Algorithms and Data Structures (ADS) - COMP1819

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

Group Name: COMP1819/ITLab/01-21_05

1. **Create unique solutions!**

Student 1-6's solution

Md Abdul Raihan Tanzim 001341954:

Understanding the Problem

The task is to locate special numbers in a specified range. Prime numbers that are also palindromic—that is, they read the same both forward and backward—are referred to as special numbers. The aim is to generate these unique numbers and display certain information about them, like the total count and the first and last three digits.

Approach to Solving the Problem

I started by coming up with several strategies and looking for advice from a variety of sources, including online tutorials, lecture slides, and published literature, to produce my plan. I studied the properties of special numbers, especially prime and palindromic numbers, and considered ways to produce and recognise them within a range in an effective manner. I created a mind map that focused on the problem's clarity, simplicity, and efficacy after gaining insights and weighing several approaches. To consistently produce palindromic numbers and discover prime numbers, this approach required the development of algorithms, structures, and a great deal of

debugging. Eventually, this would lead to the identification of special numbers within the given range.

Description of the Code

I've defined the functions to produce palindromic numbers and verify primality. After that, I culled palindromic prime numbers that fell inside the specified range. Lastly, I've asked the user to enter a range of numbers for the beginning and end, computed and shown the unique numbers that fall inside that range, and displayed the execution time. But I've discovered that the code isn't optimised, so when the input gets bigger, it could not run as well.

Brazington, Ella 001305508

Understanding the problem

The problem we're tackling is all about finding and listing special numbers within a range set by the user. Here's the breakdown: first off, the program asks the user for two numbers. These special numbers must meet two conditions: prime and palindromic. To check if a number is prime, we run a check from 2 to the square root of the number. For palindromes, we convert each number into a string and compare it with its reverse. Once we've got our special numbers within the given range, the program tells us how long it took, how many special numbers were found, and lists the first and last three special numbers.

Approach to solving the problem

Initially, I tackled this challenge by diving into coding practices and exploring various data structures. I experimented with multiple versions of the code based on my research, but none of them quite matched the requirements.

I realized the need for optimization, I started a personal research project to find better ways to improve my code's efficiency. I also reached out to peers for fresh perspectives and insights. Through our collaboration and further exploration, I managed to refine the code, but it still fell short of meeting the requirements.

Description of the code

My Python script is designed to find and display special numbers (a number that is both prime and a palindrome) within a user-specified range.

The `'main()'` function is the entry point in my code, it prompts the user to input two integers (m and n) which represent the lower and upper bounds of the range. The code then begins to record the start time. The 'specialNumbers()' function finds the special numbers from within this range. The special numbers are those that are both prime and palindromic, which are determined using the '`prime()`' and '`palindrome()`' functions. Once the search has been completed the code calculates and displays the execution time using the 'display_execution_time()' function. It also presents the count of special numbers identified within the range and it lists the first and last special numbers.

Ainhoa Prada 001352985:

Understanding the Problem

The objective of this activity is to find all the prime palindromes within a range of numbers. Initially, we will code a solution that pass as many predetermined test as possible, trying different algorithms, structures and data types. From the five solutions we will chose the two

best programs to be optimized, and then compare between them to find the fastest solution for the problem.

Approach to Solving the Problem

My initial approach involved researching various programming constructs such as loops, higher-order functions, and recursion to determine the optimal coding strategy. Further research focused on identifying the best algorithm for the two main functions, primes and palindromes. The goal was to try out various algorithms and record the speed of each solution for comparison. However, upon coding the initial solution, I realized the challenge of processing a trillion numbers solely with Python's arithmetic functions. This led me to seek tutorials on code optimization, where I learned about libraries like Numba and techniques like memoization for caching numbers. Nonetheless, the most crucial aspect was improving the algorithm itself. Thus, mathematical knowledge became integral to the project, guiding the discovery of patterns such as the pyramidal behaviour of prime palindromes and their distribution.

Description of the Code

The code is separated in three functions, a function that calculates prime palindromes, one to skip intervals with no prime palindromes and one to print the outputs:

- Primes: By discarding values such as zero and one, checks if a number is prime by verifying if its division by numbers below its square root results in zero. This is because the square root of a number is divisible by that number, so we only need to check factors below it. It returns a Boolean.

- Palindromes: using a while loop, takes a number, reverses it, and compares it to the original given number. It returns a Boolean.

If we focus on a number and attempt to create palindromic primes based on it, we'll realize there are no palindromic numbers with even digits (2, 4, 6, 8, 10...) except for 11, as all

numbers with that digit count are divisible by 11. Thus, I decided to eliminate these intervals from my numeric range, using a while loop that would increase m or decrease n if it fell within one of these intervals. Meanwhile, I incremented m by +1 to make m = n and terminate the loop.

Lubis Aliyah 001315223

Understanding the problem:

To tackle the problem, I'd start by understanding that it requires identifying prime and palindromic numbers within a given range. I'd prompt the user for two positive integers, ensuring the first is smaller than the second. Then, I'd iterate through the range, checking each number for both prime and palindrome properties. Skipping certain numbers for efficiency, I'd collect and display the special numbers found, following the specified format. This approach aims to efficiently find and display special numbers within the given range, adhering to the constraints provided.

Approach to solving the problem:

Given a task to identify special numbers (palindromic and prime), I attempted to locate a formula that could simultaneously identify both types of numbers. Unfortunately, my search for such a formula was unsuccessful. Consequently, I decided to pursue an alternative approach: first, I searched for palindromic numbers within the given range, recognizing that this subset of numbers would likely be smaller than the set of prime numbers. Then, I proceeded to identify prime numbers among the palindromes found. This strategy was motivated by the understanding that palindromic numbers typically have fewer members than the set of prime numbers, potentially resulting in a faster runtime. This optimization was particularly crucial since we aimed to complete the program within one hour, which presented a significant challenge.

Description of the code:

I've written a program to find and show "special numbers" within a user-specified range in this Python program. To do this, I've created a function called {is_palindrome(num)} that checks if a given number is a palindrome—that is, if it reads the same both forward and backward. In addition, I have tested whether an integer is prime using the {isprime(num)} function from the SymPy library.

I prompt the user to enter two positive integers, {m} and {n}, into the `main()` function, making sure that {m} is smaller than {n}. Then, as even numbers and integers ending in five are unlikely to be prime, I iterate across the given range. I determine whether every remaining integer is both prime and a palindrome. If so, I include it in a list of the special numbers.

Finally, I print the list of special numbers along with their count. If the count exceeds 5, I display the first three and last three special numbers. The program provides efficient algorithms for checking palindrome and primality, optimizing the search for special numbers within the specified range while also calculating the execution time for reference.

Petko Georgiev 001318621:

Understanding the problem:

The problem entails locating special numbers within a given range. Special numbers are defined as palindromic prime numbers. However, it is necessary to reduce the time required for each test case by optimizing the code. Exploring various algorithms and Python libraries to solve this problem. After everyone has finished their code, choose the top two programmes that run the fastest.

Approach to solving the problem:

The hardest part of this algorithm was for the palindromic prime numbers to give an efficient solution while taking care of the performance limits. But not just this only there was more

challenges that could also include managing large input ranges and optimizing the algorithm to satisfy runtime specifications. Furthermore, making sure the algorithm is correct and efficiently managing edge cases are crucial factors. I explored every type of sieve algorithm not just these but also using maps, timeit, primerange, itertools but also python library like sympy. But also try to find out any numerical method or generating palindrome numbers

Description of the code:

After i explored all the algorithms and python library i decided to continue my program with Sieve of Eratosthenes. The python program identifies palindromic prime numbers within a specified range. This function determines whether an integer number is a palindrome by taking it as input. Then generate palindromic prime numbers up to a certain length. It generates palindromic primes of either odd or even lengths. After this a list of palindromic primes within the length with the sieve of Eratosthenes algorithm for prime number generation. This function generates prime numbers up to a given limit of use. It iterates over the numbers up to the square root of the limit, marking multiples of each prime number as non-prime. This program runs all the test cases iteratively on given range and reports the count and specific palindromic prime numbers found within each range. Overall, this program efficiently finds and prints special numbers within specified ranges using sequential processing.

Results

Md Abdul Raihan Tanzim 001341954:

#	Input	Output	Running time (s)
1	`1 2000`	`List of special numbers = [2, 3, 5, 797, 919, 929]` `Total number of special numbers = 20`	`Execution time: 0.000000 seconds`
2	`100 10_000`	`List of special numbers = [101, 131, 151, 797, 919, 929]` `Total number of special numbers = 15`	`Execution time: 0.000000 seconds`

3	`20_000 80_000`	`Range: 20000 to 80000`
		`List of special numbers = [30103, 30203, 30403, 79397, 79697, 79997]`
		`Total number of special numbers = 48`
			`Execution time: 0.001014 seconds`

4	100_000 2_000_000	List of special numbers = [1003001, 1008001, 1022201, 1993991, 1995991, 1998991] Total number of special numbers = 190	Execution time: 0.014309 seconds
5	2_000_000 9_000_000	List of special numbers = [3001003, 3002003, 3007003, 7985897, 7987897, 7996997] Total number of special numbers = 327	Execution time: 0.045169 seconds
6	10_000_000 100_000_000	List of special numbers = [] Total number of special numbers = 0	Execution time: 0.037682 seconds
7	100_000_000 400_000_000	List of special numbers = [100030001, 100050001, 100060001, 399737993, 399767993, 399878993] Total number of special numbers = 2704	Execution time: 1.885647 seconds
8	1_100_000_000 15_000_000_000	List of special numbers = [10000500001, 10000900001, 10001610001, 14998289941, 14998589941, 14998689941] Total number of special numbers = 5474	Execution time: 28.169907 seconds
9	15_000_000_000 100_000_000_000	List of special numbers = [15001010051, 15002120051, 15002320051, 99998189999, 99998989999, 99999199999] Total number of special numbers = 36568	Execution time: 387.802233 seconds
10	1 1_000_000_000_000	List of special numbers = [2, 3, 5, 99998189999, 99998989999, 99999199999] Total number of special numbers = 47995	Execution time: 423.590071 seconds

Brazington, Ella 001305508:

#	Input	Output	Running time (s)
1	1 2000	There are 20 special numbers between 1 and 2000: The first three smallest special numbers: [2, 3, 5] The last three largest special numbers: [797, 919, 929]	Execution time: 0 minutes, 0.0010216236114501953 seconds
2	100 10_000	There are 15 special numbers between 100 and 10000: The first three smallest special numbers: [101, 131, 151] The last three largest special numbers: [797, 919, 929]	Execution time: 0 minutes, 0.0039906501770019531 seconds
3	20_000 80_000	There are 48 special numbers between 20000 and 80000: The first three smallest special numbers: [30103, 30203, 30403] The last three largest special numbers: [79397, 79697, 79997]	Execution time: 0 minutes, 0.0338959693908691406 seconds
4	100_000 2_000_000	There are 190 special numbers between 100000 and 2000000: The first three smallest special numbers: [1003001, 1008001, 1022201] The last three largest special numbers: [1993991, 1995991, 1998991]	Execution time: 0 minutes, 3.5495104789733886719 seconds
5	2_000_000 9_000_000	There are 327 special numbers between 2000000 and 9000000:	Execution time: 0 minutes, 27.083285808563232 4219 seconds

