Algorithms and Data Structures (ADS) - COMP1819

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

Group Name: **21_06**
Team members:

Member	Name	ID	Contribution %
1	Hoque, Aminul	001309101	100%
2	Majmundar, Tanisha Tapan	001296006	100%
3	Risma, Israt Jahan	001339277	100%
4	Toba, Sabiha Ahmed	001340510	100%
5	Butt, Omer Tariq	001331777	0%

Contents

1. Create unique solutions!

Student 1-6's solution

Explain your understanding of the problem, and approach to solve it.

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

Member 1: Hoque, Aminul [001309101]

Understanding the problem:
According to how I see it, the goal is to write a program that can find every prime palindrome between m and n. The input, the computations, and the output should be the main components of this program. The range that needs to be used to identify special numbers should be defined by the input of the integers m and n. A number should be calculated to see if it is only divisible by 1 and itself and if it stays the same when its digits are switched. Additionally, the output must show the correct figures in addition to the overall count of palindromic primes.

Approach to the problem:
Initially, the primary approach for solving the problem was to use the Sieve of Eratosthenes method to locate the prime numbers from a range of m and n, and then to determine if those numbers were palindromes. However, we discovered that Python lists are limited to 2 billion elements when testing large numbers. The new method for solving the problem was to create all of the palindromes by creating the first half of a number, flipping it, and then joining it together. Then, it is determined whether or not each number is a prime number and falls between the ranges of m and n.

Description of the code:

- **palindrome_gen()** draws inspiration from an ancient mathematical tool, the abacus. This function depicts the left half of the palindromic number by taking half the length of the upper limit and creating a list of that length, where each element represents a digit. After that, the number is flipped up to the 2nd to last digit, at which point both numbers are combined. We discovered as a group that, other than 11, there are no prime palindromes with an even number of digits. Thus, numbers are created with an odd number of digits only.

- **check_prime()** determines whether a given number is prime by returning true. The function divides a given number, x, by each odd number up to x's square root. If the number isn't divisible, or is considered even, the function will return False.

Results

#	Input	Output	Running time (s)
1	1, 2_000	Total: 20 First three: [2, 3, 5] Last three: [797, 919, 929]	0.000997304916381836
2	100, 10_000	Total: 15 First three: [101, 131, 151] Last three: [797, 919, 929]	0.003991603851318359
3	20_000, 80_000	total: 48 First three: [30103, 30203, 30403] Last three: [79397, 79697, 79997]	0.0049855709075927734
4	100_000, 2_000_000	total: 190 First three: [1003001, 1008001, 1022201] Last three: [1993991, 1995991, 1998991]	0.04887056350708008
5	2_000_000, 9_000_000	total: 327 First three: [3001003, 3002003, 3007003] Last three: [7985897, 7987897, 7996997]	0.09075570106506348
6	10_000_000, 100_000_000	total: 0 []	0.3151569366455078
7	100_000_000, 400_000_000	total: 2704 First three: [100030001, 100050001, 100060001] Last three: [399737993, 399767993, 399878993]	3.284210681915283
8	1_100_000_000, 15_000_000_000	total: 5474 First three: [10000500001, 10000900001, 10001610001] Last three: [14998289941, 14998589941, 14998689941]	43.5370888710022
9	15_000_000_000, 100_000_000_000	total: 36568 First three: [15001010051, 15002120051, 15002320051] Last three: [99998189999, 99998989999, 99999199999]	630.5995135307312
10	1, 1_000_000_000_000	total: 47995 First three: [2, 3, 5] Last three: [99998189999, 99998989999, 99999199999]	758.1990327835083

Member 2: Majmundar, Tanisha Tapan [001296006]

Understanding the problem:

The task is to find special numbers that are both prime (only divisible by 1 and themselves) and palindromes (read the same forwards and backwards) within a certain range. We'll carefully check each number from 'm' to 'n' to see if it fits both criteria. If we find less than six of these unique numbers, we'll just list them all. But if we find more than six, we'll simplify things by showing only the first three smallest and the last three largest special numbers. This way, we can neatly display these interesting numbers without overwhelming anyone, making sure to present them in a clear and organized way based on how many we find in the range.

Approach to the problem:

- is_prime Function: Checks if a number is prime by testing its divisibility, handling special cases efficiently.
- generate_palindromes Function: Generates palindromic integers within a specified range, covering both even and odd lengths.
- Find_palindromi_primes_in_range Function: Identifies specisl palindromic prime numbers within a given range by combining prime checking and palindrome generation.
- Test Case Loop: Utilizes a loop structure to iterate over test cases, identifying unique integers and displaying the first three unique palindromic primes.

  Overall: The script offers a concise and effective method for locating and presenting Special palindromic prime integers within predefined ranges, ensuring both clarity and speed in execution.

Results

	Input	Output	Running time (s)
1.	1 to 2000	First 3 Special Palindromic Prime Numbers: [2, 3, 5] Last 3 Special Palindromic Prime Numbers: [797, 919, 11] Total Special Palindromic Primes: 16	0.001354217529296875 seconds
2.	100 to 10000	First 3 Special Palindromic Prime Numbers: [131, 151, 191] Last 3 Special Palindromic Prime Numbers: [757, 797, 919] Total Special Palindromic Primes: 09	0.04256010055541992 seconds
3.	20000 to 80000	First 3 Special Palindromic Prime Numbers: [30103, 30703, 31513]	0.07870674133300781 seconds

		Last 3 Special Palindromic Prime Numbers: [78787, 79397, 79997] Total Special Palindromic Primes: 26	
4.	100000 to 2000000	First 3 Special Palindromic Prime Numbers: [1003001, 1035301, 1043401] Last 3 Special Palindromic Prime Numbers: [1987891, 1993991, 1995991] Total Special Palindromic Primes: 100	7.012953758239746 seconds
5.	2000000 to 9000000	First 3 Special Palindromic Prime Numbers: [3001003, 3007003, 3065603] Last 3 Special Palindromic Prime Numbers: [7977797, 7985897, 7987897] Total Special Palindromic Primes: 327	37.604594707489014 seconds
6.	10000000 to 100000000	No special palindromic prime numbers found in the specified range. Total Special Palindromic Primes: 0	44.1017582416534424 seconds
7.	100000000 to 400000000	First 3 Special Palindromic Prime Numbers: [100030001, 100050001, 100060001] Last 3 Special Palindromic Prime Numbers: [399737993, 399767993, 399878993] Total Special Palindromic Primes: 2704	57.1301037311553955 seconds
8.	1100000000 to 15000000000	First 3 Special Palindromic Prime Numbers: [10000500001, 10000900001, 10001610001] Last 3 Special Palindromic Prime Numbers: [14998289941, 14998589941, 14998689941] Total Special Palindromic Primes: 5474	61.04614963531494 seconds
9.	15000000000 to 100000000000	First 3 Special Palindromic Prime Numbers: [15001010051, 15002120051, 15002320051] Last 3 Special Palindromic Prime Numbers: [99998189999, 99998989999, 99999199999] Total Special Palindromic Primes: 36568	703.3549198627472 seconds

10.	1 to 1000000000000	First 3 Special Palindromic Prime Numbers: [2, 3, 5] Last 3 Special Palindromic Prime Numbers: [99998189999, 99998989999, 99999199999] Total Special Palindromic Primes: 47993	428.4940645694732 7 seconds

Member 3: Risma, Israt Jahan[001339277]

Understanding the problem:
The assignment requires to create a Python program that finds special numbers in given ranges. In this case special numbers are those that are both prime (only divisible by 1 and themselves) and palindromic (read the same forwards and backwards). The program should ask for two positive numbers, m and n (where m is smaller than n), and then count and display how many special numbers are between m and n, including m and n. If there are fewer than 6 special numbers, it should show all of them. If there are 6 or more, it should show the first three smallest and the last three biggest.

