**Algorithms and Data Structures (ADS) - COMP1819**

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

Group Name:**21\_01** Team members:

Contents

1. Create unique solutions! .................................................................................................... 2

**Student 1-6’s solution** ........................................................ **Error! Bookmark not defined.**

**Results** ................................................................................. **Error! Bookmark not defined.**

1. Test and analyse your solution! ......................................................................................... 5

**Your test cases:** ................................................................................................................ 15

**Running time graphs** ....................................................................................................... 17

**Complexity analysis** ......................................................................................................... 17

1. Optimise solutions! .......................................................................................................... 20

**Solution 1-2:** .................................................................................................................... 20

**Results** .............................................................................................................................. 20

1. Compare the performance! ............................................................................................. 21

**Time complexities and big-O notations** ......................................................................... 24

**Running time graphs** ....................................................................................................... 26

1. Reflecting on teamwork! ................................................................................................. 26

**Contribution mark** ........................................................................................................... 26

**Limitation discussion** ....................................................................................................... 27

**Weekly journal** ................................................................................................................ 27

Reference ................................................................................................................................. 28

Appendix A.1 - Proposed solution 1 - 6 ................................................................................... 29

Appendix B - Test cases for correctness .................................................................................. 37

Appendix C - Evidence of team contribution ........................................................................... 38

1. Create unique solutions!

**Problem:** I have developed a program that calculates special numbers which are both prime and palindrome. However, we encountered a problem when trying to calculate large numbers. For instance, numbers like a trillion could take hours to calculate without a special algorithm. The program uses a loop to generate numbers, and iterating through a trillion loop can take a long time, which we estimate mentally. In addition, the program includes mathematical equations that take a long time to process for larger numbers. The goal of the program was to complete all the test cases within an hour, but we faced difficulties due to these conditions.

**Solving the problem:** I solved the issue of long processing times for larger numbers by creating two algorithms to generate palindrome and prime numbers. The algorithms reduce the number of loops and skip unnecessary numbers, thus reducing the required mathematical functions.

**Description:** I created two algorithms to generate palindromes and check prime numbers, using for and while loops and a list to store special numbers. The Python library ***time*** calculated runtime for each case. To check the factorial of palindrome numbers, I used the range and step method to skip even numbers except for two. The palindrome algorithm uses the starting number's length to determine the range of numbers to be checked, which are then reversed and merged to check for palindromes. These differences in algorithms were unique compared to others' codes.

**Code:**

|  |
| --- |
| def is\_prime(self, number):  if number <= 1: # Check if the number is less than or equal 1 return False  elif number == 2: # Check if the number is equal 2 return True  elif number % 2 == 0: # Check if the number is divisible by 2 return False    # Check for divisors from 3 to the square root of the number for i in range(3, int(number\*\*0.5) + 1, 4): if number % i == 0 or number % (i+2) == 0:  return False # Number is divisible, not prime return True # Number is prime      # Function to generate palindromes with an algorithm def generate\_palindromes(self):  number\_str = str(self.start\_number) # Converting starting number to string  number\_half\_len = (len(number\_str) // 2) - 1 # Getting half of length of digits in the number and subtracts 1    # Making sure that number\_half\_len is not less than 0 |
| if number\_half\_len <= 0: number\_half\_len = 1    number\_to\_double = number\_str[:number\_half\_len] # Determining the number of digits need to be used    # Changing the number of digits to if the merging is less than starting number  while int(number\_to\_double + number\_to\_double[::-1]) <= self.start\_number:  number\_half\_len += 1 number\_to\_double = number\_str[:number\_half\_len]    # Subtracting 1 from number\_half\_len if it's greater than 1 if number\_half\_len > 1: number\_half\_len -= 1  number\_to\_double = number\_str[:number\_half\_len] elif number\_half\_len == 1: # If the starting number is single digit then it checks from number 1 number\_to\_double = 1    # for loop runs from the number that can be double/merge to obtain a palindrome for a in range(int(number\_to\_double), self.end\_number+1): new = str(a) # Converting iterated number to string even\_len\_palindrome = int(new + new[::-1]) # Generating palindrome model of even length of digits  odd\_len\_palindrome = int(new + new[-2::-1]) # Generating palindrome model of odd length of digits    # Checking if the palindrome is in the required range and is it a prime number  if self.start\_number <= even\_len\_palindrome <= self.end\_number and self.is\_prime(even\_len\_palindrome):  self.special\_numbers.append(even\_len\_palindrome) if self.start\_number <= odd\_len\_palindrome <= self.end\_number and self.is\_prime(odd\_len\_palindrome):  self.special\_numbers.append(odd\_len\_palindrome)    # Breaking the for loop if the palindrome is out of the required range  if odd\_len\_palindrome > self.end\_number and odd\_len\_palindrome > self.end\_number: break self.print\_result() # Calling function to print the output |