		The first three smallest special numbers: [3001003, 3002003, 3007003] The last three largest special numbers: [7985897, 7987897, 7996997]	
6	10_000_000 100_000_000	There are 0 special numbers between 10000000 and 100000000: The first three smallest special numbers: [] The last three largest special numbers: []	Execution time: 15 minutes, 18.813135862350463 8672 seconds
7	100_000_000 400_000_000	-	-
8	1_100_000_000 15_000_000_000	-	-
9	15_000_000_000 100_000_000_000	-	-
10	1 1_000_000_000_000	-	-

Ainhoa, Prada 001352985:

c	Input	Output	Running Time (s)
1	1, 2000	Total special numbers: 20 [3, 5, 7, 797, 919, 929]	0.0030106999911367893
2	100, 10000	Total special numbers: 15 [101, 131, 151, 797, 919, 929]	0.002355399978114292
3	20000, 80000	Total special numbers: 48 [30103, 30203, 30403, 79397, 79697, 79997]	0.3106100000150036

4	100000, 2000000	Total special numbers: 190 [1003001, 1008001, 1022201, 1993991, 1995991, 1998991]	5.789247299981071
5	2000000, 9000000	Total special numbers: 327 [3001003, 3002003, 3007003, 7985897, 7987897, 7996997]	44.96425869999803
6	10000000, 100000000	[]	9.100011084228754e-06
7	100000000, 400000000	-	Execution time: -
8	1100000000, 15000000000	-	Execution time: -
9	15000000000, 100000000000	-	Execution time: -
10	1, 1000000000000	-	Execution time: -

Lubis Aliyah 001315223 :

#	Input	Output	Running time (s)
1	1 2000	List of special numbers = [2, 3, 5, 797, 919, 929] Total number of special numbers = 20	Execution time: 0.0 seconds
2	100 10_000	List of special numbers = [101, 131, 151, 797, 919, 929] Total number of special numbers = 15	Execution time: 0.001007080078125 seconds
3	20_000 80_000	Range: 20000 to 80000 List of special numbers = [30103, 30203, 30403, 79397, 79697, 79997] Total number of special numbers = 48	Execution time: 0.0107879638671875 seconds
4	100_000 2_000_000	List of special numbers = [1003001, 1008001, 1022201, 1993991, 1995991, 1998991] Total number of special numbers = 190	Execution time: 0.32878947257995605 seconds
5	2_000_000 9_000_000	List of special numbers = [3001003, 3002003, 3007003, 7985897, 7987897, 7996997] Total number of special numbers = 327	Execution time: 1.1934237480163574 seconds
6	10_000_000 100_000_000	List of special numbers = [] Total number of special numbers = 0	Execution time: 14.761929273605347 seconds
7	100_000_000 400_000_000	List of special numbers = [100030001, 100050001, 100060001, 399737993, 399767993, 399878993] Total number of special numbers = 2704	Execution time: 49.90164566040039 seconds
8	1_100_000_000 15_000_000_000	-	Execution time: -
9	15_000_000_000 100_000_000_000	-	Execution time: -

10	1 1_000_000_000_000	-	Execution time: -

Petko, Georgiev 001318621:

c	Input	Output	Running Time (s)
1	1, 2000	Total special numbers: 20 [3, 5, 7, 797, 919, 929]	0.000618
2	100, 10000	Total special numbers: 15 [101, 131, 151, 797, 919, 929]	0.003139
3	20000, 80000	Total special numbers: 48 [30103, 30203, 30403, 79397, 79697, 79997]	0.019289
4	100000, 2000000	Total special numbers: 190 [1003001, 1008001, 1022201, 1993991, 1995991, 1998991]	0.348369
5	2000000, 9000000	Total special numbers: 327 [3001003, 3002003, 3007003, 7985897, 7987897, 7996997]	1.680772
6	10000000, 100000000	Total number of special numbers = 0 List of special numbers = []	17.595350

7	100000000, 400000000	Total number of special numbers = 2704 [List of special numbers = [100030001, 100050001, 100060001, 399737993, 399767993, 399878993]	154.285075
8	1100000000, 15000000000	-	Execution time: -
9	15000000000, 100000000000	-	Execution time: -
10	1, 1000000000000	-	Execution time: -

Test and analyse your solution!

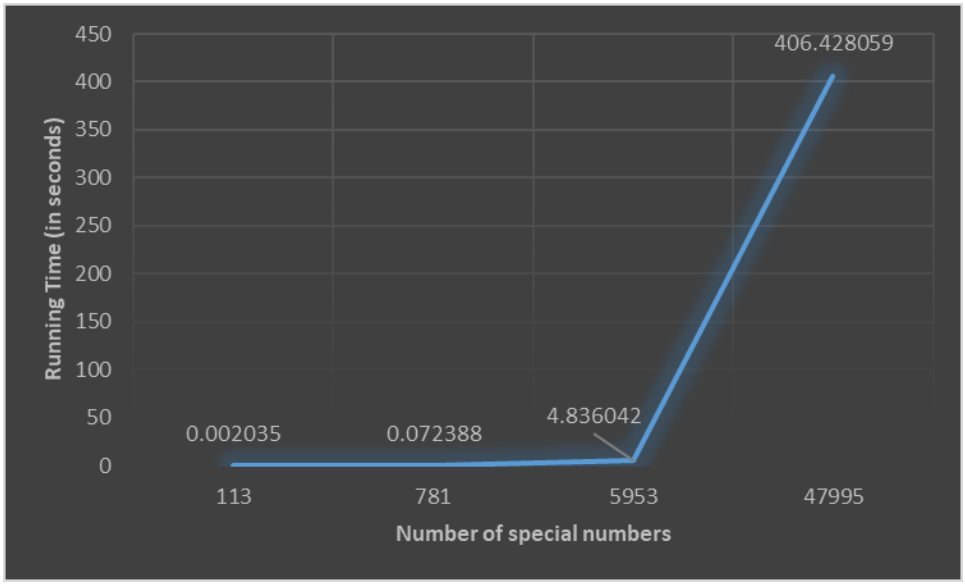
Md Abdul Raihan Tanzim 001341954:

Your test cases:

c	Input	Output	Justification	Student X results
1	-100 100	List of special numbers = [2, 3, 5, 7, 11] Total number of special numbers = 5	We observe quick and accurate results with almost instant output. However there isn't a specified error to indicate that a negative number is invalid.	Execution time: 0.000000 seconds

2	Any letter	ValueError: invalid literal for int() with base 10: 'q'	This test case indicates that the code does not accept string() type input only integers, int().	
3	0 1	List of special numbers = [] Total number of special numbers = 0	This test case indicates that the code accepts integers under the limit while maintaining correct output since there are no primes or palindromes below the number 2.	Execution time: 0.000000 seconds
4	100 100	List of special numbers = [] Total number of special numbers = 0	This test case indicates that the code can accept the same integer for both n and m but it accurately doesn’t output any special numbers.	Execution time: 0.000000 seconds
5	Decimal numbers	ValueError: invalid literal for int() with base 10: '0.2'	This test case indicates that the code doesn’t accept decimals since we require an int() input and not a boolean() input.	

Running time graphs:



Complexity analysis

is_prime Function:

Time Complexity: O(sqrt(n))

In order to determine whether a number is divisible, the function iterates from 2 to the square root of the number (inclusive), which results in an O(sqrt(n)) time complexity.

Space Complexity: O(1) Regardless of the size of the input, the function needs a fixed amount of space to store variables.

generate_palindromes Function:

Time Complexity: O(max_length * (end - start))

Palindromic numbers are produced by the function up to a maximum length that is provided inside the range [start, end]. The maximum length and range size determine how many palindromes are produced, giving rise to an O(max_length * (end - start)) time complexity.

Space Complexity: O(1):

Regardless of the size of the input, the function stores variables in constant space.

special_numbers Function:
Time Complexity: O(max_length * (end - start) * sqrt(n))

The function uses the is_prime function to iteratively check for primality through each created palindrome after calling generate_palindromes to produce palindromic numbers within the given range. Moreover, the temporal complexity of is_prime is O(sqrt(n)). Therefore, O(max_length * (end - start) * sqrt(n)) represents the entire time complexity.

Space Complexity: O(max_length * (end - start))
The number of created palindromic numbers, which is proportional to the product of the maximum length and the range size, is the primary determinant of the space complexity.

process_test_case Function:

Time Complexity: Relies on the function for special numbers

Space Complexity: Relies on the function for special numbers

The special_numbers function, which iterates through palindromic numbers within the given range and verifies for primality, is mostly responsible for the code's overall time complexity. The number of palindromic integers that are created inside the range mostly determines the space complexity.

Brazington, Ella 001305508

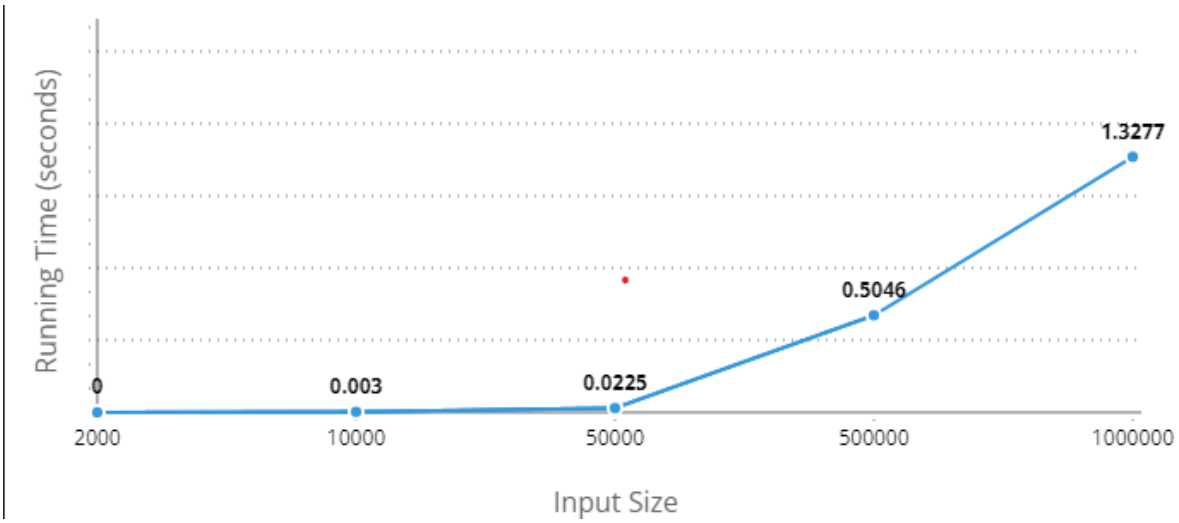
Your Test Cases:

c	Input	Output	Justification	Student X results

1	1, 2000	There are 20 special numbers between 1 and 2000: The first three smallest special numbers: [2, 3, 5] The last three largest special numbers: [797, 919, 929]	Examine the correlation between the size of the input and the algorithm's execution time.	Execution time: 0 minutes, 0.0000000000000000 seconds
2	1, 10000	There are 20 special numbers between 1 and 10000: The first three smallest special numbers: [2, 3, 5] The last three largest special numbers: [797, 919, 929]	Examine the correlation between the size of the input and the algorithm's execution time.	Execution time: 0 minutes, 0.0030148029327392578 seconds
3	1, 50000	There are 70 special numbers between 1 and 50000: The first three smallest special numbers: [2, 3, 5] The last three largest special numbers:	Examine the correlation between the size of the input and the algorithm's execution time.	Execution time: 0 minutes, 0.0225102901458740234 seconds

		[38183, 38783, 39293]		
4	1, 500000	There are 113 special numbers between 1 and 500000: The first three smallest special numbers: [2, 3, 5] The last three largest special numbers: [97879, 98389, 98689]	Examine the correlation between the size of the input and the algorithm's execution time.	Execution time: 0 minutes, 0.5046091079711914062 seconds
5	1, 1000000	There are 113 special numbers between 1 and 1000000: The first three smallest special numbers: [2, 3, 5] The last three largest special numbers: [97879, 98389, 98689]	Examine the correlation between the size of the input and the algorithm's execution time.	Execution time: 0 minutes, 1.3277976512908935547 seconds

Running Time Graph:



Complexity Analysis

`prime(num)` Function: This function determines if a given number `num` is prime. It iterates up to the square root of num to check divisibility. The time complexity of this function is O(sqrt(n)), where n is the input number.