Approach to the problem:
To solve this problem, a function is created to generate palindromic numbers. Then, another function is made to check if a number is prime. After that, each given range is traversed to find the special numbers within it. The count of special numbers is determined and displayed according to the rules: all if there are fewer than 6, or the first three smallest and the last three biggest if there are 6 or more.

Description of the code:
The **generate_palindromes** function generates palindromic numbers up to 6 digits by appending the reverse of a number to itself.
The **is_prime** function checks whether a number is prime by iterating up to its square root and checking divisibility.
The **find_special_numbers** function iterates through the given test cases, generating palindromic numbers and checking for primality and range inclusion.
It prints the total number of special numbers within each range and displays them if there are fewer than 6. Otherwise, it shows the first three smallest and last three biggest special numbers.
Finally, it calculates and prints the total time taken to execute all test cases.

Result

#	Input	Output	Running Time(s)
1	1, 2_000	Total special numbers: 20 [2, 3, 5] [797, 919, 929]	Time taken for Test Case 1: 0.7198505401611328 seconds
2	100, 10_000	Total special numbers: 15 [101, 131, 151] [797, 919, 929]	Time taken for Test Case 2: 0.742131233215332 seconds
3	20_000, 80_000	Total special numbers: 48 [30103, 30203, 30403] [79397, 79697, 79997]	Time taken for Test Case 3: 0.9688525199890137 seconds
4	100_000, 2_000_000	Total special numbers: 190 [1003001, 1008001, 1022201] [1993991, 1995991, 1998991]	Time taken for Test Case 4: 0.7586617469787598 seconds

5	2_000_000, 9_000_000	Total special numbers: 327 [3001003, 3002003, 3007003] [7985897, 7987897, 7996997]	Time taken for Test Case 5: 0.6930677890777588 seconds
6	10_000_000, 100_000_000	Total special numbers: 0 Total special numbers: []	Time taken for Test Case 6: 0.5128610134124756 seconds
7	100_000_000, 400_000_000	Total special numbers: 2704 [100030001, 100050001, 100060001] [399737993, 399767993, 399878993]	Time taken for Test Case 7: 1.2279651165008545 seconds
8	1_100_000_000, 15_000_000_000	Total special numbers: 5474 [10000500001, 10000900001, 10001610001] [14998289941, 14998589941, 14998689941]	Time taken for Test Case 8: 13.160057067871094 seconds
9	15_000_000_000, 100_000_000_000	Total special numbers: 36568 [15001010051, 15002120051, 15002320051] [99998189999, 99998989999, 99999199999]	Time taken for Test Case 9: 175.9407331943512 seconds
10	1, 1_000_000_000_000	Total special numbers: 47995 [2, 3, 5] [99998189999, 99998989999, 99999199999]	Time taken for Test Case 10: 188.04311800003052 seconds

Member 4: Toba, Sabiha Ahmed [001340510]

Understanding the problem:
The task is to find all prime numbers between 'm' and 'n', which are positive integers, that have the property of being palindromes, within a particular range. Prime and palindrome qualities combined, these unique numbers are carefully found and kept ready for display. All are displayed if, within the given range, the count of such special numbers is less than six. The first three smallest and final three largest special numbers are all that are shown on the display if there are more than six. This methodical technique makes sure that the exceptional numbers that have been identified are presented in an orderly and thorough manner, accommodating different situations according to the quantity that is found inside the range.

Approach to the problem:
is_prime Function: Checks a number's divisibility to see if it's prime.

generate_palindromes Function: Within a certain range, generates palindromic integers.

find_palindromic_primes_in_range Function: Within a specified range, finds special palindromic prime numbers.

Test Case Loop: This loops over test cases, identifies unique integers and shows the first three unique palindromic primes.

Overall, this script ensures clarity and speed in execution by offering a simple method for locating and displaying special palindromic prime integers within designated ranges.

Results

	Input	Output	Running time (s)
11.	1 to 2000	First 3 Special Palindromic Prime Numbers: [2, 3, 5] Last 3 Special Palindromic Prime Numbers: [797, 919, 929] Total Special Palindromic Primes: 18	1.048879623413086 seconds
12.	100 to 10000	First 3 Special Palindromic Prime Numbers: [101, 131, 151] Last 3 Special Palindromic Prime Numbers: [797, 919, 929] Total Special Palindromic Primes: 15	1.064882755279541 seconds
13.	20000 to 80000	First 3 Special Palindromic Prime Numbers: [30103, 30203, 30403] Last 3 Special Palindromic Prime Numbers: [79397, 79697, 79997]	1.0761044025421143 seconds

		Total Special Palindromic Primes: 48	
14.	100000 to 2000000	First 3 Special Palindromic Prime Numbers: [1003001, 1008001, 1022201] Last 3 Special Palindromic Prime Numbers: [1993991, 1995991, 1998991] Total Special Palindromic Primes: 190	1.0316343307495117 seconds
15.	2000000 to 9000000	First 3 Special Palindromic Prime Numbers: [3001003, 3002003, 3007003] Last 3 Special Palindromic Prime Numbers: [7985897, 7987897, 7996997] Total Special Palindromic Primes: 327	1.050037145614624 seconds
16.	10000000 to 100000000	No special palindromic prime numbers found in the specified range. Total Special Palindromic Primes: 0	1.1017582416534424 seconds
17.	100000000 to 400000000	First 3 Special Palindromic Prime Numbers: [100030001, 100050001, 100060001] Last 3 Special Palindromic Prime Numbers: [399737993, 399767993, 399878993] Total Special Palindromic Primes: 2704	2.7301037311553955 seconds
18.	1100000000 to 15000000000	First 3 Special Palindromic Prime Numbers: [10000500001, 10000900001, 10001610001] Last 3 Special Palindromic Prime Numbers: [14998289941, 14998589941, 14998689941] Total Special Palindromic Primes: 5474	31.14614963531494 seconds
19.	15000000000 to 100000000000	First 3 Special Palindromic Prime Numbers: [15001010051, 15002120051, 15002320051] Last 3 Special Palindromic Prime Numbers: [99998189999, 99998989999, 99999199999] Total Special Palindromic Primes: 36568	399.6549198627472 seconds
20.	1 to 1000000000000	First 3 Special Palindromic Prime Numbers: [2, 3, 5] Last 3 Special Palindromic Prime Numbers: [99998189999, 99998989999, 99999199999] Total Special Palindromic Primes: 47993	428.49406456947327 seconds

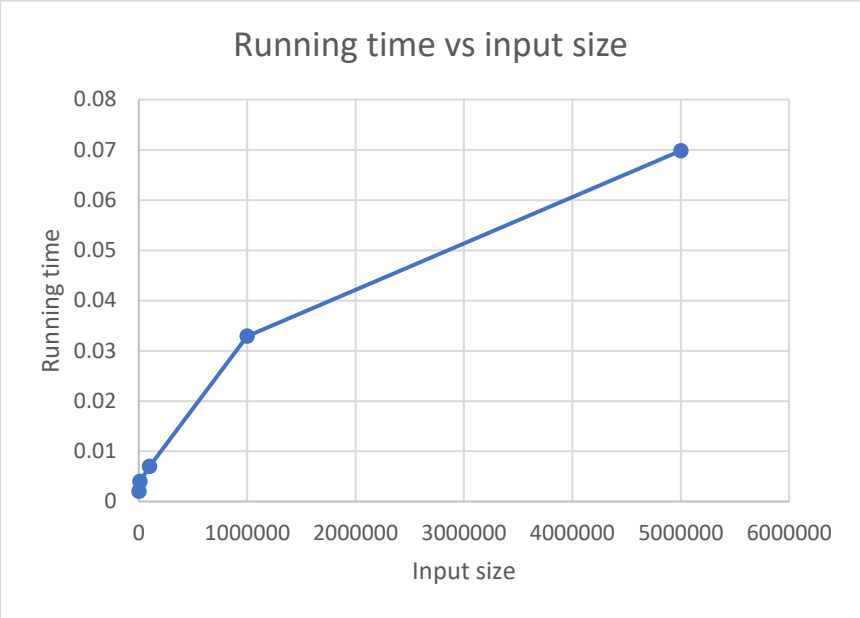
2. Test and analyse your solution!