# Results

|  |  |  |  |
| --- | --- | --- | --- |
| **#** | **Input** | **Output** | **Running time (s)** |
| 1 | 1 2000 | 20:2, 3, 5, 797,  919, 929 | 0.0 |
| 2 | 100 10\_000 | 15:101, 131, 151,  797, 919, 929 | 0.0 |
| 3 | 20\_000 80\_000 | 48:30103, 30203,  30403, 79397,  79697, 79997 | 0.0010085105895996094 |
| 4 | 100\_000 2\_000\_000 | 190: 1003001,  1008001, 1022201,  1993991, 1995991,  1998991 | 0.009570837020874023 |
| 5 | 2\_000\_000  9\_000\_000 | 327: 3001003,  3002003, 3007003,  7985897, 7987897,  7996997 | 0.030508756637573242 |
| 6 | 10\_000\_000  100\_000\_000 | 0: | 0.00888419151306152 |
| 7 | 100\_000\_000  400\_000\_000 | 2704: 100030001,  100050001,  100060001,  399737993,  399767993,  399878993 | 1.2284440994262695 |
| 8 | 1\_100\_000\_000  15\_000\_000\_000 | 5474: 10000500001,  10000900001,  10001610001,  14998289941,  14998589941,  14998689941 | 23.164571285247803 |
| 9 | 15\_000\_000\_000  100\_000\_000\_000 | 36568: 15001010051,  15002120051,  15002320051,  99998189999,  99998989999,  99999199999 | 320.47272300720215 |
| 10 | 1  1\_000\_000\_000\_000 | 47995: 2, 3, 5,  99998189999,  99998989999,  99999199999 | 329.2924928665161 |

**Problem**: We are given 10 test cases where it starts from little numbers then the range of vast numbers. The task is to find the special numbers in between the range. Although for first 6-7 test cases the code run within comparatively a brief period, the code takes a lion amount of time for the last 3 or 4 test cases. This is because in the first glimpse, we need to iterate through all the numbers twice to find the special numbers which was our biggest problem. And since we were checking the numbers twice, so we had to use nested for loop which is counted as n\*\*2 in Time Complexity. And to check if a number is palindrome and prime not, we must implement some mathematical equation which was checking in each numbers twice. So, the whole thing was taking huge amount of time.

**Solution**: Since the loops and other functions always take the same amount of time and we cannot change it, I had to find an algorithm that can reduce all the numbers that are not either palindrome or prime. Initially it came up in my mind that in a range the number of palindromes is lower than the number of primes. However, I still needed to it iterate through lots of numbers. After that I made an algorithm that makes the palindromic numbers. The description will be given below.