`palindrome(num)` Function: This function checks if a given number `num` is a palindrome. It converts the number to a string and compares it with its reverse. The time complexity of this function is O(d), where d is the number of digits in the input number.

`SpecialNumbers(m, n)` Function: This function iterates through the range from `m` to n inclusive, and checks if each number is both prime and palindrome. Inside the loop, it calls `prime(num)` and `palindrome(num)` functions. The time complexity of this function depends on the number of iterations, which is O(n-m) in the worst case.

`main()` Function: This function primarily involves user input, function calls to `specialNumbers()`, and time tracking. It calls `specialNumbers()` once, and the time complexity of this function is dominated by the time complexity of `specialNumbers(m, n)` which is O(n-m).

Overall, the time complexity of the program is dominated by the `specialNumbers(m, n)` function, which is O(n-m) considering the worst-case scenario, where `m` is the smaller number and n is the larger number provided by the user.

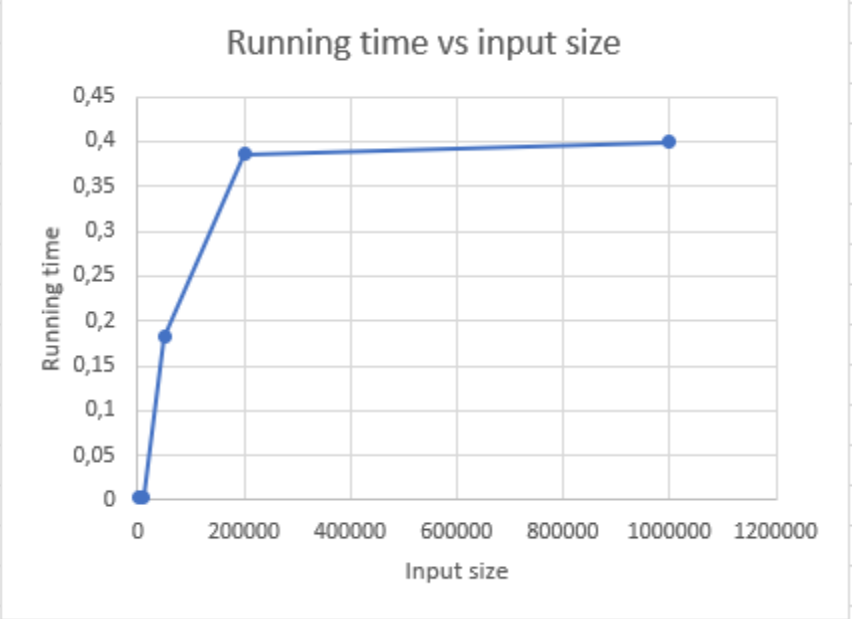
Ainhoa Prada 001352985:

Your test cases:

c	Input	Output	Justification	Student X results
1	1, 2000	Total special numbers: 20 [2, 3, 5, 797, 919, 929]	To check cumulative distribution and time complexity	Time: 0.002890000003390014s
2	1, 10000	Total special numbers: 20 [2, 3, 5, 797, 919, 929]	To check cumulative distribution, time complexity and prime palindromes distribution as numbers remain the same since last input.	Time: 0.004108799999812618s
3	1, 50000	Total special numbers: 70 [2, 3, 5, 38183, 38783, 39293]	To check cumulative distribution and time complexity.	Time: 0.18171340000117198s
4	1, 200000	Total special numbers: 113	To check cumulative distribution and time complexity.	Time: 0.38525490000029095

		[2, 3, 5, 97879, 98389, 98689]		
5	1, 1000000	Total special numbers: 113 [2, 3, 5, 97879, 98389, 98689]	To check cumulative distribution and that even digits don't have prime palindromes (no difference with previous input).	Time: 0.3992491999815684 s

Running time graphs:



Complexity analysis

Is_prime_and_palindromic(num): This functions first checks if a number is palindrome by reversing its digits, which has a time complexity of (O(log n)) where n is the number of digits in

the number. Then, it checks if the number is prime by iterating up to the square root of the number, which give us a time complexity of (O(sqrt(n)). Therefore, the time complexity of this function is (O(log n + sqrt(n)).

In this function, space complexity is (O(1)) because the amount of memory used does not increase with the size of the input, as we aren't using variables to store our output, instead we return a Boolean.

Prime_palindromes(m,n): This function skips ranges from m to n when they fall in a interval with where prime palindromes cannot exist. The loop also increments m by one in each operation. The time complexity in this case only depends on the input size, therefore it is (O(n)).

In this function, the space complexity is O(n), where n is the number of prime palindromes within the specified range. This is because we are storing the output in the special_numbers list, which can potentially grow linearly with the number of prime palindromes found.

Finally, the function find_special_numbers is only responsible for printing the result and displaying the execution time. Therefore, regarding time and space complexity, it is O(1), regardless of the input.

In conclusion, the overall solution will have a time complexity of (O(n + log n + sqrt(n))) and a space complexity of (O(n)).

Lubis Aliyah 001315223 :

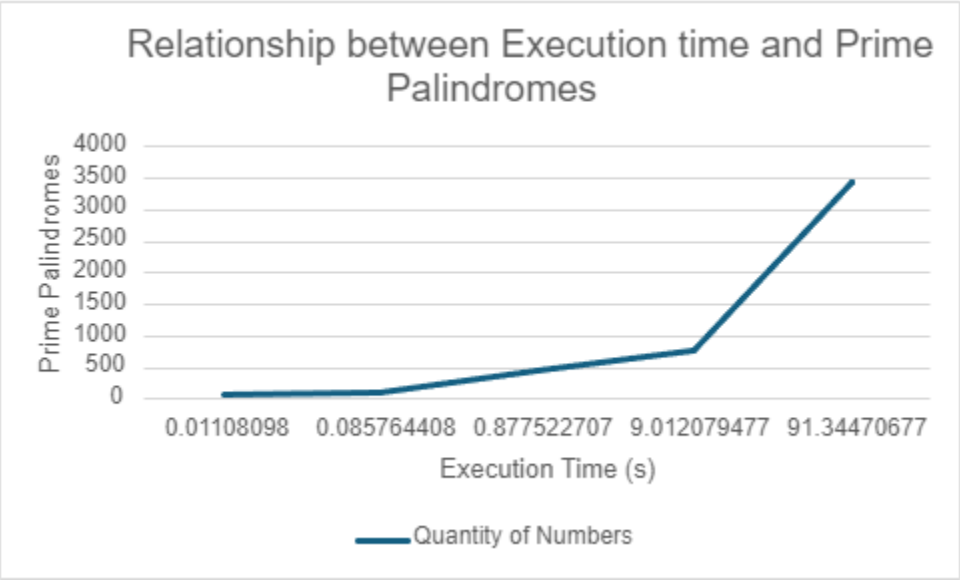
Your test cases:

c	Input	Output	Justification	Student X results
1	1, 100000	List of special numbers = [2, 3,	The input consists of different ranges of	Execution time:

		5] [97879, 98389, 98689] Total number of special numbers = 68	integers, while the output includes the list of special numbers within each range and the total number of special numbers found.	0.011080980300 90332 seconds
2	20 500_000	List of special numbers = [101, 131, 151 97879, 98389, 98689] Total number of special numbers = 108	The input sizes are increasing exponentially, from small ranges. This allows us to observe how the algorithm's performance scales with increasing input size.	Execution time: 0.085764408111 57227 seconds
3	200 5_000_000	List of special numbers = [313, 353, 373 3994993, 3997993, 39989930] Total number of special numbers = 465	The output includes the list of special numbers and the total count for each input range. This helps in verifying the correctness of the algorithm's results and allows for comparison between different test cases.	Execution time: 0.877522706985 4736 seconds
4	2_000 50_000_000	List of special numbers = [10301, 10501, 10601 9980899, 9981899, 9989899] Total number of special numbers = 761	The execution time for each test case is provided in seconds. As the input size increases, the execution time also increases, demonstrating the algorithm's scalability and efficiency (or lack thereof) for larger inputs.	Execution time: 9.012079477310 18 seconds
5	20_000 500_000_00 0	List of special numbers = [30103, 30203, 30403 399737993, 399767993, 399878993] Total number of special numbers = 3439	If the execution time grows much faster than the input size, it hints at a higher time complexity, potentially quadratic or worse. Conversely, if the execution time scales linearly or logarithmically with the input size, it suggests better	Execution time: 91.34470677375 793 seconds

			scalability and lower time complexity.	

Running time graphs



Complexity analysis

- **Input Reading**:
  - Reading two positive integers **m** and **n** takes constant time.

    Complexity: O(1)

- **Loop Iteration**:
  - The loop iterates over each integer in the range [m, n]. The size of this range determines the primary time complexity factor.
  - If the range is from **m** to **n**, the loop executes **n - m + 1** times.

    Complexity: O(n - m)

- **is_palindrome() Function**:
  - Converting an integer to a string takes time proportional to the number of digits in the integer. *Complexity: O(length_of_num)*
  - Reversing a string and comparing two strings also have linear time complexity proportional to the string's length. *Complexity: O(length_of_string)*

- **sympy.isprime() Function**:
  - The time complexity of determining whether a number is prime depends on the algorithm used within the **sympy** library.
  - For practical purposes, let's assume the implementation utilizes one of these efficient algorithms. The time complexity of these algorithms is typically logarithmic or sublinear in terms of the value of the number being tested.
- **Appending to special_numbers List**:
  - Appending to a list takes constant time on average. While occasionally resizing, the list may require O(n) time, this doesn't significantly affect overall time complexity. *Complexity: O(1)*
- **Printing Results**:
  - Printing the special numbers and other information takes time proportional to the number of special numbers found (**total_special_numbers**). *Complexity: O(total_special_numbers)*
- **Overall Time Complexity**:
  - The dominant factor is the loop iteration over the range [m, n], making it O(n - m). However, actual time complexity may vary based on the efficiency of functions like **is_palindrome()** and **sympy.isprime()**.

