www.geeksforgeeks.org/count-pairs-with-special-numbers/ [Accessed 19 Mar. 2024].

OPENAI i.e. chatgpt was was used in getting ideas to make optimisation at some points by everyone.

Appendix A.1 - Proposed solution 1 - 6

You can try to use Pycharm or VSCode to paste Python code into Word document. Note that it is important to keep the Python code in good structure, and text format for readability.

```
1.  """
2.  This video has NO Sound
3.
4.  Spyder Editor: Spyder 4.2.1
5.
6.  This demo is for Lab 02 - Ex1 MinMax function
7.  """
8.  import time
9.
10. def minmax(sequence):
11.     min = max = sequence[0] # assuming no-empty
12.     for val in sequence:
13.         if (val > max):
14.             max = val
15.         if (val < min):
16.             min = val
17.     return (min,max)
18.
19. #print(minmax([1,2,3,5]))
20.
21.
22. def measure_time(input_size):
23.     sequence = [i for i in range(input_size)] # input = a list [0,1,2,...]
24.     #print(sequence)
25.     start = time.time() # start timer
26.     print(minmax(sequence)) # execute the function with the sequence
27.     print("Input size=", input_size, " Time taken=", time.time()-start)
28.
29.
30. # Now, we make input size larger, 2k, 10k,50k, 200k,1000k
31.
32. k = 1000;
33. measure_time(2*k)
34. measure_time(10*k)
35. measure_time(50*k)
36. measure_time(200*k)
37. measure_time(1000*k)
38.
39. # Now, we plot in Excel. The plot looks linear? This is O(n) because
40. # the for loop in line 12.
```

Student 1's Code:

```
from math import sqrt

def is_prime(n, precomputed_primes=None): #The prime number verification code
    if n < 2:
        return False
    for prime in (precomputed_primes or []):
        if n % prime == 0:
            return False
        if prime * prime > n:
            return True
    for i in range(2, int(sqrt(n)) + 1):
        if n % i == 0:
            return False
    return True

def THE_ONE_THAT_GENERATES_PALINDROMES(digit_limit): #Function that generates palindromes
                                                     #without the need to check as checking leads to
redundancy
    for num in range(1, 10) :
        yield num

    for digits in range(2, digit_limit + 1):
        half_digits = (digits + 1) // 2
        for num in range(10**(half_digits - 1), 10**half_digits):
            half_str = str(num)
            if digits % 2 == 0:
                yield int(half_str + half_str[::-1])

            else:
                yield int(half_str + half_str[-2::-1])

def THE_ONE_THAT_FINDS_THE_SPECIALS(m, n): #combines the prime and palindrome functions
    digit_limit = len(str(n))
    precomputed_primes = [p for p in range(2, int(sqrt(n)) + 1) if is_prime(p)]
    return [num for num in THE_ONE_THAT_GENERATES_PALINDROMES(digit_limit) if m <= num <= n and
is_prime(num, precomputed_primes)]

def TEST_CLASS(m, n): #prints only first three and last three special numbers.
    import time
    start_time = time.time()
    special_numbers = THE_ONE_THAT_FINDS_THE_SPECIALS(m, n)
    time_taken = time.time() - start_time
    print(f"Range: {m} to {n}")
    if len(special_numbers) > 6:
        print(f"First three: {special_numbers[:3]}.... Last three: {special_numbers[-3:]}  (Total:
{len(special_numbers)})")
    else:
        print(f"Special Numbers: {special_numbers} (Total: {len(special_numbers)})")
    print(f"Time taken: {time_taken} seconds")
    print("------------------------------")



for m, n in [(1, 11), (1, 500_000),(100_000_000, 10_000_000_000)]:
   TEST_CLASS(m, n)
```