Member 1: Hoque, Aminul [001309101]

Your test cases:

c	Input	Output	Justification	Student 1 results
1	1, 1_000	total: 20 First three: [2, 3, 5] Last three: [797, 919, 929]	Provides a test for a smaller range	0.0019948482513427734 seconds
2	1_000, 10_000	total: 0 []	Test if there are no outputs for even digit numbers while keep a consistent increase in test range	0.003988981246948242 seconds
3	10_000, 100_000	total: 93 First three: [10301, 10501, 10601] Last three: [97879, 98389, 98689]	Increase the test range by x10	0.006982326507568359 seconds
4	100_000, 1_000_000	total: 0 []	Test for even digit numbers with a bigger range	0.0329132080078125 seconds
5	1_000_000, 5_000_000	total: 362 First three: [1003001, 1008001, 1022201] Last three: [3994993, 3997993, 3998993]	A test for 7-digit numbers while keeping a moderate increase in test range	0.06981253623962402 seconds

Running time graphs



Complexity analysis

O((10^(d/2)) * ((√n)/2))

check_prime() Function:
This function takes the larger value, n, and iterates up to the square root of it. The worst case for this function is to iterate from 3 to square root n, which gives you a time complexity of O(√n). However, the function excludes even numbers, therefore the iteration is cut in half. Therefore, the time complexity is **O((√n)/2)**.

palindrome_gen() Function:
Let 'd' equal the number of digits of the larger value, n. The value d is then cut in half. Since the function uses a loop depending on the number of digits d/2, for each digit, a for loop iterates 10 times. Therefore, the time complexity for this function is **O(10^(d/2))**.

In conclusion, the overall time complexity of the code would approximately be **O((10^(d/2)) * ((√n)/2))**, where n is the value provided by user input, and d is the number of digits in the value n.

Member 2: Majmundar, Tanisha Tapan [001296006]

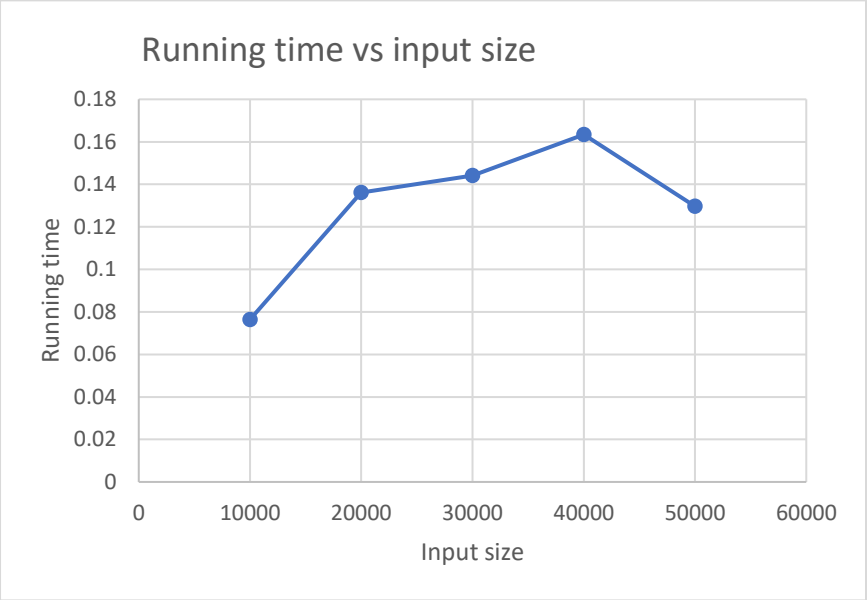
Your test cases:

c	Input	Output	Student X results
1	1 to 100	First 3 Special Palindromic Prime Numbers: [2, 3, 5] Last 3 Special Palindromic Prime Numbers: [2, 3, 5] Total Special Palindromic Primes: 7	1.4108490467071533 seconds
2	100 to 500	First 3 Special Palindromic Prime Numbers: [131, 151, 191] Last 3 Special Palindromic Prime Numbers: [313, 353, 373] Total Special Palindromic Primes: 6	0.0011470317840576172 seconds
3	500 to 2500	First 3 Special Palindromic Prime Numbers: [727, 757, 787] Last 3 Special Palindromic Prime Numbers: [797, 919, 929] Total Special Palindromic Primes: 6	0.007490873336791992 seconds
4	2500 to 12500	First 3 Special Palindromic Prime Numbers: [10301, 10501, 10601] Last 3 Special Palindromic Prime Numbers: [11311, 11411, 12421] Total Special Palindromic Primes: 6	0.09402680397033691 seconds
5	12500 to 62500	First 3 Special Palindromic Prime	0.15943551063537598 seconds

		Numbers: [12721, 13331, 13931] Last 3 Special Palindromic Prime Numbers: [37573, 38183, 38783] Total Special Palindromic Primes: 25	

The test cases provided serve to evaluate the performance and efficacy of the find_palindromic_primes_in_range function across a spectrum of range sizes. These test cases encompass a range of magnitudes, from smaller ones, such as (1, 100), to larger ones, such as (12, 500, 62500). By varying the sizes of the ranges, we can observe how the function performs under different computational workloads. This comprehensive testing approach allows us to assess the function's capability to handle diverse input ranges and provides insights into its efficiency and utility across various scenarios.

Running time graphs



Complexity analysis

O(N * sqrt(N))

is_prime Function:
- The is_prime function determines if a given number is prime by checking divisibility up to its square root.
- Time Complexity: O(sqrt(n)), where n is the input number.

generate_palindromes Function:
- The generate_palindromes function creates palindromic numbers up to 10^max_digits.
- Time Complexity: O(10^max_digits), involving iterating over every conceivable number of digits.

find_palindromic_primes_in_range Function:
- The find_palindromic_primes_in_range function utilizes generate_palindromes to produce palindromic integers.
- Subsequently, it uses is_prime to verify if the palindromic integers fall within the specified range and are prime.
- Time Complexity: O(N * sqrt(N)), where N is the length of the specified range.