Petko Georgiev

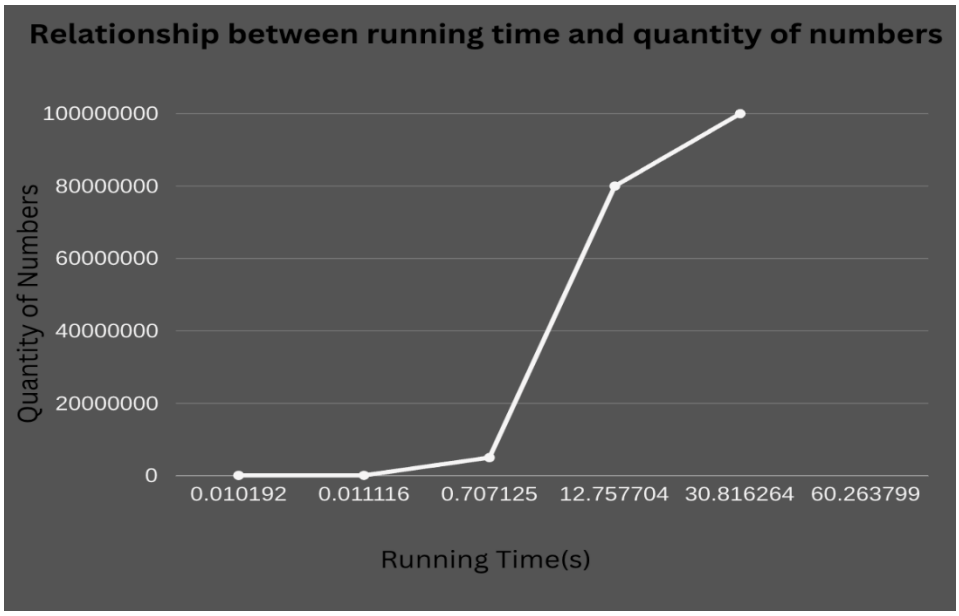
Your test cases:

c	Input	Output	Justification	Student X results
1	1, 100000	List of special numbers = [2, 3, 5] [97879, 98389, 98689] Total number of special numbers = 113		Execution time: 0.011116 seconds
2	1, 90000	List of special numbers = [2, 3, 5] [79397, 79697, 79997]		Execution time: 0.010192 seconds

		Total number of special numbers = 94		
3	100, 5000000	List of special numbers = [101, 131, 151] [3994993, 3997993, 3998993] Total number of special numbers = 470		Execution time: 0.707125 seconds
4	50000, 80000000	List of special numbers = [70207, 70507, 70607] [9980899, 9981899, 9989899] Total number of special numbers = 711		Execution time: 12.757704 seconds
5	500000, 100000000	List of special numbers = [1003001, 1008001, 1022201] [9980899, 9981899, 9989899] Total number of special numbers = 668		Execution time: 30.816264seconds

Running time graphs

1.



Complexity analysis

is_palindrome function:

•Time complexity: O(n), where n is the input number. It iterates each digit one by one and checks if they are the same both forward and backward.

• Space complexity: O(1), the function is used for a fixed amount of memory to store the input number and reverse the number.

generate_palindrome function:

• Time complexity: This function generates all palindromes where O(n) n is the input number by iterating over the range of numbers from 1 to 10^n and after these checks if it is palindromes. prime palindromes by leveraging (**sympy.isprime).**

• Space complexity: O(n) where stores all the palindromes numbers of the given length in a list.

sieve_of_eretosthenes function:

• Time complexity: O(n log log n) where n is the limit of the sieve. This function finds out all of the prime numbers by using the sieve of Eratosthenes algorithm. It has a time complexity O(n log log n) in the worst case.

• Space complexity: This function stores all the prime numbers in a given limit on a list.

find_special_numbers_sequential function:

• Time complexity: O(n^2) where n is the number of test cases. This function iterates over each test case where the given range m to n and checks if each number is prime but also have to be a palindrome. The total number in the range has an equal number of iterations. O(n) Overall time complexity for this function is O(n^2).

• Space complexity: O(n), as each test case's list of special numbers is stored by the function, and the list's length increases with the number of test cases.

Optimise solutions!

Solution 1-2:

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

Our team carefully assessed each team member's code for performance and compliance with the criteria of the problem. Petko's code was notable for its methodical approach to producing palindromic numbers and its algorithm to generate prime numbers. Alternatively, Ainhoa's method identified special numbers inside the given range by efficiently combining prime number validation

with palindrome production. These two codes were chosen because they were able to complete the test cases in the allotted hour.

We mostly worked independently on our individual programmes during the optimisation process, with Ainhoa taking the position of group leader and offering direction, help with troubleshooting, and helpful advice as needed. Our approach to optimization involved a combination of trial and error, research into efficient algorithms and techniques, and thorough troubleshooting of any bottlenecks or inefficiencies in the code. Every optimisation stage was thoroughly thought out and put into practice, and to guarantee better performance while upholding adherence to the criteria of the problem, testing and improvement were done continuously. All things considered, our combined efforts and individual contributions enabled our codes to be successfully optimised, producing effective solutions that satiated the predetermined standards.

Code 1 – Md Abdul Raihan Tanzim

Sieve of Eratosthenes Implementation: Compared to the Draft's individual prime checking method, the Final version employs a more optimised strategy by employing the Sieve of Eratosthenes algorithm to efficiently create prime numbers.

Prime Number Checking Optimisation: In the Final version, the number of iterations required for prime checking is decreased by employing a precomputed list of primes produced by the Sieve of Eratosthenes algorithm.

Better Palindrome Generation: While the two variants produce palindromic numbers in a comparable way, the Final version gains from the Sieve of Eratosthenes algorithm's efficiency in producing primes, which tangentially improves palindrome generation.

Error Handling: A “try” and “except” block together with a couple of conditional statements “if” have been used in the optimised version of the code to handle error handling, which catches “ValueError” exceptions that might occur during user input.

```
Enter the starting number: 3
Enter the ending number: 3
Error: Starting and ending numbers cannot be equal.

Process finished with exit code 0
```

```
Enter the starting number: 0
Enter the ending number: 1
Error: Numbers must be greater than or equal to 2.

Process finished with exit code 0
```

```
Enter the starting number: q
Error: invalid literal for int() with base 10: 'q'

Process finished with exit code 0
```

```
Enter the starting number: -100
Enter the ending number: 100
Error: Numbers must be greater than or equal to 2 and must not be negative.

Process finished with exit code 0
```

```
Enter the starting number: 3.5
Error: invalid literal for int() with base 10: '3.5'

Process finished with exit code 0
```

Results

c	Input	Output	Running Time (s)
1	1, 100_000	Total special numbers: 113 Prime Palindromes: [2, 3, 5] [97879, 98389, 98689]	Time: 0.0019970 s
2	1, 1_000_000	Total special numbers: 113	Time: 0.0045060 s

		Prime Palindromes: [2, 3, 5] [97879, 98389, 98689]	
3	1, 10_000_000	Total special numbers: 781 Prime Palindromes: [2, 3, 5] [9980899, 9981899, 9989899]	Time: 0.0210260 s
4	1, 100_000_000	Total special numbers: 781 Prime Palindromes: [2, 3, 5] [9980899, 9981899, 9989899]	Time: 0.0499930 s
5	1, 1_000_000_000	Total special numbers: 5953 Prime Palindromes: [2, 3, 5] [999676999, 999686999, 999727999]	Time: 0.8102180 s
6	1, 10_000_000_00 0	Total special numbers: 5953 Prime Palindromes: [2, 3, 5] [999676999, 999686999, 999727999]	Time: 1.1197760 s
7	1, 100_000_000_0 00	Total special numbers: 5474 Prime Palindromes: [10000500001, 10000900001, 10001610001] [14998289941, 14998589941, 14998689941]	Time: 64.374890 s
8	1, 1_000_000_000 _000	Total special numbers: 47995 Prime Palindromes: [2, 3, 5] [99998189999, 99998989999, 99999199999]	Time: 68.4293890 s

Code 2 - Prada Tello, Ainhoa:

In the second solution, while the primes' function remains in its original form, the palindrome function has been optimized. This function now generates palindromes by first generating the first half and then duplicating it on the other side. Additionally, in this solution, we skip the ranges where there are no prime palindromes by skipping when the digits are even numbers. The result is an array containing all the palindrome numbers from 1 up to the greatest given number. Now, we simply need to search for prime numbers within the palindrome array and return the special numbers.

The reason for choosing this approach is because the use of plain Python arithmetic is very slow, making it difficult to check a trillion numbers in a loop within an hour. After weeks of programming, research, and documentation, Petko came up with the idea of generating palindromes instead of checking them. The code could still be improved by optimizing the prime function and using external libraries such as Numba and its file compiler.

c	Input	Output	Running Time (s)
1	1, 100_000	Total special numbers: 113 Prime Palindromes: [2, 3, 5] [97879, 98389, 98689]	Time: 0.00403509999159723 5
2	1, 1_000_000	Total special numbers: 113 Prime Palindromes: [2, 3, 5] [97879, 98389, 98689]	Time: 0.01387389999581500 9
3	1, 10_000_000	Total special numbers: 781 Prime Palindromes: [2, 3, 5]	Time: 0.2676627000037115

		[9980899, 9981899, 9989899]	
4	1, 100_000_000	Total special numbers: 781 Prime Palindromes: [2, 3, 5] [9980899, 9981899, 9989899]	Time: 0.4635567999794148
5	1, 1_000_000_000	Total special numbers: 5953 Prime Palindromes: [2, 3, 5] [999676999, 999686999, 999727999]	Time: 23.5558052000124
6	1, 10_000_000_000	Total special numbers: 5953 Prime Palindromes: [2, 3, 5] [999676999, 999686999, 999727999]	Time: 22.538999100012006
7	1, 100_000_000_000	Total special numbers: 5474 Prime Palindromes: [10000500001, 10000900001, 10001610001] [14998289941, 14998589941, 14998689941]	Time: 111.44282350002322
8	1, 1_000_000_000_000	Total special numbers: 47995 Prime Palindromes: [2, 3, 5] [99998189999, 99998989999, 99999199999]	Time: 1817.4956098000112

Compare the performance!

Time complexities and big-O notations

Code 1:

Up to a specified limit, the **Sieve of Eratosthenes function** produces all prime numbers in an efficient manner. It functions by repeatedly identifying as composite the multiples of every prime number, so excluding non-prime integers from consideration. This algorithm's **time complexity is commonly estimated to be O(n log log n)**, where n is the upper bound. This is due to the algorithm pointing out multiples of each prime number as iterates through the numbers up to the limit's square root. For every number being checked, **the is_prime function**—which determines whether the given number is prime—contributes an extra **O(sqrt(n)) complexity**. The **function generate_palindromes**, which produces palindromic numbers within a range, usually has a **time complexity that is linear with the range's length**.

With all these factors combined, the Sieve of Eratosthenes method dominates the total time complexity of the special_numbers function, which makes use of these components. This leads to a time complexity of around **O(n log log n)**, where n is the upper limit of the range.