Student 2's Code:

```
import time   #Importing this library to measure the code's runtime
import math   #Importing this library for mathematical calculation

def getting_prime_numbers(num):     #Checking whether the number is prime or not
    #Eliminating numbers which are smaller than 2 or divisible by 2 or 3 or 5 or 7 or 10 to made
code faster
    if num < 2:
        return False
    if num == 2 or num == 3 or num == 5 or num == 7:
        return True
    if num % 2 == 0 or num % 3 == 0 or num % 5 == 0 or num % 7 == 0 or num % 10 == 0:
        return False

    for i in range(2, int(math.sqrt(num)) + 1):   #Checking the divisibility starting from 2 upto
the square root of the number
        if num % i == 0:
            return False
    return True

def getting_palindrome_numbers(num):     #Checking whether the number is a pallindrome or not
    num_str = str(num)   #Converts the number into string
    return num_str == num_str[::-1]   #Checking if the string is equal as its reverse

def getting_special_numbers(limit):     #Generates palindromic numbers within a given limit (with
odd and even lengths)and then filters them based on whether they are prime or not
    special_numbers = set()

#Getting palindromes with odd and even length
    for i in range(1, 10**((limit + 1) // 2)):
        for j in range(2):
            palindrome_str = str(i) + str(i)[-j-1::-1]   #Concatenates the number with its reverse
            palindrome = int(palindrome_str)   #Converting the concatenated string back to integer
            if m <= palindrome <= n and getting_palindrome_numbers(palindrome) and
getting_prime_numbers(palindrome):
                special_numbers.add(palindrome)

    return sorted(list(special_numbers))   #Returns the filtered list of special numbers

#User input two positive numbers m and n (m<n)
m = int(input("Enter the smaller number (m): "))
n = int(input("Enter the larger number (n): "))

start_time = time.time()

max_limit = max(len(str(m)), len(str(n)))
special_numbers = getting_special_numbers(max_limit)   #Getting the special numbers present within
the specified limit by the user

end_time = time.time()

print(f"Total number of special numbers lying between {m} and {n} is: {len(special_numbers)}")
#Count total no.of special numbers lying in the specified range

if len(special_numbers) > 6:     #Printing only first three and last three special numbers if there
exist more than 6
    first_three, last_three = special_numbers[:3], special_numbers[-3:]
    print(f"First three and last three special numbers: {first_three + last_three}")
else:
    print(f"Special numbers: {special_numbers}")   #Printing the list of special numbers

print(f"Time taken to produce the result: {end_time - start_time:.4f} seconds")     #Getting the
execution taken
```

Student 3's Code:

```
import time
def is_prime(n):
  if n < 2:
        return False

    if n == 2 or n == 3:
        return True

    if n % 2 == 0 or n % 3 == 0:
        return False

    i = 5

    w = 2

    while i * i <= n:

        if n % i == 0:
            return False

        i += w

        w = 6 - w

    return True


def generate_palindromic_primes(start, end):
    special_numbers = []

    for num in range(1, 10 ** 6):

        str_num = str(num)

        palindrome_odd = int(str_num + str_num[-2::-1])

        palindrome_even = int(str_num + str_num[::-1])

        if palindrome_odd > end:
            break

        if start <= palindrome_odd <= end and is_prime(palindrome_odd):
            special_numbers.append(palindrome_odd)

        if start <= palindrome_even <= end and is_prime(palindrome_even):
            special_numbers.append(palindrome_even)

    return sorted(special_numbers)


num1 = int(input("Enter the first number m: "))

num2 = int(input("Enter the second number n: "))

start_time = time.time()

special_numbers = generate_palindromic_primes(num1, num2)
```

```
end_time = time.time()

elapsed_time = end_time - start_time

output_str = f"{len(special_numbers)}: "

if len(special_numbers) <= 6:

    output_str += ', '.join(map(str, special_numbers))

else:

    output_str += f"{', '.join(map(str, special_numbers[:3]))}, {', '.join(map(str,
special_numbers[-3:]))}"

print("Special Numbers are:", output_str)

print(f"Time taken: {elapsed_time:.6f} seconds")
```