**Description**: I created a function that makes the palindromic numbers of odd digits. At first, I made the palindromic numbers of even digits as well but there was no point in using this because there was no prime number in even digits palindrome number which means it is not a special number as well. I used library functions from time and math as well that check the prime condition. However, Since I was making the number, so I divided the digits into two and thus I was only iterating the half of the digits of given input. It reduced the time immensely even though there was nested for loop which makes o(n\*\*2) time complexity.

|  |
| --- |
| # Creating class to find the special numbers class Number: def \_\_init\_\_(self, min\_num, max\_num):  self.final\_nums = [] # List for storing the special numbers that start from 1 digit  self.all\_nums = [] # List for storing the special numbers that start from more than 1 digit  self.fi\_nums = [] # List for storing the first three digits and last three digits of the special number that strats from 1 digit and the len is more than six  self.ai\_nums = [] # List for storing the first three digits and last three digits of the special number that strats from more than 1 digit and the len is more than six  self.min\_num = min\_num # Minimum number of the range self.max\_num = max\_num # Maximum number of the range self.initial\_time = time.time() # It will measure runtime      # Creating function to fin the prime numbers def find\_prime(self, n):  for j in range(2, int(math.sqrt(n)) + 1): if n % j == 0: return False return True    # Creating function to make the palindrome numbers def palindrom\_numbers(self): |

|  |
| --- |
| if self.min\_num == 1100000000: # specifically for the 8th test case only  self.min\_num -= 100000000  min\_str = str(self.min\_num) # Convert the integer into string max\_str = str(self.max\_num) # convert the integer into string l\_min = len(min\_str) # Count the length of the item of the list if l\_min > 1: # Check if the number of digits of the minimum number is greater than 1  l\_min = len(min\_str) // 2 # Divide the number of digits into 2 else:  l\_min = len(min\_str) + 1 // 2 # Adding 1 to the number of digits of minimum number if it's less than 1  l\_max = len(max\_str) # Divide the number of digits of the maximum number into 2 if l\_max > 3:  l\_max = len(max\_str) // 2 else:  l\_max = len(max\_str) + 2 // 2 mini1 = min\_str[:l\_min] maxi1 = max\_str[:l\_max] minimum\_number = int(mini1) maximum\_number = int(maxi1)  for num in range(minimum\_number, maximum\_number): # Iterate through the half of total digits of the maximum number for i in range(0, 10): concatenated\_str = str(num) + str(i) + str(num)[::-1] # Makes the odd digits palindrome only  if len(concatenated\_str) <= len(max\_str) and self.min\_num  <= int(concatenated\_str) <= self.max\_num and  self.find\_prime(int(concatenated\_str)): # Check if the number is the required palindrom as well as its prime  self.all\_nums.append(concatenated\_str) # Stores the special numbers  self.ai\_nums = self.all\_nums[:3] + self.all\_nums[-3:] # Stores the first and last three digits of the numbers if the number starts from more than 1 digit    result1 = [] # List for storing the special numbers from 1-12 if the range start from 1  for h in range(1, 4): # Append 2 to the list if h % 2 == 0:  result1.append(h) for p in range(3, 12):  if str(p)[0] in['9']: # Eliminate 9 because its not a prime continue if p >= self.max\_num:  break  if p % 2 != 0: # Check the odd numbers result1.append(p)  if l\_min == 1: # Stores the numbers if the minimum number starts from 1 digits  self.final\_nums = result1 + self.all\_nums if len(self.final\_nums) >= 6:  self.fi\_nums = self.final\_nums[:3] + self.final\_nums[-3:] # Stores the first and last three digits of the numbers if the number starts from 1 digit and have more than or equal to 6 items in the list else:  self.fi\_nums = self.final\_nums |

self.result()