O(n log log n)

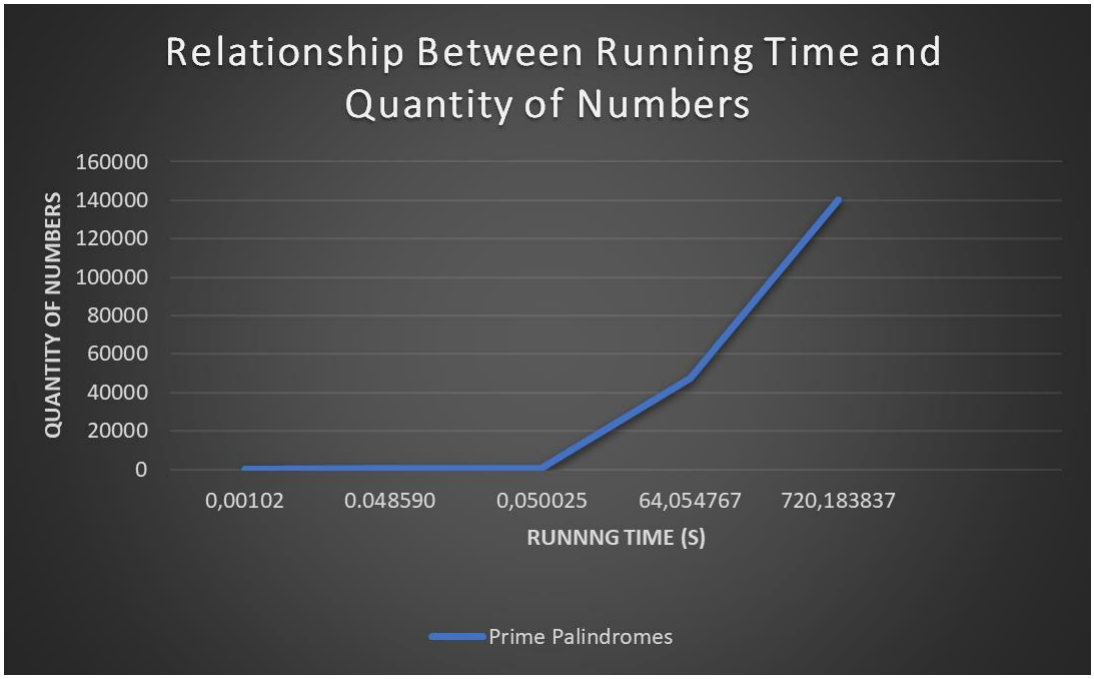
Code 2:

Within a specified range, this program generates prime palindrome integers. Within the given range, the **generate_palindromes function** produces palindromic numbers. The production of palindromic numbers, which is usually **linear in relation to range length**, and the **range's length determine the system's temporal complexity**. The **is_prime function** iterates up to the integer's square root to determine if it is prime. For each number it tests, it adds an **O(sqrt(n)) complexity**. To find prime palindromes, the **find_prime_palindromes function** filters the generated palindromic numbers. The number of palindromic numbers created determines how complex it is in terms of time. Although the exact time complexity of this program is unknown, it is generally thought to be **O(n sqrt(n)), where n is the range's length.**

O(n sqrt(n))

Running time graphs

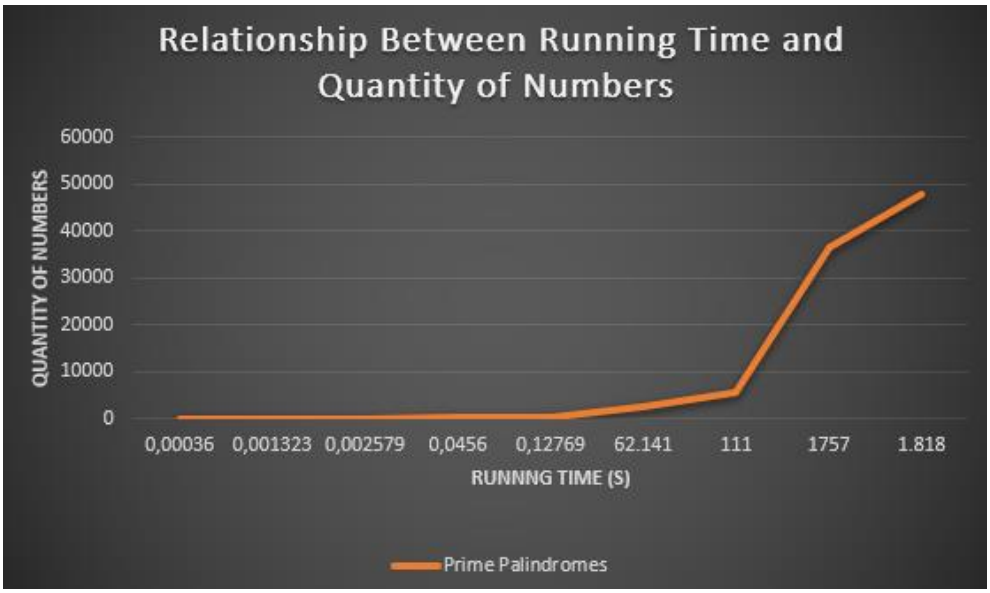
Code 1:



Big O Notation = **O(n log log n)**

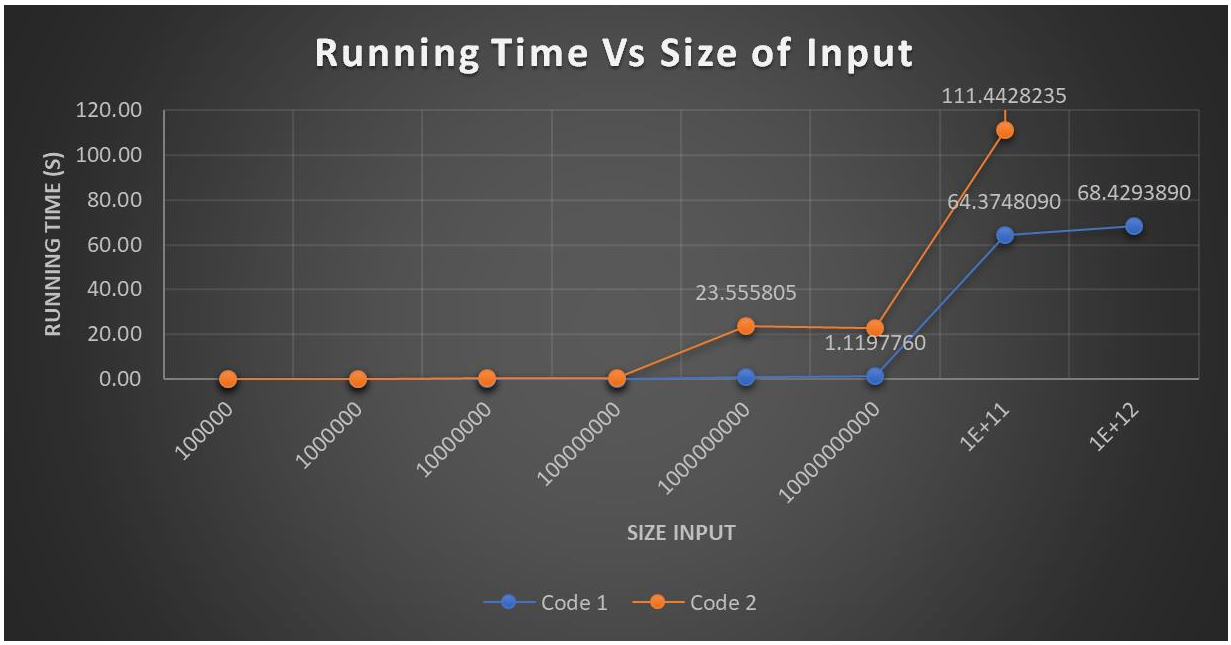
This notation denotes a time complexity in which the algorithm's runtime increases proportionately to n times the logarithm of n's logarithm. This graph shows that as the number increases, the runtime climbs gradually.

Code 2:



Big O Notation = **O(n sqrt(n))**

An algorithmic time complexity indicated by the notation O(n sqrt(n)) grows in runtime proportional to n multiplied by the square root of n. Although linearithmic (O(n log n)) algorithms are more efficient than O(n sqrt(n)) algorithms, for moderate-sized inputs, they can still be useful and effective, nevertheless. They are appropriate for many different applications because they achieve a balance between complexity and efficiency. The graph shows that the runtime increases with the input size (n), although not as quickly as the first code did.



Reflecting on teamwork!

Team members:

Member	Name	ID	Contribution %
1	Georgiev, Petko	001318621	100%
2	Brazington, Ella	001305508	100%
3	Prada Tello, Ainhoa	001352985	100%
4	Tanzim, Md Abdul Raihan	001341954	100%
5	Lubis, Aliyah	001315223	100%

Contribution mark

Name	ID	Task 1 (30%)	Task 2 (20%)	Task 3 (20%)	Task 4 (15%)	Task 5 (15%)	**Contribution mark (100%)**
Prada Tello, Ainhoa (Group leader)	001352985	20%	20%	20%	20%	20%	100%
Brazington, Ella	001305508	20%	20%	20%	20%	20%	100%
Georgiev, Petko	001318621	20%	20%	20%	20%	20%	100%
Lubis, Aliyah	001315223	20%	20%	20%	20%	20%	100%
Tanzim, Md Abdul Raihan	001341954	20%	20%	20%	20%	20%	100%

Limitations and Potential discussion

Participation/ engagement: Working collaboratively, our group found task engagement seamless due to our shared document, established since the project's inception. This document served as a repository of our collective progress and insights, facilitating easy access for all team members and fostering active involvement in the task. Additionally, our weekly planning sessions ensured continuity and focus by delineating clear objectives for each member to tackle. This proactive approach kept us aligned and empowered each team member to make meaningful contributions consistently.

Collaboration: Collaboration among our team was seamless thanks to our shared document and group chat, which provided convenient platforms for exchanging ideas and offering support

round the clock. Additionally, regular communication during module labs further strengthened our cohesion and synergy. Leveraging these channels, we engaged in brainstorming sessions, ensuring that each member's input was valued and integrated effectively, thus maximizing our collective effort towards achieving excellence.

Leadership: Ainhoa assumed the role of group leader, prominently showcasing her leadership skills by initiating numerous conversations in both the labs and the group chat. However, in her absence or unavailability, we all had the opportunity to demonstrate leadership by dividing tasks and responsibilities among ourselves. This ensured that the group continued to function smoothly and efficiently, highlighting our collective ability to adapt and take initiative when needed.

Problem-solving skills: Within the initial weeks of embarking on the project, each member of our team started showcasing their problem-solving. It quickly became evident that the task posed more challenges than we initially anticipated. This realization sparked a significant increase in communication within our group, as we leaned on each other for ideas and solutions to meet the project requirements. Our greatest strength in problem-solving lay in our ability to rely on one another, fostering a collaborative environment where every member's input was valued and utilized effectively.

Creativity and Innovation: In the initial weeks, each team member showcased their creativity while grappling with the task's requirements. As we pooled our ideas together, our collective brainstorming sessions led to even more innovative solutions. This collaborative process not only enhanced the quality of our ideas but also fostered a sense of camaraderie and shared ownership of the project's success. However, there were challenges for some of us when it came to our code. To address these difficulties, we adopted an innovative approach, exploring new solutions and techniques to overcome obstacles and improve the quality of our work.

Communication dynamic: During some weeks, communication posed a challenge, particularly if certain members were absent during lab sessions. Nevertheless, this is where our group chat came into play, serving as a reliable means of communication. Over time, we noticed an improvement in our communication efforts, with interactions becoming more frequent. As the deadline drew nearer, our communication intensified, with daily exchanges becoming the norm as we worked together to meet project milestones.