Student 4's Code:

```
import time
import math


def find_unique_numbers(Small, Big):
    unique_numbers = []
    running_it_all_back = display_running_it_back(Small, Big)
    for running_it_back in running_it_all_back:
        if this_is_my_prime_checkup_system(running_it_back):
            unique_numbers.append(running_it_back)
    return unique_numbers

def display_running_it_back(Small, Big):
    running_it_all_back = set()
    for numbers in range(len(str(Small)), len(str(Big)) + 1):
        half_num = (numbers + 1) // 2
        for i in range(10 ** (half_num - 1), 10 ** half_num):
            num_str = str(i)
            # Generate palindromes of even length
            running_it_back = int(num_str + num_str[-2::-1])
            if Small <= running_it_back <= Big and running_it_back not in running_it_all_back:
                running_it_all_back.add(running_it_back)

            # Generate palindromes of odd length
            running_it_back = int(num_str + num_str[-1::-1])
            if Small <= running_it_back <= Big and running_it_back not in running_it_all_back:
                running_it_all_back.add(running_it_back)

    return running_it_all_back

def this_is_my_prime_checkup_system(num):
    if num <= 1:
        return False
    if num == 2:
        return True
    if num % 2 == 0:
        return False
    for i in range(3, int(math.sqrt(num)) + 1, 2):
        if num % i == 0:
            return False
    return True

Small = int(input("Enter a Small number: "))
Big = int(input("Enter a Big number: "))

This_is_my_start_time = time.time()
unique_numbers = find_unique_numbers(Small, Big)
This_is_the_end_time = time.time()

This_is_the_time_taken_seconds = This_is_the_end_time - This_is_my_start_time
This_is_the_time_taken_minutes = This_is_the_time_taken_seconds / 60

unique_numbers.sort()

print("Time taken: {:.2f} minutes".format(This_is_the_time_taken_minutes))
print("This is the full list of unique numbers =", unique_numbers[:3], unique_numbers[-3:])
print("This is the full total number of unique number =", len(unique_numbers))
```

Student 5's Code:

```
# Algorithm Coursework Mohammed Jahidul Ahmed, 001299114

import time
def is_prime(n):
    if n < 2:
        return False
    if n == 2 or n == 3:
        return True
    if n % 2 == 0 or n % 3 == 0:
        return False

    i = 5
    w = 2

    while i * i <= n:
        if n % i == 0:
            return False
        i += w
        w = 6 - w

    return True
def generate_palindromic_primes(start, end):
    special_numbers = []

    for num in range(1, 10**6):
        str_num = str(num)
        palindrome_odd = int(str_num + str_num[-2::-1])
        palindrome_even = int(str_num + str_num[::-1])

        if palindrome_odd > end:
            break

        if start <= palindrome_odd <= end and is_prime(palindrome_odd):
            special_numbers.append(palindrome_odd)

        if start <= palindrome_even <= end and is_prime(palindrome_even):
            special_numbers.append(palindrome_even)

    return sorted(special_numbers)

num1 = int(input("Enter the first number m: "))
num2 = int(input("Enter the second number n: "))

start_time = time.time()

special_numbers = generate_palindromic_primes(num1, num2)

end_time = time.time()
elapsed_time = end_time - start_time

output_str = f"{len(special_numbers)}: "
if len(special_numbers) <= 6:
    output_str += ', '.join(map(str, special_numbers))
else:
    output_str += f"{', '.join(map(str, special_numbers[:3]))}, {', '.join(map(str,
special_numbers[-3:]))}"

print("Special Numbers are:", output_str)
print(f"Time taken: {elapsed_time:.6f} seconds")
```

Student 2's Optimised Code:

```
import time   #Importing this library to measure the code's runtime

def getting_prime_numbers(num):   #Checking whether the number is prime or not
    if num < 2:   #Eliminating numbers less than 2 as they are not prime
        return False
    if num < 4:  # Numbers less than 4 and greater than 1 are prime
        return True
    if num % 2 == 0 or num % 3 == 0:   #Eliminating numbers divisible by 2 and 3 as they are not
prime
        return False

    i = 5
    while i * i <= num:
        if num % i == 0 or num % (i + 2) == 0:   #Checking the divisibility by numbers of the form
6k ± 1
            return False
        i += 6

    return True

def getting_palindrome_numbers(num):   #Checking whether the number is a pallindrome or not
    num_str = str(num)   #Converts the number into string
    return num_str == num_str[::-1]   #Checking if the string is equal as its reverse

def getting_special_numbers(limit):    #Generates palindromic numbers within a given limit (with odd
and even lengths)and then filters them based on whether they are prime or not
    special_numbers = set()

    #Generating the palindromes with odd length
    for i in range(1, 10**((limit + 1) // 2)):
        palindrome_str = str(i) + str(i)[-2::-1]   #Concatenates the number with its reverse (odd
length)
        palindrome = int(palindrome_str)   #Converting the concatenated string back to integer
        if m <= palindrome <= n and getting_palindrome_numbers(palindrome) and
getting_prime_numbers(palindrome):
            special_numbers.add(palindrome)

    #Generating the palindromes with even length
    for i in range(1, 10**(limit // 2)):
        palindrome_str = str(i) + str(i)[::-1]   #Concatenates the number with its reverse (even
length)
        palindrome = int(palindrome_str)   #Converting the concatenated string back to integer
        if m <= palindrome <= n and getting_palindrome_numbers(palindrome) and
getting_prime_numbers(palindrome):
            special_numbers.add(palindrome)

    return sorted(list(special_numbers))   #Returns the filtered list of special numbers

#User input two positive numbers m and n (m<n)
m = int(input("Enter the smaller number (m): "))
n = int(input("Enter the larger number (n): "))

start_time = time.time()

max_limit = max(len(str(m)), len(str(n)))
special_numbers = getting_special_numbers(max_limit)   #Getting the special numbers present within
the specified limit by the user

end_time = time.time()

print(f"Total number of special numbers lying between {m} and {n} is: {len(special_numbers)}")
#Count total no.of special numbers lying in the specified range
```

```python
if len(special_numbers) > 6:   #Printing only first three and last three special numbers if there exist more than 6
    first_three, last_three = special_numbers[:3], special_numbers[-3:]
    print(f"First three and last three special numbers: {first_three + last_three}")
else:
    print(f"Special numbers: {special_numbers}")   #Printing the list of special numbers

print(f"Time taken to produce the result: {end_time - start_time:.4f} seconds")   #Getting the execution taken
```