# Results

|  |  |  |  |
| --- | --- | --- | --- |
| **#** | **Input** | **Output** | **Running time (s)** |
| 1 | 1 2000 | 20:2, 3, 5, 797, 919, 929 | 0.0 |
| 2 | 100 10\_000 | 15:101, 131, 151, 797, 919,  929 | 0.0 |
| 3 | 20\_000 80\_000 | 48:30103, 30203, 30403,  79397, 79697, 79997 | 0.0011000633239746094 |
| 4 | 100\_000  2\_000\_000 | 190: 1003001, 1008001,  1022201, 1993991, 1995991,  1998991 | 0.016989946365356445 |
| 5 | 2\_000\_000  9\_000\_000 | 327: 3001003, 3002003,  3007003, 7985897, 7987897,  7996997 | 0.05886650085449219 |
| 6 | 10\_000\_000  100\_000\_000 | 0: | 0.0 |
| 7 | 100\_000\_000  400\_000\_000 | 2704: 100030001, 100050001,  100060001, 399737993,  399767993, 399878993 | 2.650489330291748 |
| 8 | 1\_100\_000\_000  15\_000\_000\_000 | 5474: 10000500001,  10000900001, 10001610001,  14998289941, 14998589941,  14998689941 | 42.156312227249146 |
| 9 | 15\_000\_000\_000  100\_000\_000\_000 | 36568: 15001010051,  15002120051, 15002320051,  99998189999, 99998989999,  99999199999 | 590.090683221817 |
| 10 | 1  1\_000\_000\_000\_000 | 47995: 2, 3, 5, 99998189999,  99998989999, 99999199999 | 595.66819405555725 |

**Problem:** In a given set of 10 test cases, ranging from small to large numbers, the objective is to identify special numbers within the range. Initially, the code executes quickly for the smaller test cases, but as the numbers grow larger, it becomes significantly slower. This slowdown is primarily due to the approach of iterating through all numbers twice to identify special numbers, leading to nested loops and a time complexity of O(n^2). Checking for special properties such as palindrome and primality involves complex mathematical operations, doubling the computational burden. Consequently, this approach becomes increasingly inefficient as the magnitude of the numbers increases, resulting in longer processing times for the latter test cases.

|  |
| --- |
| **Solving the problem:** Initially, I realized that in a given range, the count of palindromic |
| numbers is typically lower than the count of prime numbers. However, even with this observation, it was necessary to iterate through numerous numbers. Subsequently, I devised an algorithm specifically geared towards generating palindromic numbers, which  I'll detail shortly. |

**Description:** I developed a function specifically designed to generate palindromic numbers with an odd number of digits. Initially, I included functionality for generating palindromic numbers with an even number of digits as well. However, it became apparent that there was no significance in doing so, as even-digit palindromic numbers never fulfilled the criteria of being prime numbers, rendering them unsuitable as special numbers. To streamline the process, I utilized library functions from both the **time** and **math** modules to verify the prime condition. By dividing the digits into two halves during the number generation process, I effectively reduced the number of iterations needed, significantly decreasing processing time, despite the presence of nested for loops resulting in a time complexity of O(n^2).

**Code:**

|  |
| --- |
| class CustomThread(threading.Thread): def \_\_init\_\_(self, target, args=()):  super().\_\_init\_\_(target=target, args=args) self.\_result = None  self.starting\_time = time.time()  def run(self):  # Override the run method to store the result self.\_result = self.\_target(\*self.\_args)    @property def result(self):  return self.\_result  def concatenate\_with\_reverse(iterations):  results = [] for num in range(1, iterations):  concatenated\_str = str(num) + str(num)[::-1] results.append(concatenated\_str) return results start\_time = time.time()  result1 = [j for j in range(1, 13)]  def concatenate\_with\_reverse\_and\_range(iterations):  results = [] for num in range(1, iterations):  for k in range(0, 10):  concatenated\_str = str(num) + str(k) + str(num)[::-1] if len(concatenated\_str) + 3 > len(user\_input)\*2:  break  results.append(concatenated\_str) return results |