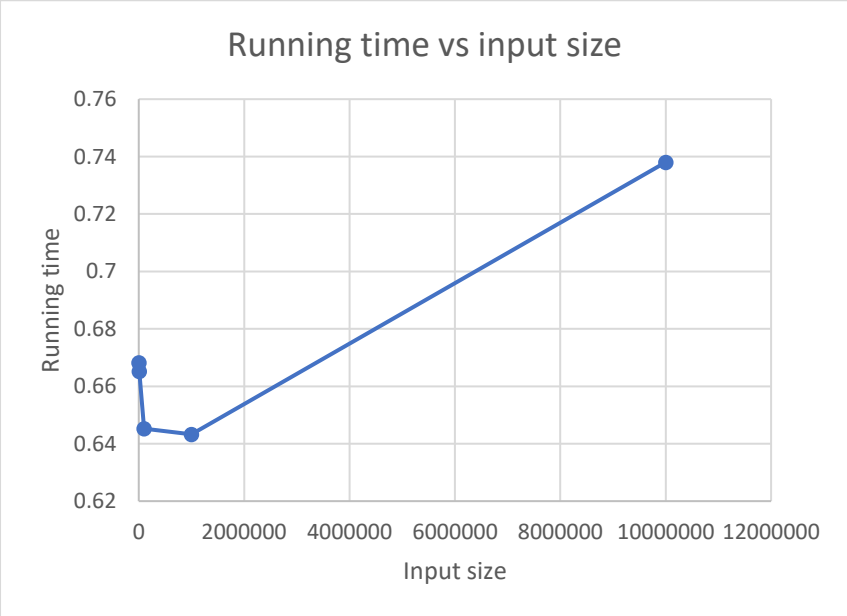
Member 3: Risma, Israt Jahan[001339277]

Your test cases:

c	Input	Output	Student 3 results
1	1, 1_000	Total special numbers: 20 [2, 3, 5] [797, 919, 929]	0.6682071685791016 seconds
2	1_000, 10_000	Test Case 2: Total special numbers: 0 Total special numbers: []	0.6652190685272217 seconds
3	10_000, 100_000	Test Case 3: Total special numbers: 93 [10301, 10501, 10601] [97879, 98389, 98689]	0.6452748775482178 seconds
4	100_000, 500_000	Test Case 4: Total special numbers: 0 Total special numbers: []	0.6432785987854004 seconds
5	100_000, 1_000_000	Test Case 5: Total special numbers: 668 [1003001, 1008001, 1022201] [9980899, 9981899, 9989899]	0.7380237579345703 seconds

For every stage, these test cases increase by a multiple of 10. Through changing the input range from smaller to higher, we may evaluate the program's performance at varying computational complexity levels.

Running time graphs



Complexity analysis

O(10^6 * √n)

The function is_prime() take the value n and square roots it. For the worst case scenario, the function will iterate from 3 to square root n, therefore the time complexity for this function would be O(√n)
The generate_palindromes() function take a constant value of 6 to represent how many digits should be utilised to create the palindromes. Worst case scenario is the function will loop from 8 to 10^6. Thus, the time complexity for this function is constant at O(10^6)
The time complexity for the whole program is approximately O(10^6 * √n)

Member 4: Toba, Sabiha Ahmed [001340510]

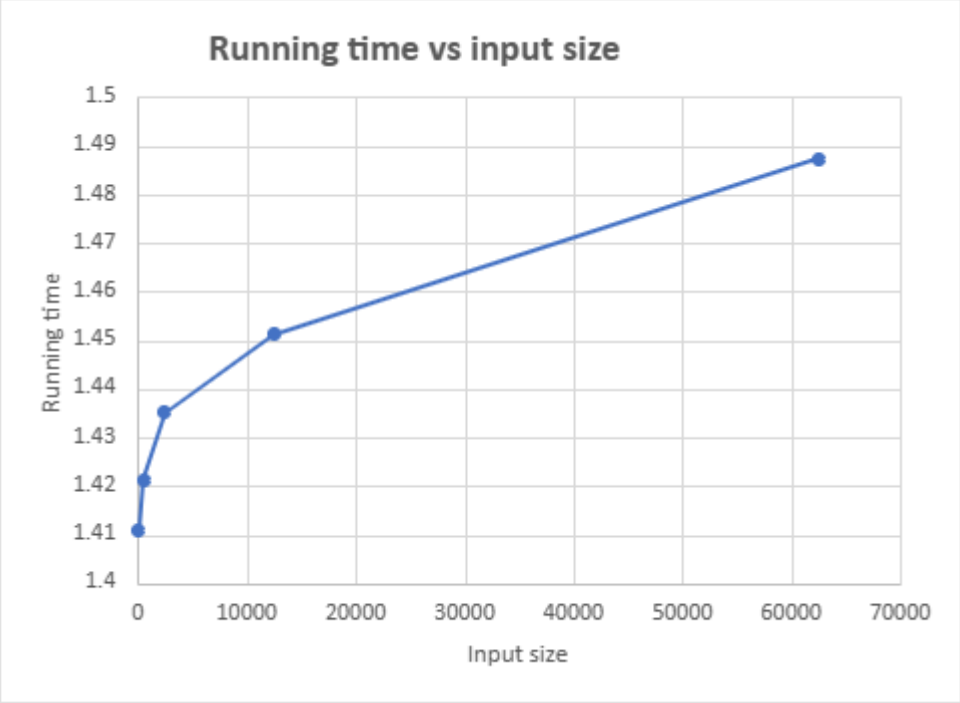
Your test cases:

c	Input	Output	Student X results
1	1 to 100	First 3 Special Palindromic Prime Numbers: [2, 3, 5] Last 3 Special Palindromic Prime Numbers: [2, 3, 5] Total Special Palindromic Primes: 3	1.4108490467071533 seconds
2	100 to 500	First 3 Special Palindromic Prime Numbers: [101, 131, 151] Last 3 Special Palindromic Prime Numbers: [353, 373, 383] Total Special Palindromic Primes: 9	1.4211678218841553 seconds
3	500 to 2500	First 3 Special Palindromic Prime Numbers: [727, 757, 787] Last 3 Special Palindromic Prime Numbers: [797, 919, 929] Total Special Palindromic Primes: 6	1.4350895977020264 seconds
4	2500 to 12500	First 3 Special Palindromic Prime Numbers: [10301, 10501, 10601] Last 3 Special Palindromic Prime Numbers: [11311, 11411, 12421] Total Special Palindromic Primes: 6	1.4512822341918945 seconds
5	12500 to 62500	First 3 Special Palindromic Prime Numbers: [12721, 12821, 13331] Last 3 Special Palindromic Prime Numbers: [38183, 38783, 39293]	1.487254409790039 seconds

		Total Special Palindromic Primes: 44	

The test cases that are offered are designed to assess the find_palindromic_primes_in_range function's performance and usefulness over various ranges. The sizes of the test cases vary, ranging from smaller ones (like 1, 100) to bigger ones (like 12, 500, 62500).
The function's performance under various computing workloads may be observed by adjusting the range sizes.

Running time graphs



Complexity analysis

is_prime function:
The is_prime function climbs to the number's square root to determine if a given number is prime or not.
Therefore, O(sqrt(n)), where n is the input number, is the time complexity.

generate_palindromes function:
Palindromic numbers up to 10^max_digits may be created using this function.

O(10^max_digits) is the temporal complexity of the procedure, which entails iterating over every conceivable number of digits.

find_palindromic_primes_in_range function:
It adds O(10^max_digits) to the time complexity by using generate_palindromes to produce palindromic integers.

Afterwards, it uses the is_prime function to verify if the palindromic integers it has filtered fall under the specified range and are prime.
The range's length, N, is the primary factor that determines the temporal complexity.

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.

As a group, we decided to take Aminul's and Israt's code since they both have similar methods yet many differences between each code.

Starting with both codes, the main optimisation features we implemented were:

- When checking if a number is prime, check if each number up to sqrt(n) is divisible rather than checking every number up until n. This brings down the time complexity of this feature from O(n) to O(sqrt(n)).
- We also found that we only need to check if a number is prime with odd numbers only, this cuts the time complexity in half.
  We found that checking if larger numbers were prime took the most time, therefore this was our first step of optimisation.
- When creating the palindrome numbers, we took advantage of slicing strings to flip the number. The main concern was that it would take too long to convert the integer into a string then find the correct position to cut the string in half and flip it. By implementing slicing, the code became much more efficient to reduce the number of lines of code from 5+ lines into 2.