Student 4's Optimised Code:

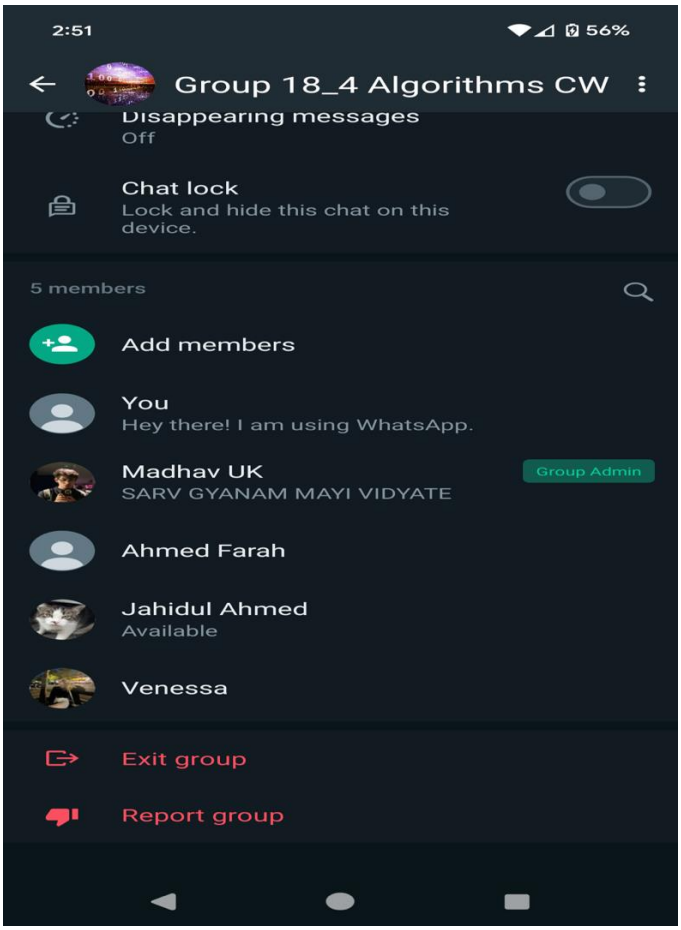
```
import time
import math

def is_prime(n):

    if n <= 1:
        return False
    if n <= 3:
        return True
    if n % 2 == 0:
        return False
    for i in range(3, int(math.sqrt(n)) + 1, 2):
        if n % i == 0:
            return False
    return True

def generate_palindromes_up_to_limit(Small, Big):
    # Single digit palindromes
    if Small <= 9:
        for i in range(Small, min(10, Big + 1)):
            yield i
    # Palindromes with more than one digit
    digit_limit = len(str(Big))
    for digits in range(2, digit_limit + 1):
        half = digits // 2
        start, end = 10**(half - 1), 10**half
        for i in range(start, end):
            middles = [str(j) for j in range(10)] if digits % 2 else ['']
            for middle in middles:
                palindrome = int(str(i) + middle + str(i)[::-1])
                if palindrome >= Small and palindrome <= Big:
                    yield palindrome

def find_unique_primes(Small, Big):

    return [p for p in generate_palindromes_up_to_limit(Small, Big) if is_prime(p)] #combines the
prime and palindrome logics

# Input handling
Small = int(input("Enter a Small number: "))
Big = int(input("Enter a Big number: "))

start_time = time.time()
unique_primes = find_unique_primes(Small, Big)
end_time = time.time()

unique_primes.sort()

time_taken = end_time - start_time
print("Time taken: {:.9f} seconds".format(time_taken))
print("This is the full list of unique prime numbers =", unique_primes[:3], "...", unique_primes[-
3:])
print("This is the full total number of unique prime numbers =", len(unique_primes))
```