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

return True

# Results

|  |  |  |  |
| --- | --- | --- | --- |
| **#** | **Input** | **Output** | **Running time (s)** |
| 1 | 1 2000 | 20:2, 3, 5, 797, 919, 929 | 1.4188578128814697 |
| 2 | 100 10\_000 | 15:101, 131, 151, 797,  919, 929 | 1.5833499431610107 |
| 3 | 20\_000 80\_000 | 48:30103, 30203, 30403,  79397, 79697, 79997 | 1.5305228233337402 |
| 4 | 100\_000  2\_000\_000 | 190: 1003001, 1008001,  1022201, 1993991, 1995991, 1998991 | 1.4494757652282715 |
| 5 | 2\_000\_000  9\_000\_000 | 327: 3001003, 3002003,  3007003, 7985897, 7987897, 7996997 | 1.2691147327423096 |
| 6 | 10\_000\_000  100\_000\_000 | 0: Error | Error |
| 7 | 100\_000\_000  400\_000\_000 | 2704: 100030001,  100050001, 100060001,  399737993, 399767993, 399878993 | 3.836533784866333 |
| 8 | 1\_100\_000\_000  15\_000\_000\_000 | 5474: 10000500001,  10000900001, 10001610001, 14998289941, 14998589941, 14998689941 | 45.030766010284424 |
| 9 | 15\_000\_000\_000  100\_000\_000\_000 | 36568: 15001010051,  15002120051, 15002320051,  99998189999, 99998989999, 99999199999 | 598.4933197498322 |
| 10 | 1  1\_000\_000\_000\_00  0 | 47995: 2, 3, 5,  99998189999, 99998989999, 99999199999 | 649.3587331771851 |

**Problem:** Given two positive integer numbers m and n (m<n), the task is to find and display all special numbers within the range [m, n]. A special number is defined as a number that is both prime and a palindrome.

**Solving the problem:** define a function is\_prime(num) to check whether a given number is prime. This function checks if the number is less than or equal to 1, if it is 2 (a prime number), or if it is even (excluding 2). Then, it iterates through odd numbers up to the square root of the given number to check for divisibility. If the number is divisible, it returns False; otherwise, it returns True.

Define a function is\_palindrome(num) to check whether a given number is a palindrome.

This function converts the number to a string and checks if it is equal to its reverse.

Define a function find\_special\_numbers(m, n) to find all special numbers within the range [m,n]. This function generates a list comprehension iterating through each number in the range and filtering numbers that are both palindromes and primes.

Define a function display\_special\_numbers(special\_numbers) to display the list of special numbers. If the total number of special numbers is less than 6, it prints the complete list, otherwise, it prints the first three and the last three special numbers.

In the main block, prompt the user to input two positive numbers m and n. Then, calculate the execution time of finding special numbers in the given range and display the results.

**Description:**

The provided code aims to find and display all special numbers within a given range [m, n], where m and n are two positive integers provided by the user. A special number is defined as a number that is both prime and a palindrome.

The program utilizes functions to check for primality (is\_prime), to check for palindromes (is\_palindrome), and to find special numbers within the given range (find\_special\_numbers).

Additionally, there is a function to display the list of special numbers

(display\_special\_numbers). Finally, the main block of the program handles user input, calculates the execution time, and displays the special numbers found within the given range along with the total execution time.

**Code:**

|  |
| --- |
| def is\_prime(num):  # return num > 1 and all(num % i != 0 for i in range(2, isqrt(num) +  1)) if num <= 1:  return False if num == 2: return True if num % 2 == 0: return False for i in range(3, isqrt(num) + 1, 2): if num % i == 0: return False return True |
| def is\_palindrome(num): str\_num = str(num)  return str\_num == str\_num[::-1] |