Results

Aminul's results:

#	Input	Output	Running time (s)	#
1	1, 2_000	Total: 20 First three: [2, 3, 5] Last three: [797, 919, 929]	0.000997304916381836	1
2	100, 10_000	Total: 15 First three: [101, 131, 151] Last three: [797, 919, 929]	0.003991603851318359	2
3	20_000, 80_000	total: 48 First three: [30103, 30203, 30403]	0.0049855709075927734	3

		Last three: [79397, 79697, 79997]		
4	100_000, 2_000_000	total: 190 First three: [1003001, 1008001, 1022201] Last three: [1993991, 1995991, 1998991]	0.04887056350708008	4
5	2_000_000, 9_000_000	total: 327 First three: [3001003, 3002003, 3007003] Last three: [7985897, 7987897, 7996997]	0.09075570106506348	5

Israt's results:

#	Input	Output	Running Time(s)
1	1, 2_000	Total special numbers: 20 [2, 3, 5] [797, 919, 929]	Time taken for Test Case 1: 0.7198505401611328 seconds
2	100, 10_000	Total special numbers: 15 [101, 131, 151] [797, 919, 929]	Time taken for Test Case 2: 0.742131233215332 seconds
3	20_000, 80_000	Total special numbers: 48 [30103, 30203, 30403] [79397, 79697, 79997]	Time taken for Test Case 3: 0.9688525199890137 seconds
4	100_000, 2_000_000	Total special numbers: 190 [1003001, 1008001, 1022201] [1993991, 1995991, 1998991]	Time taken for Test Case 4: 0.7586617469787598 seconds
5	2_000_000, 9_000_000	Total special numbers: 327 [3001003, 3002003, 3007003] [7985897, 7987897, 7996997]	Time taken for Test Case 5: 0.6930677890777588 seconds
6	10_000_000, 100_000_000	Total special numbers: 0 Total special numbers: []	Time taken for Test Case 6: 0.5128610134124756 seconds
7	100_000_000, 400_000_000	Total special numbers: 2704 [100030001, 100050001, 100060001] [399737993, 399767993, 399878993]	Time taken for Test Case 7: 1.2279651165008545 seconds

8	1_100_000_000, 15_000_000_000	Total special numbers: 5474 [10000500001, 10000900001, 10001610001] [14998289941, 14998589941, 14998689941]	Time taken for Test Case 8: 13.160057067871094 seconds
9	15_000_000_000, 100_000_000_000	Total special numbers: 36568 [15001010051, 15002120051, 15002320051] [99998189999, 99998989999, 99999199999]	Time taken for Test Case 9: 175.9407331943512 seconds
10	1, 1_000_000_000_000	Total special numbers: 47995 [2, 3, 5] [99998189999, 99998989999, 99999199999]	Time taken for Test Case 10: 188.04311800003052 seconds

4. Compare the performance!

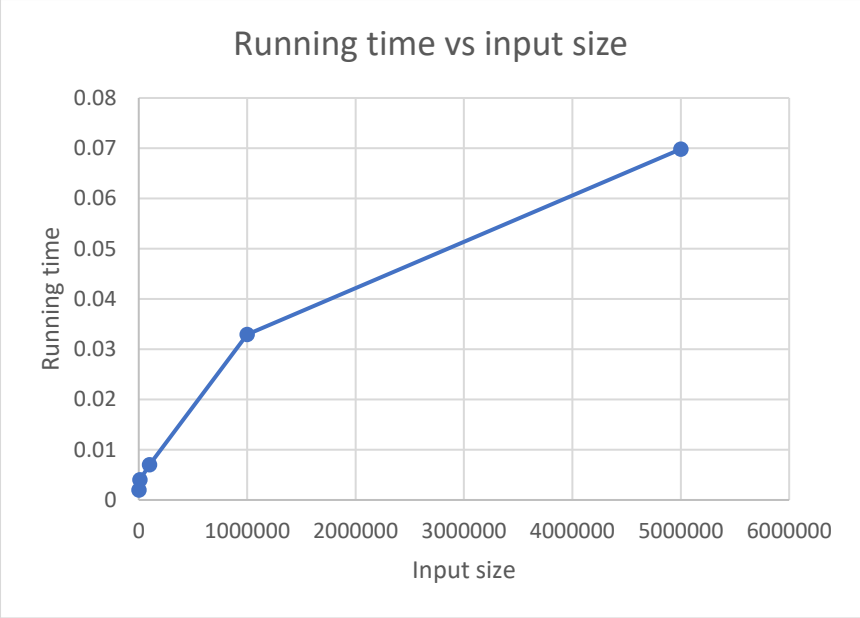
Time complexities and big-O notations

Since both codes use similar methods to find the prime palindromes, both time complexities are similar.
However, Israt's code is more efficient when it comes to run time since she converts the integer straight into a string. Aminul's method of using the array to generate palindromes takes more time to process. Therefore, Aminul's runtime is much slower.
Aminul's code on the other hand only generates palindromes with odd number of digits. As a result of even number of digits not getting checked, this reduces the time complexity by half.

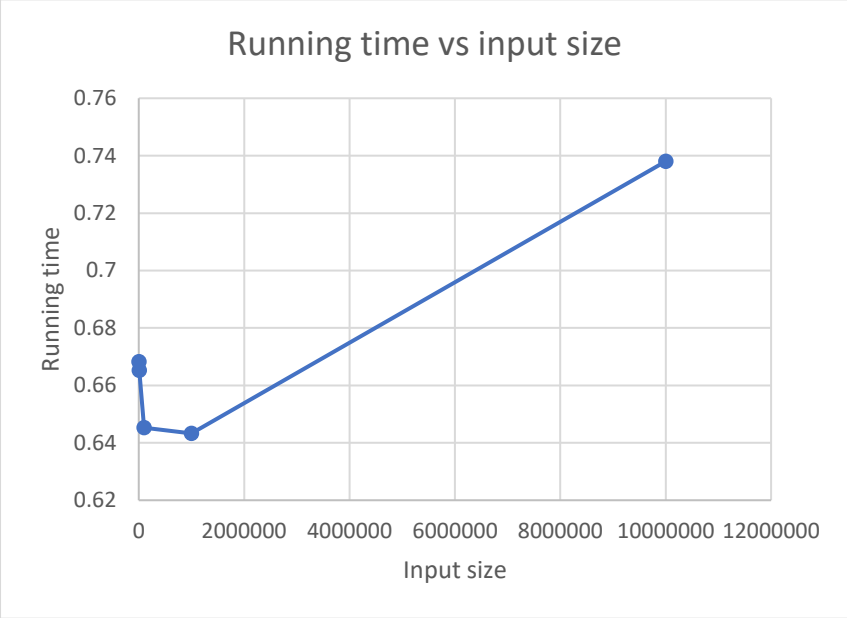
Therefore the time Complexity for Aminul's code is ***O((10^(d/2)) * ((√n)/2))*** and for Israt's code, it is ***O(10^6 * √n)***

Running time graphs

Aminul's Graph



Israt's Graph



5. Reflecting on teamwork!

Contribution mark

Name	ID	Task 1 (30%)	Task 2 (20%)	Task 3 (20%)	Task 4 (15%)	Task 5 (15%)	**Contribution mark (100%)**
Hoque, Aminul (Group leader)	001309101	30%	20%	20%	15%	15%	100%
Majmundar, Tanisha Tapan	001296006	30%	20%	20%	15%	15%	100%
Risma, Israt Jahan	001339277	30%	20%	20%	15%	15%	100%
Toba, Sabiha Ahmed	001340510	30%	20%	20%	15%	15%	100%
Butt, Omer Tariq	001331777	0%	0%	0%	0%	0%	0%

Limitation discussion

Your group might discuss the technical challenges, participation/engagement, collaboration, leadership, problem-solving skills, creativity and innovation, or communication dynamic topics.