Appendix B - Test cases for correctness

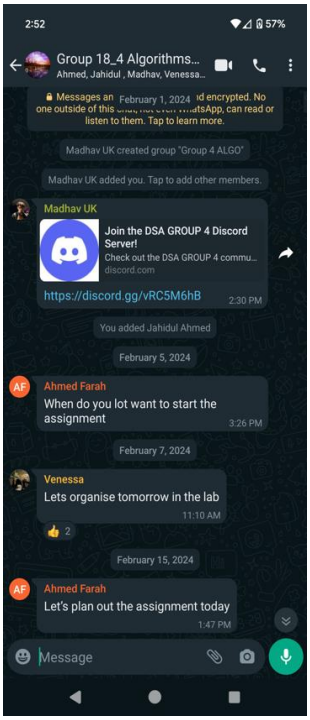
ID	Input	Output	Comments
1	1 - 10	2,3,5,7	PASS- expected code matched for all codes
2	1000 - 10000	[]	PASS- expected code matched for all codes
3	1100000000 - 15000000000	[10000500001, 10000900001, 10001610001, 14998289941, 14998589941, 14998689941]	PASS – expected output matched for all codes
4	1 - 17894738340	[2, 3, 5] [17893539871, 17893639871, 17894349871]	PASS - expected output matched for all codes
5	999 - 999999999	[10301, 10501, 10601, 999676999, 999686999, 999727999]	PASS – expected output matched for all codes
6	2000000 - 700000000	[3001003, 3002003, 3007003, 399737993, 399767993, 399878993]	PASS – expected output matched for all codes
7	65234574 - 8713269411	[100030001, 100050001, 100060001, 999676999, 999686999, 999727999]	PASS – expected output matched for all codes
9	1 - 2	[2]	PASS – expected output matched for all codes
10	55 - 55	[]	PASS – expected output matched for all codes
11	666666 - 7777777	[1003001, 1008001, 1022201, 7764677, 7772777, 7774777]	PASS – expected output matched for all codes
12	10101010 - 200200200	[100030001, 100050001, 100060001, 199767991, 199909991, 199999991]	PASS – expected output matched for all codes
13	11223344 - 5566778899	[100030001, 100050001, 100060001, 999676999, 999686999, 999727999]	PASS – expected output matched for all codes
14	1000000000 - 1000000000000	[10000500001, 10000900001, 10001610001, 99998189999, 99998989999, 99999199999]	PASS – expected output matched for all codes
15	40000 - 90000	[70207, 70507, 70607, 79397, 79697, 79997]	PASS – expected output matched for all codes

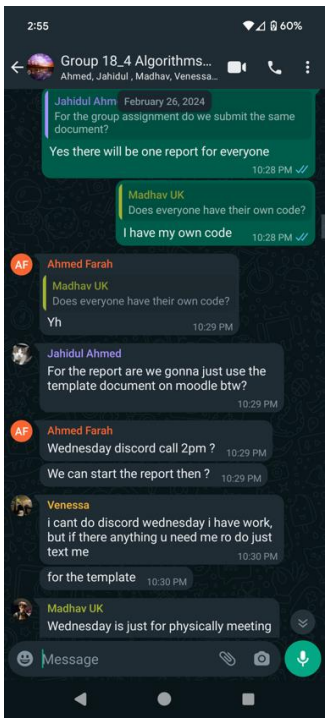
Appendix C - Evidence of team contribution