Weekly journal

	Task note	Status
Week 1: 5th – 11th		
Prada Tello, Ainhoa (Group leader)	Research of the best data structures, python optimization and comparing the best algorithms for both main functions.	Done
Brazington, Ella	The first basic sample of code initiated couldn't complete all test cases within the hour limit. While it managed the initial ones, it struggled with slowness and failed to handle the remaining ones effectively.	Done
Tanzim, Md Abdul Raihan Tanzim	Researched the project requirements and established a plan for the coding tasks. Gathered necessary resources and set up the development environment.	Done
Lubis, Aliyah	I thoroughly read and analysed the requirements necessary for addressing the problem and conducted extensive research to understand various methodologies and approaches for initiating the problem-solving process.	Done
Georgiev, Petko	On the first I tried to learn different types of algorithms and how these works. Then I started with the initial code and make an observation to see what I should do to gone through all the test cases.	Done
Week 2: 12th to 18th		
Prada Tello, Ainhoa (Group leader)	The first solution has been coded, the functions have multiple loops that make the solution very slow. Research on external libraries and other techniques.	Done
Brazington, Ella	Despite daily efforts to edit the code, it continues to lack efficiency. However, there has been improvement over time.	Done
Tanzim, Md Abdul Raihan	Implemented the initial version of the code, focusing on functionality over optimization. Ran test cases to identify any bugs or errors.	Done
Lubis, Aliyah	I started coding and tested various approaches to address the problem, but I'm	Done

	still finding it challenging to ensure that the code executes within one hour.	
Georgiev, Petko	Then I tried with different types of sieve algorithms to make it quicker but on every attempt, it was not going faster than earlier. After this I thought that I should use python library. Then I was making improvement.	Done
Week 3: 19th to 25th		
Prada Tello, Ainhoa	Multiple solutions have been coded using different algorithms for primes and palindromes. Finally, I have developed a function that combines both to minimize the number of loops, focusing on the outer loop with the range of numbers.	Done
Brazington, Ella	I've been following a similar routine to last week, encountering some challenges along the way. To overcome these hurdles, I've been engaging in discussions with my peers and group members, actively seeking their input and incorporating their suggestions into my work.	Done
Tanzim, Md Abdul Raihan	Reviewed the code and identified areas for optimization. Researched algorithms and techniques to improve performance. Made necessary revisions to the code.	Done
Lubis, Aliyah	I've discovered the most effective method for identifying the special numbers and incorporated it into my code. While it still doesn't perform within an hour, this approach runs more efficiently compared to the others I've experimented with.	Done
Georgiev, Petko	I continued my research and try with changing the functions for prime and palindrome. I also used array and list to see the difference of the speed. Then I used a numerical method for palindrome.	Done
Week 4: 26th to 3rd		
Prada Tello, Ainhoa	I research on mathematical rules and the distribution of prime palindromes. The outer loop will be implemented as a while loop to facilitate skipping ranges.	Done

	Meanwhile, measuring and comparing each change.	
Brazington, Ella	This week has proven to be particularly challenging for me. I found myself revisiting the lectures and conducting extensive research in an effort to find solutions. Despite my efforts, my code still falls short of completing the final few test runs within the designated hour timeframe.	Done
Tanzim, Md Abdul Raihan	Tested the optimized code with sample inputs to ensure correctness. Continued refining the code based on feedback and additional testing.	Done
Lubis, Aliyah	Despite not achieving the desired runtime of under an hour, I shared my code with my group members and informed them of my efforts to meet that time constraint.	Done
Georgiev, Petko	After all this research I tried a lot, but I couldn't make it through to all the test cases. I shared my code to my group members and said what I have done it.	Done
Week 5: 4th to 10th		
Prada Tello, Ainhoa	Focusing on optimizing the two chosen solutions, Petko found a faster algorithm to optimize the code by generating the palindromes instead of looping. Finally, tests 9 and 10 were passed.	Done
Brazington, Ella	I'm currently in the process of arranging a call with my group to select the two best options. Once chosen, we'll have a week to refine and improve them to the best of our abilities. In the meantime, I'm persistently working on my code to ensure it meets the requirements. Feeling a bit stuck, I decided to make progress on the report while I continue to tackle the challenges with my code.	Done
Tanzim, Md Abdul Raihan	Integrated user input functionality and error handling. Tested the code with different input scenarios to verify robustness.	Done
Lubis, Aliyah	I engaged in a discussion with my groupmates regarding potential	Done

	enhancements for my code, which I then implemented. Despite these adjustments, the desired time limit remained elusive. Subsequently, I began writing the report.	
Georgiev, Petko	Then we choose the two best solutions from our groupmates. But I continuously tried to improve mine, also continued my research to help them with the two best solutions.	Done
Week 6: 11th to 17th		
Prada Tello, Ainhoa	I contacted our tutor and realized that some libraries I was using weren't accepted, so I had to modify the code, which made it slower. After that, I focused mainly on working on the report.	
Brazington, Ella	Even though we've already selected the two best codes, I persevered in my efforts to optimize mine to run within the hour limit during my downtime.	Done
Tanzim, Md Abdul Raihan	Documented the code and added comments for clarity. Conducted final testing to confirm that all requirements were met.	In Progress
Lubis, Aliyah	I concentrated on drafting the report to finish it on time, paying close attention to the big o notations and time complexity on the two final codes that we selected.	Done
Georgiev, Petko	I gave my focus on report and added comment on the code of mine. Then focusing to do the big o notations and time complexity for the final two code.	Done
Week 7 : 18th to 19th (due date)		
Prada Tello, Ainhoa	The project is now finished, and the report has been completed.	
Brazington, Ella	Finished the project and submitted. Reflected on what I have learnt.	Done
Tanzim, Md Abdul Raihan	Finalized the project and submitted the code. Reflected on the overall process and identified lessons learned for future projects.	Done

Lubis, Aliyah	Finalized the report, reread the report thoroughly to make sure it is ready for submission.	Done
Georgiev, Petko	The project is finished, and report is completed. After all of these I learned a lot of things on algorithms and waiting to explore more.	Done

Reference's

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

Md Abdul Raihan Tanzim

2. Irkl1_Irkl1_ (2021) Python function to check if number is prime, Stack Overflow. Available at: https://stackoverflow.com/questions/65728113/python-function-to-check-if-number-is-prime

3. Programming, L. (2017) *Algorithms in python: Binary search*, *YouTube*. Available at: https://www.youtube.com/watch?v=zeULw-a7Mw8&list=PL5tcWHG-UPH1K7oTJgIbWy6rCMc8-8Lfm

4. Alexandre, J. *et al.* (2016) *How to generate prime palindromes in python 3*, *Stack Overflow*. Available at: https://stackoverflow.com/questions/36263254/how-to-generate-prime-palindromes-in-python-3
5. ChiChi and TrebledJTrebledJ 8 (2018) Prime numbers, sieve of Erasthosthenes , python, Stack Overflow. Available at: https://stackoverflow.com/questions/53454250/prime-numbers-sieve-of-erasthosthenes-python

Brazington, Ella

1. geeksforgeeks (2018). *Analysis of Algorithms | Big-O analysis - GeeksforGeeks*. GeeksforGeeks. Available at: https://www.geeksforgeeks.org/analysis-algorithms-big-o-analysis/.

2. C.B. (2014) Palindromic prime number in Python, stack overflow. Available at https://stackoverflow.com/questions/22699625/palindromic-prime-number-in-python.

3. GeeksforGeeks. (2016). *Palindromic Primes*. Available at: https://www.geeksforgeeks.org/palindromic-primes/.

Ainhoa Prada Tello:

1. Anon, (2005). *Primes, Palindromes, and Pyramids*. [online] Available at: https://www.sciencenews.org/article/primes-palindromes-and-pyramids#:~:text=In%20decimal%20notation%2C%20the%20sequence [Accessed 18 Mar. 2024].

2. GeeksforGeeks. (2016). *Palindromic Primes*. [online] Available at: https://www.geeksforgeeks.org/palindromic-primes/.

3. Banks, W.D., Hart, D.N. and Sakata, M. (2004). Almost All Palindromes Are Composite. *Mathematical Research Letters*, 11(6), pp.853–868. doi:https://doi.org/10.4310/mrl.2004.v11.n6.a10.

Georgiev, Petko

1. geeksforgeeks. Sieve of Eratosthenes https://www.geeksforgeeks.org/sieve-of-eratosthenes/

2. Time Complexity https://blog.devgenius.io/python-time-complexities-1988ec5d16d9

3. reddit. Prime palindrome https://www.reddit.com/r/learnpython/comments/1bd3jse/finding_palindromic_primes/?rdt=46919

4. MIT opencourseware. Big O Notation https://ocw.mit.edu/courses/6-00sc-introduction-to-computer-science-and-programming-spring-2011/40c9c46fcf6cd68e91a011ea04feb1d9_MIT6_00SCS11_rec04.pdf

5. stackoverflow. Palindrome Generator https://stackoverflow.com/questions/17435448/palindrome-generator

Lubis, Aliyah:

1. geeksforgeeks (2018). *Analysis of Algorithms | Big-O analysis - GeeksforGeeks*. [online] GeeksforGeeks. Available at: https://www.geeksforgeeks.org/analysis-algorithms-big-o-analysis/.

2. Olawanle, J. (2022). *Big O Cheat Sheet – Time Complexity Chart*. [online] freeCodeCamp.org. Available at: https://www.freecodecamp.org/news/big-o-cheat-sheet-time-complexity-chart/.

3. upGrad blog. (n.d.). *How To Check Palindrome Number in Python?* [online] Available at: https://www.upgrad.com/blog/palindrome-using-python/.

4. GeeksforGeeks. (2019). *Python | sympy.prime() method*. [online] Available at: https://www.geeksforgeeks.org/python-sympy-prime-method/.

5. Stack Overflow. (n.d.). *How do I measure elapsed time in Python?* [online] Available at: https://stackoverflow.com/questions/7370801/how-do-i-measure-elapsed-time-in-python.

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.

Md Abdul Raihan Tanzim 001341954:

Draft Code:

```
# Petko Georgiev 01318621, Unoptimized version of the code aka Draft.

import time

def is_prime(num):
    # Check if 'num' is prime by checking each number for primality individually
    if num < 2:
        return False
    for i in range(2, int(num**0.5) + 1):
        if num % i == 0:
            return False
    return True



def generate_palindromes(max_length):
    # Generate palindromic numbers with specified maximum length
    for length in range(1, max_length + 1):
        half_length = (length + 1) // 2
        start = 10 ** (half_length - 1)
        end = 10 ** half_length

        for i in range(start, end):
```

```
            str_i = str(i)
            # Create palindromes by reversing the first half
            yield int(str_i + str_i[-(length % 2)-1::-1])



def special_numbers(start, end):
    max_length = len(str(end))
    # Filter palindromic numbers within the specified range that are also prime
    special_numbers_list = [p for p in generate_palindromes(max_length) if start <= p <=
end and is_prime(p)]

    total_special_numbers = len(special_numbers_list)

    # Return the required format of special numbers
    if total_special_numbers <= 6:
        return special_numbers_list, [], total_special_numbers
    else:
        return special_numbers_list[:3], special_numbers_list[-3:], total_special_numbers



def process_test_case():
    m = int(input("Enter the starting number: "))
    n = int(input("Enter the ending number: "))

    start_time = time.time()
    # Find and display special numbers for the given range
    result_start, result_end, total_special_numbers = special_numbers(m, n)
    end_time = time.time()

    print(f"\nRange: {m} to {n}")
    print(f"List of special numbers = {result_start + result_end}")
    print(f"Total number of special numbers = {total_special_numbers}")
    print(f"Execution time: {end_time - start_time:.6f} seconds")



if __name__ == "__main__":
    process_test_case()
```