As a group, we found that approaching the problem head on led to us making many mistakes in our code, hence making the code very unoptimized and inefficient. As numbers grew larger, checking for prime numbers took up most of the time. As a result, the group decided to find the palindromes first so that there were less numbers to check if they were primes, which greatly improved the code's efficiency.
Coming to this conclusion required mostly collaborative teamwork and meeting together once a week to provide updates with the code or any ideas that we may have. We took time out of each week to look at each other's code to either draw inspiration or fix any errors that they may have.

Weekly journal

	Task note	**Status**
Week 1: 07/02 – 14/02		
Hoque, Aminul (Group leader)	Introduced to the group. Read the coursework specification.	Not started the code
Majmundar, Tanisha Tapan	Introduced to the group. Read the coursework specification.	Not started the code
Risma, Israt Jahan	Introduced to the group. Read the coursework specification.	Not started the code
Toba, Sabiha Ahmed	Introduced to the group. Read the coursework specification.	Not started the code
Week 2: 14/02 – 21/02		

Hoque, Aminul (Group leader)	Assigned tasks for everyone to research a way to find prime numbers	Wrote a code to check for prime numbers
Majmundar, Tanisha Tapan	Write a code that checks for prime numbers	Wrote a code to check for prime numbers
Risma, Israt Jahan	Write a code that checks for prime numbers	Wrote a code to check for prime numbers
Toba, Sabiha Ahmed	Write a code that checks for prime numbers	Wrote a code to check for prime numbers
Week 3: 21/02 – 28/02		
Hoque, Aminul (Group leader)	Assigned everyone to find an efficient way to find palindromes within a range	Wrote a code to find all the prime palindromes within a range
Majmundar, Tanisha Tapan	Write a code that checks finds palindromes and combine both codes	Wrote a code to find all the prime palindromes within a range
Risma, Israt Jahan	Write a code that checks finds palindromes and combine both codes	Wrote a code to find all the prime palindromes within a range
Toba, Sabiha Ahmed	Write a code that checks finds palindromes and combine both codes	Wrote a code to find all the prime palindromes within a range
Week 4: 28/02 – 06/03		
Hoque, Aminul (Group leader)	Tried a different approach to find the palindromes. As a group, came to the conclusion to generate palindromes rather than checking each number	Testing out new methods
Majmundar, Tanisha Tapan	Find a way to generate palindromes	Testing out new methods
Risma, Israt Jahan	Find a way to generate palindromes	Testing out new methods
Toba, Sabiha Ahmed	Find a way to generate palindromes	Testing out new methods
Week 5: 6/03 – 13/03		
Hoque, Aminul (Group leader)	Finalise the code, pick two different methods to optimise	Finished individual codes
Majmundar, Tanisha Tapan	Help optimise one of the codes	Testing out new methods
Risma, Israt Jahan	Help optimise one of the codes	Testing out new methods

Toba, Sabiha Ahmed	Help optimise one of the codes	Testing out new methods
Week 6: 13/03 – 19/03		
Hoque, Aminul (Group leader)	Write the report, including commenting on the code and analysing complexity	
Majmundar, Tanisha Tapan	Write the report, including commenting on the code and analysing complexity	
Risma, Israt Jahan	Write the report, including commenting on the code and analysing complexity	
Toba, Sabiha Ahmed	Write the report, including commenting on the code and analysing complexity	

Reference

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

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.

```
"""
Author: Aminul Hoque [001309101]
"""

import time

# This function checks if number 'x' is prime
def check_prime(x):
    # Checks if the number is even, or equal to 1, or if its already in the
    # list of special numbers.
    if x % 2 == 0 or x == 1 or x in special_num:
        return False
    # Square root the number to reduce the time from O(n) to O(sqrt(n))
    sqrt_x = int(x ** 0.5)
    # Divides the number we are testing with every odd number up to root x
    for i in range(3, sqrt_x + 1, 2):
        if x % i == 0:
            return False
    return True

# This function creates all the palindromes
def palindrome_gen(m, n):
    # Takes the number of digits for n and half the value
    half = (len(str(n)))//2
    # If it is an odd number of digits, we need to +1 to the variable
'half'
    # There are no even digit palindromes
    if len(str(n)) % 2 != 0:
        half += 1
```

```
    # Create a list where each element represents each digit
    digits = [0 for i in range(int(half))]

    while digits[0] != 10:
        left_half = ""
        right_half = ""
        digits[-1] += 1

        # Checks each digit if it has reached its limit e.g. 9999
        for ref in range(len(digits)-1, -1, -1):
            if digits[0] == 10:
                break
            elif digits[ref] == 10:
                digits[ref - 1] += 1
                digits[ref] = 0

        # All the elements in 'digits' are combined into one integer
        # The right half of the number is flipped up until the 2nd to last
digit
        left_half = ''.join(str(x) for x in digits)
        left_half = int(left_half)
        right_half = str(left_half)[-2::-1]

        # Combines both halves to create the palindrome
        num = int(str(left_half) + str(right_half))
        # Check if the number is withing the ranges of m and n
        # as well as calling the check_prime() function
        if m <= num <= n and check_prime(num):
            special_num.append(num)

    return special_num


# Test Cases
test_cases = [
    [1, 2000],#test case 1
    [100, 10000],#test case 2
    [20000, 80000],#test case 3
    [100000, 2000000],#test case 4
    [2000000, 9000000],#test case 5
    [10000000, 100000000],#test case 6
    [100000000, 400000000],#test case 7
    [1100000000, 15000000000],#test case 8
    [15000000000, 100000000000],#test case 9
    [1, 1000000000000],#test case 10
]

test_num = 1
for test in test_cases:
    special_num = []
    # Hardcode the first 5 prime palindromes if m is smaller than 11
    if test[0] <= 11 <= test[1]:
        special_num = [2, 3, 5, 7, 11]
    start = time.time()
    palindrome_gen(test[0], test[1])

    print()
    print(f"Test Case {test_num}: ", test)
    print("total: ", len(special_num))
    if len(special_num) < 6:
        print(sorted(special_num))
```

```
else:
    # Print the first and last 3 special numbers
    print("First three: ", sorted(special_num[:3]))
    print("Last three: ", sorted(special_num[-3:]))

end = time.time()
test_num += 1
print(end-start, "seconds.")
print()
```

```
"""
Author: Majmundar, Tanisha Tapan [001296006]
"""

import time

def is_prime(num):
    if num < 2:
        return False
    if num == 2:
        return True
    if 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 generate_palindromes(max_digits):
    palindromes = [2, 3, 5]
    for num in range(1, 10 ** max_digits, 2):
        str_num = str(num)
        palindromes.append(int(str_num + str_num[-2::-1]))
    for num in range(1, 10 ** (max_digits - 1)):
        str_num = str(num)
        palindromes.append(int(str_num + str_num[::-1]))
    return palindromes

def find_palindromic_primes_in_range(start, end):
    start_time = time.time()
    max_digits = len(str(end))  # Determine the maximum number of digits in
the range
    palindromes = generate_palindromes(max_digits)
    palindromic_primes = [num for num in palindromes if start <= num <= end
and is_prime(num)]

    # Instead of generating all primes in the range, check for primality
dynamically
    primes = [num for num in range(start, end + 1) if is_prime(num)]

    end_time = time.time()
    execution_time = end_time - start_time
    return palindromic_primes, primes, execution_time

def get_input():
    start_range = int(input("Enter the start of the range: "))
    end_range = int(input("Enter the end of the range: "))
    return start_range, end_range

def print_results(palindromic_primes, primes, execution_time, start_range,
end_range):
    print(f"Palindromic Prime Numbers between {start_range} and
{end_range}: {palindromic_primes}")
    print(f"Prime Numbers between {start_range} and {end_range}: {primes}")
    print(f"Execution time: {execution_time} seconds")

def main():
    start_range, end_range = get_input()
    palindromic_primes, primes, execution_time =
find_palindromic_primes_in_range(start_range, end_range)
```

```
    print_results(palindromic_primes, primes, execution_time, start_range,
end_range)

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