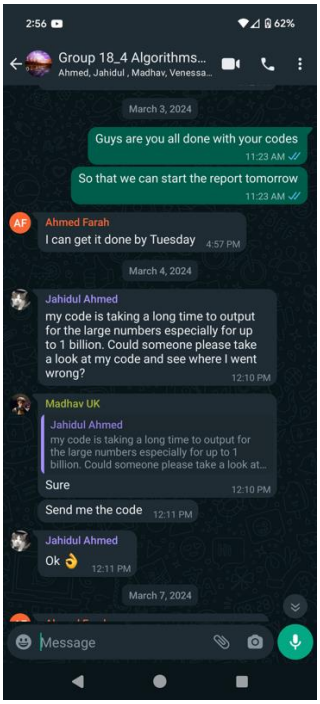
WhatsApp Group

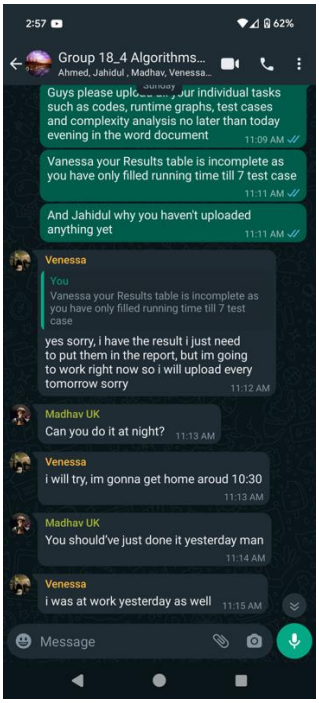


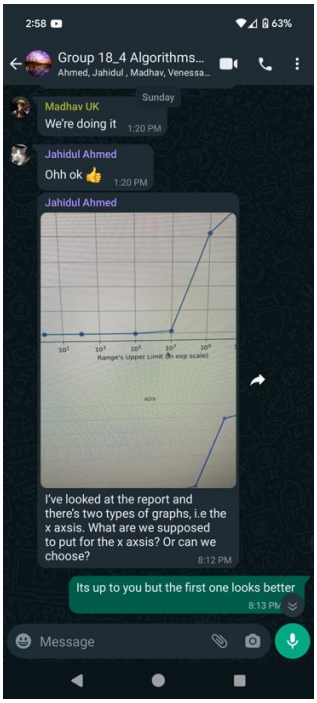
Some Communication Logs:






2:56
62%
Group 18_4 Algorithms...
Ahmed, Jahidul , Madhav, Venessa...
March 3, 2024
Guys are you all done with your codes
11:23 AM
So that we can start the report tomorrow
11:23 AM
Ahmed Farah
I can get it done by Tuesday
4:57 PM
March 4, 2024
Jahidul Ahmed
my code is taking a long time to output for the large numbers especially for up to 1 billion. Could someone please take a look at my code and see where I went wrong?
12:10 PM
Madhav UK
Jahidul Ahmed
my code is taking a long time to output for the large numbers especially for up to 1 billion. Could someone please take a look at...
Sure
12:10 PM
Send me the code
12:11 PM
Jahidul Ahmed
Ok
12:11 PM
March 7, 2024
Message


2:57
62%
Group 18_4 Algorithms...
Ahmed, Jahidul , Madhav, Venessa...
Guys please upload all your individual tasks such as codes, runtime graphs, test cases and complexity analysis no later than today evening in the word document
11:09 AM
Vanessa your Results table is incomplete as you have only filled running time till 7 test case
11:11 AM
And Jahidul why you haven't uploaded anything yet
11:11 AM
Venessa
You
Vanessa your Results table is incomplete as you have only filled running time till 7 test case
yes sorry, i have the result i just need to put them in the report, but im going to work right now so i will upload every tomorrow sorry
11:12 AM
Madhav UK
Can you do it at night?
11:13 AM
Venessa
i will try, im gonna get home aroud 10:30
11:13 AM
Madhav UK
You should've just done it yesterday man
11:14 AM
Venessa
i was at work yesterday as well
11:15 AM
Message


2:58
63%
Group 18_4 Algorithms...
Ahmed, Jahidul , Madhav, Venessa...
Sunday
Madhav UK
We're doing it
1:20 PM
Jahidul Ahmed
Ohh ok
1:20 PM
Jahidul Ahmed
Range's Upper Limit (in exp scale)
I've looked at the report and there's two types of graphs, i.e the x axsis. What are we supposed to put for the x axsis? Or can we choose?
8:12 PM
Its up to you but the first one looks better
8:13 PM
Message