Optimized Code:

```python
# Petko Georgiev 001318621

import time



def sieve_of_eratosthenes(limit):

# Sieve of Eratosthenes for generating prime numbers up to 'limit'
    primes = [True] * (limit + 1)
    primes[0] = primes[1] = False

    for num in range(2, int(limit**0.5) + 1):
        if primes[num]:
            # Mark multiples of 'num' as non-prime
            primes[num*num: limit+1: num] = [False] * ((limit - num*num)//num + 1)

    return [num for num in range(2, limit + 1) if primes[num]]



def is_prime(num, primes):
    # Check if 'num' is prime using a list of primes up to its square root
    if num < 2:
        return False
    for prime in primes:
        if prime * prime > num:
            break
        if num % prime == 0:
            return False
    return True



def generate_palindromes(max_length):
    # Generate palindromic numbers with specified maximum length
    for length in range(1, max_length + 1):
        half_length = (length + 1) // 2
        start = 10 ** (half_length - 1)
        end = 10 ** half_length

        for i in range(start, end):
            str_i = str(i)
            # Create palindromes by reversing the first half
            yield int(str_i + str_i[-(length % 2)-1::-1])



def special_numbers(start, end):
```

```
    max_length = len(str(end))
    # Generate prime numbers up to the square root of 'end'
    primes = sieve_of_eratosthenes(int(end**0.5) + 1)
    # Filter palindromic numbers within the specified range that are also prime
    special_numbers_list = [p for p in generate_palindromes(max_length) if start <= p <=
end and is_prime(p, primes)]

    total_special_numbers = len(special_numbers_list)

    # Return the required format of special numbers
    if total_special_numbers <= 6:
        return special_numbers_list, [], total_special_numbers
    else:
        return special_numbers_list[:3], special_numbers_list[-3:], total_special_numbers



def process_test_case():
    try:
        m = int(input("Enter the starting number: "))
        n = int(input("Enter the ending number: "))

        if m < 2 or n < 2:
            raise ValueError("Numbers must be greater than or equal to 2 and must not be
negative.")

        if m == n:
            raise ValueError("Starting and ending numbers cannot be equal.")

       start_time = time.time()
        # Find and display special numbers for the given range
        result_start, result_end, total_special_numbers = special_numbers(m, n)
        end_time = time.time()

        print(f"\nRange: {m} to {n}")
        print(f"List of special numbers = {result_start + result_end}")
        print(f"Total number of special numbers = {total_special_numbers}")
        print(f"Execution time: {end_time - start_time:.6f} seconds")

    except ValueError as ve:
        print(f"Error: {ve}")



if __name__ == "__main__":
    process_test_case()
```

Brazington, Ella 001305508

```
# Brazington, Ella 001305508
import time

def main():
    # Prompt the user to input the range of numbers
    m = int(input("Enter a smaller number (m): "))
    n = int(input("Enter a larger number (n): "))

    start_time = time.time()  # Record the start time

    # Find special numbers within the specified range
    special_numbers = specialNumbers(m, n)
    num_special = len(special_numbers)  # Count the number of special numbers

    end_time = time.time()  # Record the end time
    execution_time = end_time - start_time  # Calculate the execution time

    # Display the execution time
    display_execution_time(execution_time)

    # Display the count of special numbers and list them
    print(f"\nThere are {num_special} special numbers between {m} and {n}:")
    print("The first three smallest special numbers:", special_numbers[:3])
    print("The last three largest special numbers:", special_numbers[-3:])

def prime(num):
    # Check if a number is prime from within the given range
    if num <= 1:
        return False
    if num <= 3:
        return True
    if num % 2 == 0 or num % 3 == 0:
        return False
    i = 5
    while i * i <= num:
        if num % i == 0 or num % (i + 2) == 0:
            return False
        i += 6
    return True

def palindrome(num):
    # Check if a number is a palindrome from within the given range
    return str(num) == str(num)[::-1]

def specialNumbers(m, n):
```

```
    # Find special numbers within the given range
    special_numbers = []
    for num in range(m, n + 1):
        if prime(num) and palindrome(num):
            special_numbers.append(num)
    return special_numbers

def display_execution_time(execution_time):
    # Display the execution time with more digits
    minutes = int(execution_time // 60)
    seconds = execution_time % 60
    print(f"Execution time: {minutes} minutes, {seconds:.19f} seconds")


if __name__ == "__main__":
    main()
```

Ainhoa Prada 001352985:

```
from time import perf_counter


def is_prime_and_palindromic(num):
    """ Checks if a number is a palindrome, and then checks if the number is prime.
        Checks if the number is a palindrome by reversing its digits and comparing with the original number.
        Checks if the number is prime by checking if its modulo with the numbers less than or equal to the square root of the number is not equal to 0.

    Parameters:
        num (int): The integer to be checked.

    Returns:
        bool: True if the number is both a palindrome and prime, False otherwise.
```

```
"""

    temp = num
    reverse = 0
    while temp > 0:
        last_digit = temp % 10
        reverse = reverse * 10 + last_digit
        temp //= 10

    if reverse != num:
        return False

    if num < 2 or (num > 2 and num % 2 == 0):
        return False

    for i in range(3, int(num**0.5) + 1, 2):
        if num % i == 0:
            return False

    return True


def prime_palindromes(m, n):
    """Will skip the ranges where the prime palindromes don't exist and will
append to special numbers those that are prime and palindromes.

        Integer:
        n (int): The bottom number for the range
        m (int): Top number for the range
```

```
    Returns:
    special_numbers (lsit): a list containing the prime palindromes """
special_numbers = []
while m <= n:
    if (1000 <= m <= 10000):
        m = 10000
    elif (100000 <= m <= 1000000):
        m = 1000000
    elif (10000000 <= m <= 100000000):
        m = 100000000
    elif (1000000000 <= m <= 10000000000):
        m = 10000000000
    elif (100000000000 <= m <= 1000000000000):
        m = 1000000000000
    else:
        if is_prime_and_palindromic(m):
            special_numbers.append(m)
    m += 1


    if (1000 <= n <= 10000):
        n = 10000
    elif (100000 <= n <= 1000000):
        n = 1000000
    elif (10000000 <= n <= 100000000):
        n = 100000000
    elif (1000000000 <= n <= 10000000000):
        n = 10000000000
    elif (100000000000 <= n <= 1000000000000):
        n = 1000000000000
```

```
    return special_numbers

def find_special_numbers(m,n):
    print(f"Prime palindromes from {m} to {n}: ")
    start_time = perf_counter()
    special_numbers = prime_palindromes(m, n)
    end_time = perf_counter()
    print(f'Total special numbers: {len(special_numbers)}')
    if len(special_numbers) <= 6:
        print(special_numbers)
    else:
        print(special_numbers[:3] +  special_numbers[-3:])
    print(f'Time: {end_time - start_time} s')

# Inputs
m = int(input("Enter a number: "))
n = int(input("Enter a number: "))

find_special_numbers(m, n)
```

Optimized version:

```
from time import perf_counter

def generate_palindromes(start, end):
    """Generate palindromic numbers within a range.
       It will skip ranges where prime palindromes don't exist.
```

```
        Parameters:
            start (int): The start of the range.
            end (int): The end of the range.

        Returns:
            list: A list of palindromic numbers within the specified range.
        """
    palindromes = []
    max_length = len(str(end))
    for length in range(1, max_length + 1):
        # Skip lengths 4, 6, 8, and 10 (no prime palindromes with those digits) -
-> pyramid rule
        if length in (4, 6, 8, 10):
            continue

        half_length = (length + 1) // 2
        start_digit = 10 ** (half_length - 1)
        end_digit = min(10 ** half_length, end)

        for i in range(start_digit, end_digit):
            str_i = str(i)
            num = int(str_i + str_i[-(length % 2) - 1::-1])
            if start <= num <= end:
                palindromes.append(num)
    return palindromes


def is_prime(num):
    """ Checks if a number is prime.
        Checks if the modulo of the number dvided by the bottom numbers of the
square root to 2 is equal zero.
```

```
        Returns a boolean """
    if num < 2:
        return False
    for i in range(2, int(num ** 0.5) + 1):
        if num % i == 0:
            return False
    return True

def find_prime_palindromes(palindromes):
    """ Filter the palindrome array and return the prime numbers """
    prime_palindromes = [num for num in palindromes if is_prime(num)]
    return prime_palindromes

def print_prime_palindromes(prime_palindromes):
    print(f'Total special numbers: {len(prime_palindromes)}')
    print("Prime Palindromes:")
    if len(prime_palindromes) <= 6:
        print(prime_palindromes)
    else:
        print(prime_palindromes[:3])
        print(prime_palindromes[-3:])

# Inputs
first_num = int(input("Enter the first number: "))
last_num = int(input("Enter the second number: "))

start = perf_counter()

# Generate palindromic numbers within the range of the input numbers
```

```
palindromes = generate_palindromes(first_num, last_num)

# Find prime palindromes
prime_palindromes = find_prime_palindromes(palindromes)

end = perf_counter()

# Print prime palindromes
print_prime_palindromes(prime_palindromes)
print(f'Time: {end - start}')
```

Lubis, Aliyah 001315223:

```
import time
import sympy


def is_palindrome(num):
    num_str = str(num)
    return num_str == num_str[::-1]


def main():
    m, n = map(int, input("Please enter two positive integers m and n (m < n):
").split())

    if m >= n:
        print("m should be smaller than n.")
```

```
        return


    start_time = time.time()  # Start timing


    special_numbers = []


    for num in range(m, n+1):
        # Skip even numbers and numbers ending with 5
        if num % 2 == 0 or str(num)[-1] == '5':
            continue


        if is_palindrome(num) and sympy.isprime(num):
            special_numbers.append(num)


    end_time = time.time()  # End timing


    total_special_numbers = len(special_numbers)


    print("List of special numbers:", ', '.join(map(str, special_numbers)))
    print("Total number of special numbers =", total_special_numbers)


    if total_special_numbers <= 5:
        print("All special numbers are here:", ', '.join(map(str, special_numbers)))
    else:
        first_three = special_numbers[:3]
```

```
        last_three = special_numbers[-3:]
        print("First three and last three special numbers:", ', '.join(map(str, first_three)), ', '.join(map(str, last_three)))

    print("Time taken:", end_time - start_time, "seconds")

if __name__ == "__main__":
    main()
```

Petko, Georgiev 001318621

```
import time
import sympy

def is_palindrome(num):
    str_num = str(num)
    length = len(str_num)
    for i in range(length // 2):
        if str_num[i] != str_num[length - 1 - i]:
            return False
    return True

def generate_palindrome(length):
    if length < 1:
        return []

    palindromes = []

    if length == 1:
        palindromes = [2, 3, 5, 7]
```

```
    elif length % 2 == 0:
        for i in range (10**(length//2 - 1), 10**(length//2)):
            palindrome_str = str(i) + str(i)[::-1]
            if sympy.isprime(int(palindrome_str)):
                palindromes.append(int(palindrome_str))

    else:
        for i in range(10**(length//2), 10**(length//2 + 1)):
            palindrome_str = str(i) + str(i)[::-1]
            if sympy.isprime(int(palindrome_str)):
                palindromes.append(int(palindrome_str))

    return palindromes

def sieve_of_eratosthenes(limit):
    primes = [True] * (limit + 1)
    primes[0] = primes[1] = False
    p = 2
    while p * p <= limit:
        if primes[p]:
            for i in range(p * p, limit + 1, p):
                primes[i] = False
        p += 1
    return primes

test_cases = [
    (1, 2000),
    (100, 10000),
    (20000, 80000),
    (100000, 2000000),
    (2000000, 9000000),
    (10000000, 100000000),
    (100000000, 400000000),
    (1100000000, 15000000000),
    (15000000000, 100000000000),
]

def find_special_numbers_sequential():
    start_time = time.time()
    for i, (m, n) in enumerate(test_cases, start=1):
```

```
    print(f"\nTest case {i}:")
    primes = sieve_of_eratosthenes(n)
    special_numbers = []
    for j in range(m, n + 1, 1000):
        chunk_special_numbers = [num for num in range(j, min(j + 1000, n + 1)) if
primes[num] and is_palindrome(num)]
        special_numbers.extend(chunk_special_numbers)

    print(f"Total: {len(special_numbers)} Special Numbers")
    print(f"Time taken: {time.time() - start_time:.6f} seconds")
    print(f"Length of special_numbers: {len(special_numbers)}")
    if len(special_numbers) < 6:
        print(special_numbers)
    else:
        print(special_numbers[:3], special_numbers[-3:])

find_special_numbers_sequential()
```

Appendix B - Test cases for correctness

Code 1:

ID	Input	Output	Comments
1	1,100_000	Total special numbers: 113 Prime Palindromes: [2, 3, 5] [97879, 98389, 98689] Time: 0.0019970 s	Algorithm successfully identifies prime palindromes within the specified range and remonstrates consistent performance.