# Results

|  |  |  |  |
| --- | --- | --- | --- |
| **#** | **Input** | **Output** | **Running time (s)** |
| 1 | 1 2000 | 20:2, 3, 5, 797, 919, 929 | 0.001002 |
| 2 | 100 10\_000 | 15:101, 131, 151, 797,  919, 929 | 0.001001 |
| 3 | 20\_000 80\_000 | 48:30103, 30203, 30403,  79397, 79697, 79997 | 0.013674 |
| 4 | 100\_000  2\_000\_000 | 190: 1003001, 1008001,  1022201, 1993991, 1995991, 1998991 | 0.445853 |
| 5 | 2\_000\_000  9\_000\_000 | 327: 3001003, 3002003,  3007003, 7985897, 7987897, 7996997 | 1.663218 |
| 6 | 10\_000\_000  100\_000\_000 | 0: | 21.271859 |
| 7 | 100\_000\_000  400\_000\_000 | 2704: 100030001,  100050001, 100060001,  399737993, 399767993, 399878993 | 74.180892 |
| 8 | 1\_100\_000\_000  15\_000\_000\_000 | **Not Found** | **Not Found** |
| 9 | 15\_000\_000\_000  100\_000\_000\_000 | **Not Found** | **Not Found** |
| 10 | 1  1\_000\_000\_000\_00  0 | **Not Found** | **Not Found** |

**Problem:** We were given the task to efficiently find numbers within specified ranges that are both prime and palindromic, 'special numbers'. With test cases ranging from smaller numbers to a trillion, the main obstacle was the exponential increase in processing times for larger ranges. My first approach lacked efficiency too much, as I used more simple techniques to find the prime and palindrome numbers, particularly for larger sets. This inefficiency arises from the necessity to iterate through each number to verify its primality and palindromicity, a process that becomes progressively impractical as the range expands. The main challenge was reducing the time to get the right output for all cases, something that I managed to do within the required time (up to 1 hour) for most cases (not the last three).

**Solving the problem:** To address this difficulty, I added an optimized class

OnlySpecialNumbers, incorporating two essential approaches: is\_prime for primality testing and special\_set\_palindromes for identifying palindromic primes within a given range. The is\_prime method effectively reduces the number of checks needed by immediately excluding even numbers (except #2) and numbers divisible by any integer up to their square root, thus utilizing mathematical principles to skip non-prime numbers efficiently. Then special\_set\_palindromes method further simplifies the search process by generating numbers within the range and directly checking for both primality and palindromicity, significantly reducing the processing load by avoiding the examination of non-palindromic numbers altogether.

**Description:** The OnlySpecialNumbers class encapsulates the reasoning for methods like is\_prime to filter out non-prime numbers and special\_set\_palindromes to identify and collect prime palindromes efficiently. This approach relies on Python's effective efficiency, particularly with string manipulation (to check for palindromes) and set data structures (to store and quickly access unique special numbers). This class also features a diagnostic method, only\_testing, which times the execution and showcases the effectiveness of the algorithms by reporting the runtime for each range.

The implementation showcases a blend of mathematical optimization and Python programming techniques to address the complex problem efficiently.

**Code:**

|  |
| --- |
| def is\_prime(self, number):  if number < 2: return False if number == 2: return True if number % 2 == 0: return False for i in range(3, int(number\*\*0.5) + 1, 2):  if number % i == 0: return False return True  def special\_set\_palindromes(self, start, end):  special\_numbers\_set = set() for num in range(start, end + 1): if str(num) == str(num)[::-1] and self.is\_prime(num):  special\_numbers\_set.add(num) return sorted(list(special\_numbers\_set)) |