```
"""
Author: Risma, Israt Jahan [001339277]
"""

import time

def generate_palindromes(max_digits):
    palindromes = [2, 3, 5, 7, 11]
    # the range start from 8 since we can skip till 11.
    for num in range(8, 10 ** max_digits,1): #incrementing by 1 in each
iteration
        str_num = str(num)
        palindromes.append(int(str_num + str_num[-2::-1])) #converts
palindromic string to an integer.
    return palindromes

def is_prime(num):
    if num < 2:
        return False
    if num == 2:
        return True
    if num % 2 == 0:
        return False
    for i in range(3, int(num ** 0.5) + 1,2): #range starts from 3,
increments by 2 to avoid even numbers.
        if num % i == 0:
            return False
    return True

def find_special_numbers(test_cases):
    for idx, (m, n) in enumerate(test_cases, start=1): # iterate over the
test_cases list along with an index idx.
        print(f"\nTest Case {idx}:")
        start_time = time.time()
        palindromes = generate_palindromes(6) #generates palindromic
numbers upto 6.
        special_numbers = [num for num in palindromes if m <= num <= n and
is_prime(num)] #checks palindromic prime within the range m,n.
        total_special_numbers = len(special_numbers)
        print(f"Total special numbers: {total_special_numbers}")
        if total_special_numbers < 6:
            print("Total special numbers:", special_numbers)
    # for numbers that are bigger than 6 only
        else:
            first_three_smallest = sorted(special_numbers)[:3]
            last_three_biggest = sorted(special_numbers)[-3:]
            print(first_three_smallest)
            print( last_three_biggest)
        end_time = time.time() #current time for the test case
        print(f"Time taken for Test Case {idx}: {end_time - start_time}
seconds") #taken time
    return total_special_numbers

# Test Cases
test_cases = [
    (1, 2000),#test case 1
    (100, 10000),#test case 2
    (20000, 80000),#test case 3
    (100000, 2000000),#test case 4
    (2000000, 9000000),#test case 5
    (10000000, 100000000),#test case 6
```

```
    (100000000, 400000000),#test case 7
    (1100000000, 15000000000),#test case 8
    (15000000000, 100000000000),#test case 9
    (1, 1000000000000),#test case 10
]

start_time = time.time()

total_special_numbers = find_special_numbers(test_cases)

end_time = time.time()
print(f"Total time taken: {end_time - start_time} seconds")
```

```
"""
Author: Toba, Sabiha Ahmed [001340510]
"""

import time

# finding the prime numbers
def is_prime(num):
    if num < 2:
        return False
    if num == 2 or 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 generate_palindromes(max_digits):
    # Initialize a list with the single-digit prime palindromes
    palindromes = [2, 3, 5]

    # Iterate through each possible number of digits
    for num in range(10, 10 ** max_digits):
        # Convert the number to a string for manipulation
        str_num = str(num)
        # Generate palindromic numbers by appending the reverse of the
number excluding the last digit
        palindromes.append(int(str_num + str_num[-2::-1]))

    # Iterate through each possible number of digits starting with the
number 13
    for num in range(13, 10 ** (max_digits)):
        # Convert the number to a string for manipulation
        str_num = str(num)
        # Generate palindromic numbers by appending the reverse of the
number
        palindromes.append(int(str_num + str_num[::-1]))
    return palindromes


def find_palindromic_primes_in_range(start, end):
    start_time = time.time()
    palindromes = generate_palindromes(6)
    palindromic_primes = [num for num in palindromes if start <= num <= end
and is_prime(num)]
    end_time = time.time()
    execution_time = end_time - start_time
    return palindromic_primes, execution_time


# Test Cases
test_cases = [
    (1, 2000),
    (100, 10000),
    (20000, 80000),
    (100000, 2000000),
```

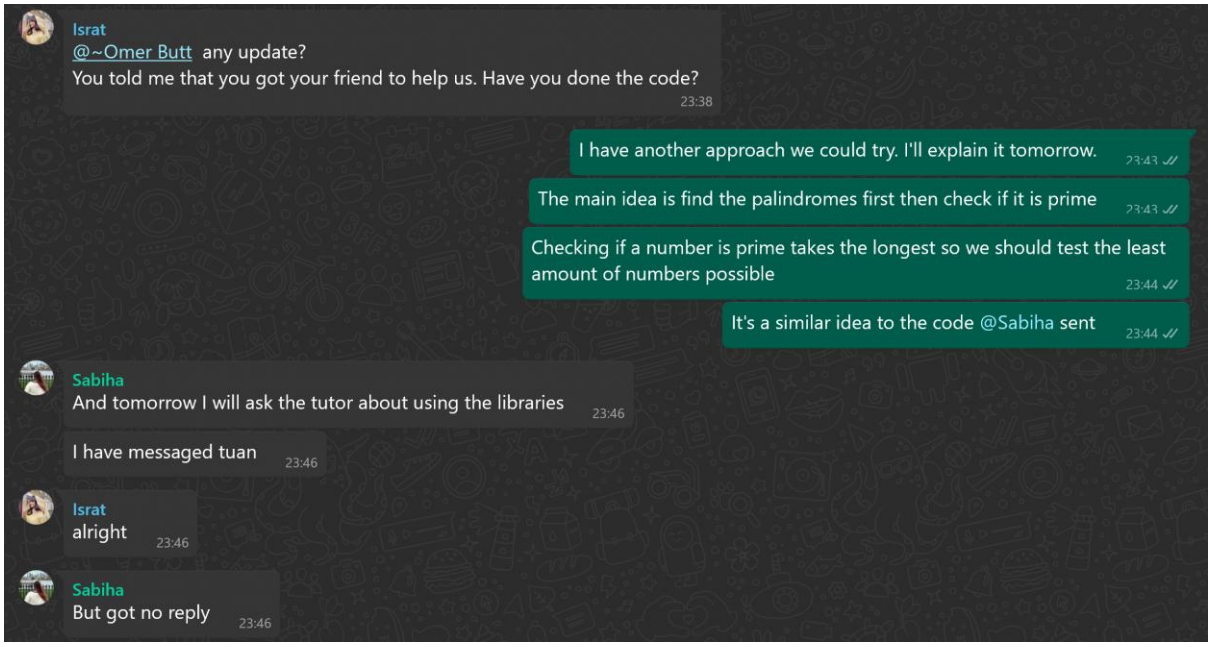
```
    (2000000, 9000000),
    (10000000, 100000000),
    (100000000, 400000000),
    (1100000000, 15000000000),
    (15000000000, 1000000000000),
    (1, 10000000000000),
]