2	1, 1_000_000	Total special numbers: 113 Prime Palindromes: [2, 3, 5] [97879, 98389, 98689] Time: 0.0045060 s	Efficiently handles larger input while maintaining accurate identification of prime palindromes.
3	1, 10_000_000	Total special numbers: 781 Prime Palindromes: [2, 3, 5] [9980899, 9981899, 9989899] Time: 0.0210260 s	Scaling well with increased range, demonstrating consistent performance and accuracy.
4	1, 100_000_000	Total special numbers: 781 Prime Palindromes: [2, 3, 5] [9980899, 9981899, 9989899] Time: 0.0499930 s	Maintains efficiency and accuracy even with significantly larger input ranges.
5	1, 1_000_000_000	Total special numbers: 5953 Prime Palindromes: [2, 3, 5] [999676999, 999686999, 999727999] Time: 0.8102180 s	Handles large input ranges with increased number of prime palindromes while maintaining acceptable execution time.
6	1, 10_000_000_000	Total special numbers: 5953 Prime Palindromes: [2, 3, 5] [999676999, 999686999, 999727999] Time: 1.1197760 s	Performance remains stable with further increase in input range, demonstrating reliability and efficiency.
7	1, 100_000_000_000	Total special numbers: 5474 Prime Palindromes: [10000500001, 10000900001, 10001610001] [14998289941, 14998589941, 14998689941]	Successfully identifies prime palindromes within a very large range, although execution time increases significantly.

		Time: 64.374890 s	
8	1, 1_000_000_000_000	Total special numbers: 47995 Prime Palindromes: [2, 3, 5] [99998189999, 99998989999, 99999199999] Time: 68.4293890 s	Demonstrates the ability to handle exceptionally large input while passing the final test case, although execution time is noticeably higher.

Code 2:

c	Input	Output	Running Time (s)
1	1, 2000	Total special numbers: 20 [3, 5, 7, 797, 919, 929]	Time: 0.04205550000187941
2	100, 10000	Total special numbers: 15 [101, 131, 151, 797, 919, 929]	Time: 0.0011705999786499888
3	20000, 80000	Total special numbers: 48 [30103, 30203, 30403, 79397, 79697, 79997]	Time: 0.00261199998203665
4	100000, 2000000	Total special numbers: 190 [1003001, 1008001, 1022201, 1993991, 1995991, 1998991]	Time: 0.04432370001450181
5	2000000, 9000000	Total special numbers: 327 [3001003, 3002003, 3007003, 7985897, 7987897, 7996997]	Time: 0.1275572999729775
6	10000000, 100000000	[]	Time: 0.08359279998694547

7	100000000, 400000000	Total special numbers: 2704 Prime Palindromes: [100030001, 100050001, 100060001] [399737993, 399767993, 399878993]	Time: 6.593996400013566
8	1100000000, 15000000000	Total special numbers: 5474 Prime Palindromes: [10000500001, 10000900001, 10001610001] [14998289941, 14998589941, 14998689941]	Time: 117.19983990001492
9	15000000000, 100000000000	List of special numbers = [15001010051, 15002120051, 15002320051, 99998189999, 99998989999, 99999199999] Total number of special numbers = 36568	Execution time: 1117.302170102133
10	1, 1000000000000	Total special numbers: 47995 Prime Palindromes: [2, 3, 5] [99998189999, 99998989999, 99999199999]	Time: 1817.4956098000112

Appendix C - Evidence of team contribution

Communication Log

Brazington, Ella Prada Tello, Ainhoa Georgiev, Petko Aliyah, Lubis Tanzim, Md Abdul Raihan	31/01/24	Spoke in Lab	We kicked off our collaboration by introducing ourselves and then devised a plan to review the project specification thoroughly, ensuring a shared understanding among the team.
Brazington, Ella Prada Tello, Ainhoa Georgiev, Petko Aliyah, Lubis Tanzim, Md Abdul Raihan	31/01/24	Message	We established a group chat to facilitate communication and collaboration among team members.
Brazington, Ella Prada Tello, Ainhoa Georgiev, Petko Aliyah, Lubis Tanzim, Md Abdul Raihan	07/02/24	Spoke in Lab	We devised a plan to maintain regular communication regarding our progress and set weekly deadlines to stay on track. Additionally, we appointed a group leader to facilitate coordination and decision-making within the team.
Brazington, Ella Prada Tello, Ainhoa Georgiev, Petko Aliyah, Lubis Tanzim, Md Abdul Raihan	07/02/24	Email	Ainhoa shared the file with the team to facilitate the creation of our report.
Brazington, Ella Prada Tello, Ainhoa Georgiev, Petko Aliyah, Lubis Tanzim, Md Abdul Raihan	07/02/24	Message	We took the initiative to introduce ourselves within the group chat.

Brazington, Ella Prada Tello, Ainhoa Georgiev, Petko Aliyah, Lubis Tanzim, Md Abdul Raihan	13/02/24	Message	We engaged in discussions within the group chat about the tasks required to complete the project and shared updates on our progress thus far.
Brazington, Ella Prada Tello, Ainhoa Georgiev, Petko Aliyah, Lubis Tanzim, Md Abdul Raihan	14/02/24	Message	We revisited our progress within the group chat, exchanging insights on our individual contributions and seeking clarification by asking questions. Additionally, we offered suggestions on different approaches to the work and discussed the possibility of meeting up next week to further collaborate and enhance our progress.
Brazington, Ella Prada Tello, Ainhoa Georgiev, Petko Aliyah, Lubis Tanzim, Md Abdul Raihan	15/02/24	Message	We discussed the possibility of meeting up during the skills week to provide mutual assistance and support. Testing was a focal point of our conversation, as we were all simultaneously engaged in this aspect of the project.
Brazington, Ella Prada Tello, Ainhoa	16/02/24	Message	We engaged in a conversation about

Georgiev, Petko Aliyah, Lubis Tanzim, Md Abdul Raihan			the challenges we were facing with the hour limit, acknowledging the difficulty it presented in completing our tasks within the specified timeframe.
Brazington, Ella Prada Tello, Ainhoa Georgiev, Petko Aliyah, Lubis Tanzim, Md Abdul Raihan	19/02/24	Message	We discussed strategies for optimizing our code and explored alternative approaches to the project, recognizing the importance of finding efficient solutions to meet the project requirements.
Brazington, Ella Prada Tello, Ainhoa Georgiev, Petko Aliyah, Lubis Tanzim, Md Abdul Raihan	20/02/24	Message	We delved into a discussion about how RAM could potentially impact the speed of our code execution. Furthermore, we revisited our ongoing tests, recognizing their significance in evaluating and refining our code's performance.
Brazington, Ella Prada Tello, Ainhoa Georgiev, Petko Aliyah, Lubis Tanzim, Md Abdul Raihan	21/02/24	Message	We shared our individual plans for the day within the group chat, ensuring alignment and coordination in our tasks and efforts.

Brazington, Ella Prada Tello, Ainhoa Georgiev, Petko Aliyah, Lubis Tanzim, Md Abdul Raihan	26/02/24	Message	After a period of relative silence, we provided updates to each other within the group chat and resumed discussing our testing progress, ensuring that everyone was on the same page and moving forward together.
Brazington, Ella Prada Tello, Ainhoa Georgiev, Petko Aliyah, Lubis Tanzim, Md Abdul Raihan	28/02/24	Message	We strategized to select the top two options for our project the following week and began coordinating to agree on a suitable time for our meeting.
Brazington, Ella Prada Tello, Ainhoa Georgiev, Petko Aliyah, Lubis Tanzim, Md Abdul Raihan	06/03/24	Message	We were in the process of finalizing a mutually convenient time for our call to decide on the final two best programs for the project.
Brazington, Ella Prada Tello, Ainhoa Georgiev, Petko Aliyah, Lubis Tanzim, Md Abdul Raihan	06/03/24	Spoke in Lab	We reached a consensus to select the two programs that achieved the highest number of passed test cases within an hour. This decision will guide us forward as we proceed with the chosen programs for the project.

Brazington, Ella Prada Tello, Ainhoa Georgiev, Petko Aliyah, Lubis Tanzim, Md Abdul Raihan	09/03/24	Spoke in Lab	We discussed among ourselves to determine which individual had successfully passed the highest number of test cases.
Brazington, Ella Prada Tello, Ainhoa Georgiev, Petko Aliyah, Lubis Tanzim, Md Abdul Raihan	11/03/24	Message	We deliberated on the approach to completing the report while ensuring that our chosen codes continued to pass tests. Ultimately, we made the decision on the two codes that we would proceed with.
Brazington, Ella Prada Tello, Ainhoa Georgiev, Petko Aliyah, Lubis Tanzim, Md Abdul Raihan	12/02/24	Message	We discussed some feedback received from a tutor, considering how we could integrate it into our project for improvement.
Brazington, Ella Prada Tello, Ainhoa Georgiev, Petko Aliyah, Lubis Tanzim, Md Abdul Raihan	13/03/24	Spoke in Lab	During the lab session, we discussed that our final task remaining was to complete the report.
Brazington, Ella Prada Tello, Ainhoa Georgiev, Petko Aliyah, Lubis Tanzim, Md Abdul Raihan	14/03/24	Message	We discussed our respective responsibilities within the report and identified the remaining sections that needed to be written.
Brazington, Ella Prada Tello, Ainhoa	15/03/24	Message	We shared our progress thus far

Georgiev, Petko Aliyah, Lubis Tanzim, Md Abdul Raihan			and exchanged questions to seek assistance and clarification from one another.
Brazington, Ella Prada Tello, Ainhoa Georgiev, Petko Aliyah, Lubis Tanzim, Md Abdul Raihan	16/03/24	Message	Once again, we discussed the remaining tasks and identified who needed to contribute more to ensure completion.