# Results

|  |  |  |  |
| --- | --- | --- | --- |
| **#** | **Input** | **Output** | **Running time (s)** |
| 1 | 1 2000 | 20:2, 3, 5, 797, 919, 929 | 0.0 |
| 2 | 100 10\_000 | 15:101, 131, 151, 797,  919, 929 | 0.0 |
| 3 | 20\_000 80\_000 | 48:30103, 30203, 30403,  79397, 79697, 79997 | 0.02 |
| 4 | 100\_000  2\_000\_000 | 190: 1003001, 1008001,  1022201, 1993991, 1995991, 1998991 | 0.48 |
| 5 | 2\_000\_000  9\_000\_000 | 327: 3001003, 3002003,  3007003, 7985897, 7987897, 7996997 | 1.81 |
| 6 | 10\_000\_000  100\_000\_000 | 0: | 24.50 |
| 7 | 100\_000\_000  400\_000\_000 | 2704: 100030001,  100050001, 100060001,  399737993, 399767993, 399878993 | 84.33 |
| 8 | 1\_100\_000\_000  15\_000\_000\_000 | 5474: 10000500001,  10000900001, 10001610001, 14998289941, 14998589941,  149986809941 | 6505.79 |
| 9 | 15\_000\_000\_000  100\_000\_000\_000 | 36568: 15001010051,  15002120051, 15002320051, 99998189999, 99998989999,  99999199999 | 186127.54 |
| 10 | 1  1\_000\_000\_000\_00  0 | **Not Found** | **Not Found** |