# Run individual test cases
for i, (start_range, end_range) in enumerate(test_cases, 1):
    # print(f"Test Case {i}:")
    # print(f"Range: {start_range} to {end_range}")
    print(f"Test Case {i}:")
    print(f"Range: {start_range} to {end_range}")
    palindromic_primes, execution_time =
find_palindromic_primes_in_range(start_range, end_range)


    palindromic_primes.sort()


    total_special_palindromic_primes = len(palindromic_primes)

    if total_special_palindromic_primes > 0:

        # print(f"First 3 Special Palindromic Prime Numbers:
{palindromic_primes[:3]}")
        print(f"First 3 Special Palindromic Prime Numbers:
{palindromic_primes[:3]}")

        # print(f"Last 3 Special Palindromic Prime Numbers:
{palindromic_primes[-3:]}")
        print(f"Last 3 Special Palindromic Prime Numbers:
{palindromic_primes[-3:]}")
    else:
        print("No special palindromic prime numbers found in the specified
range.")
    print(f"Total Special Palindromic Primes:
{total_special_palindromic_primes}")
    print(f"Execution time: {execution_time} seconds")
    print()
```

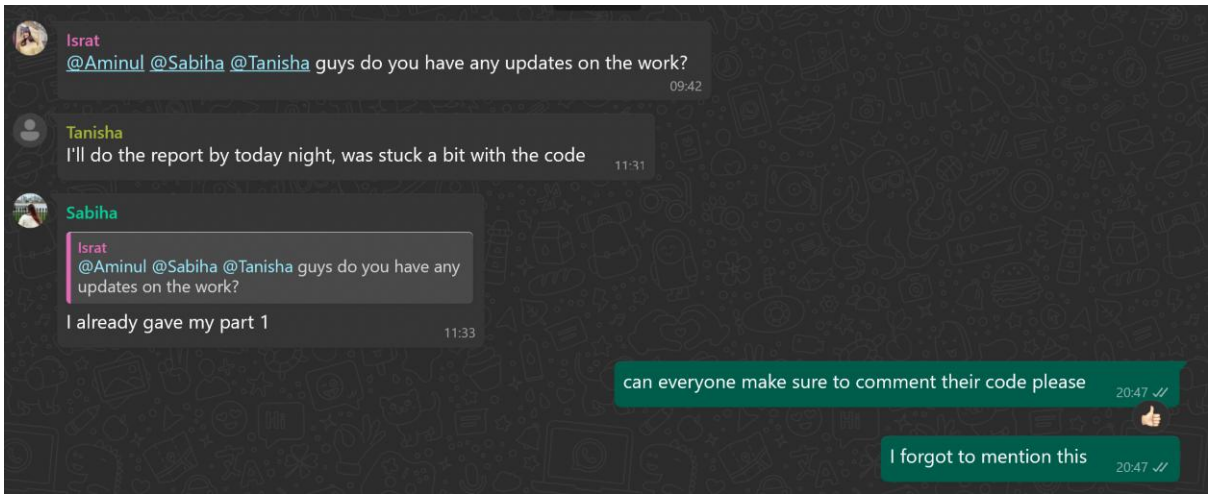
Appendix B - Test cases for correctness

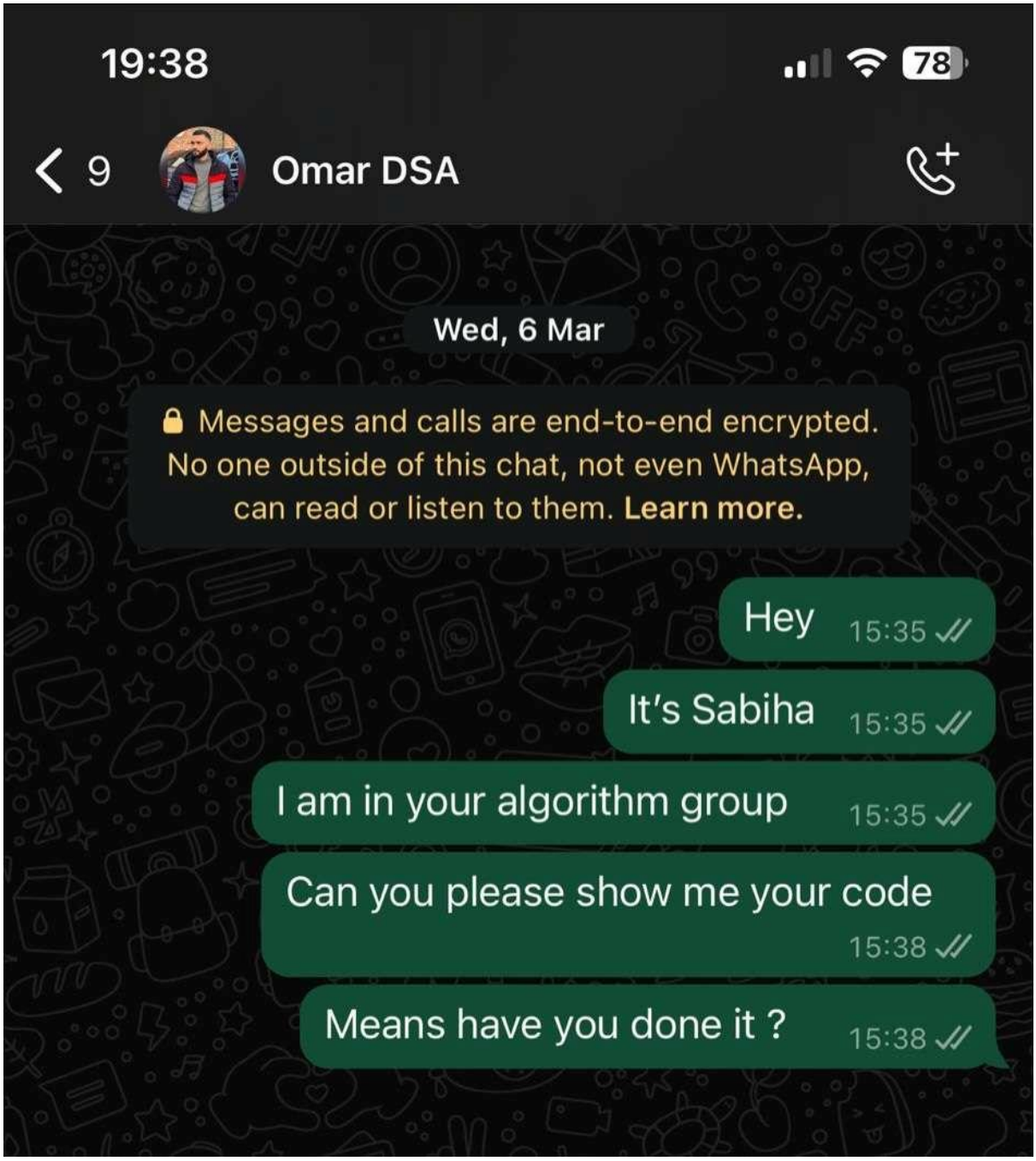
ID	Input	Output	Comments
1			
2			
3			
4			
5			
6			

Appendix C - Evidence of team contribution

Communication logs






19:38
78
9
Omar DSA
Wed, 6 Mar
Messages and calls are end-to-end encrypted. No one outside of this chat, not even WhatsApp, can read or listen to them. Learn more.
Hey 15:35
It's Sabiha 15:35
I am in your algorithm group 15:35
Can you please show me your code 15:38
Means have you done it ? 15:38