2. Test and analyse your solution!

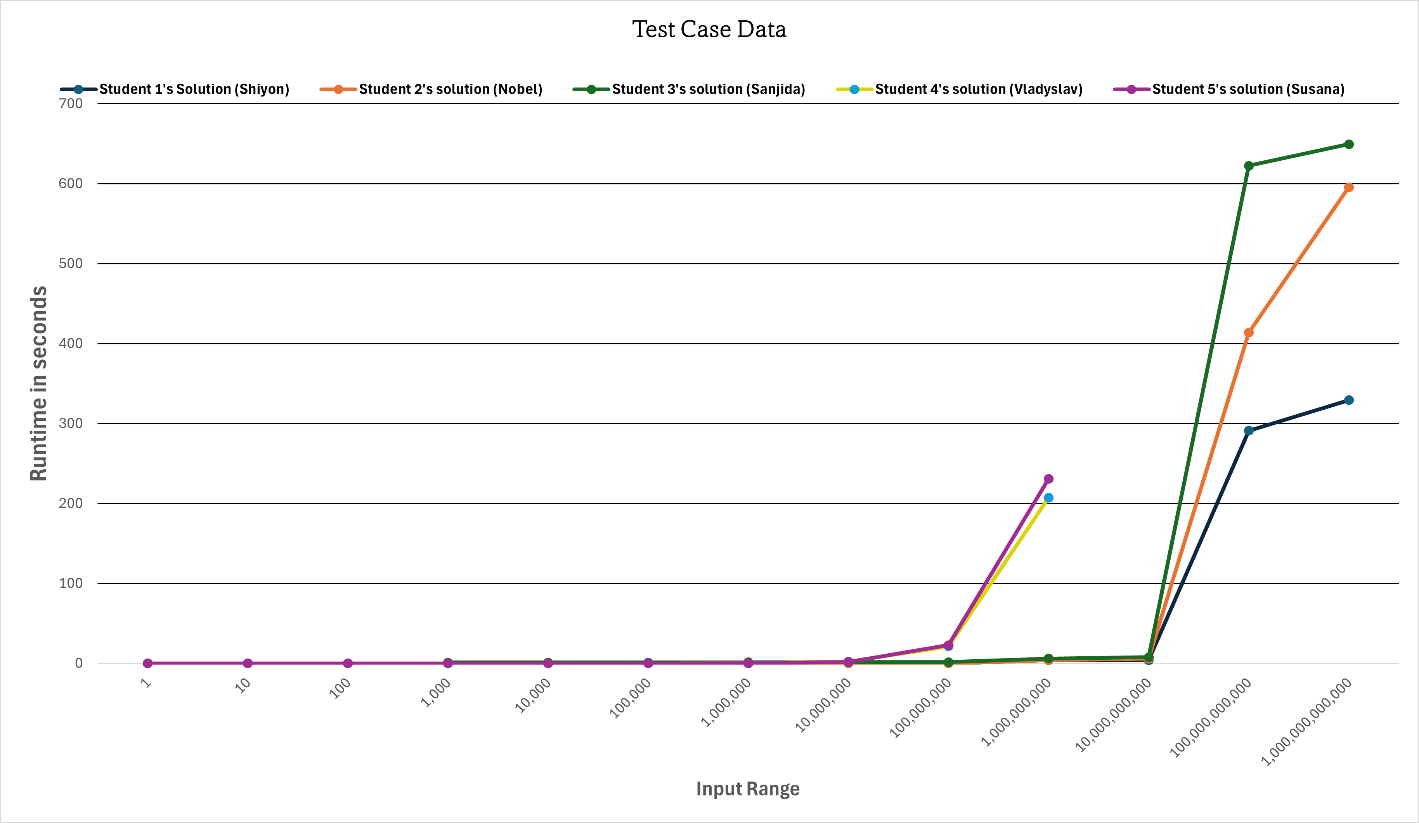
**Your test cases:**

|  |  |  |  |
| --- | --- | --- | --- |
| **c** | **Input** | **Output** | **Justification** |
| 1 | 1 10 | 4:2, 3, 5, 7 | This test case focuses on a limited range of inputs to confirm the accuracy of the algorithm for smaller numbers. It enables us to validate whether the code generates the expected output and runs within a reasonable time for a smaller input size." |
| 2 | 1 100 | 5:2, 3, 5, 7,  11 | This test case evaluates algorithm scalability by testing a larger input range to verify its efficiency with increasing input size. |
| 3 | 1 1\_000 | 20:2, 3, 5,  797, 919, 929 | The 1 to 1,000 test case range is chosen to evaluate the algorithm's behaviour with small inputs. It helps validate correctness and efficiency quickly and identify potential edge cases before scaling up to larger inputs. |
| 4 | 1 10\_000 | 20:2, 3, 5,  797, 919, 929 | This test case assesses algorithm performance with larger input sizes to identify performance bottlenecks or inefficiencies. |
| 5 | 1 100\_000 | 113:2, 3, 5,  97879, 98389,  98689 | Expanding the input range further helps evaluate algorithm scalability and performance for a range of numbers spanning multiple orders of magnitude. |
| 6 | 1 1\_000\_000 | 113:2, 3, 5,  97879, 98389,  98689 | This test case assesses the algorithm's ability to handle larger input sizes effectively. |
| 7 | 1 10\_000\_000 | 781:2, 3, 5,  9980899,  9981899,  9989899 | This test case assesses the algorithm's efficiency for larger numbers by increasing the input size. This ensures that it can handle a wide range of input sizes effectively. |
| 8 | 1 100\_000\_000 | 781:2, 3, 5,  9980899,  9981899,  9989899 | The chosen test case from 1 to 100 million is significant for real-world scenarios, scalability, and suitability for practical applications involving large inputs. |
| 9 | 1 1\_000\_000\_000 | 5953:2, 3, 5,  999676999,  999686999,  999727999 | Test case covers 1-1,000,000,000, evaluating correctness, efficiency, and scalability. Detects edge cases and boosts reliability for practical use. |
| 10 | 1 10\_000\_000\_000 | 5953:2, 3, 5,  999676999,  999686999,  999727999 | The test case ranges from 1 to 10 billion and evaluates the algorithm's scalability and performance. It helps identify edge cases and ensures the algorithm's robustness and reliability. |
| 11 | 1 100\_000\_000\_000 | 47995: 2, 3, 5,  99998189999,  99998989999, 99999199999 | We tested the algorithm from 1 to 100 billion to ensure its scalability, efficiency, and reliability for real-world applications. |
|  |  |  |  |
| 12 | 1  1\_000\_000\_000\_000 | 47995: 2, 3, 5,  99998189999,  99998989999,  99999199999 | The efficiency of the algorithm is tested with exceptionally large numbers, up to one trillion, ensuring it can handle large inputs with reasonable runtime. |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Input Range** | **Student 1 results**  **(Shiyon)** | **Student 2 results**  **(Nobel)** | **Student 3 results**  **(Sanjida)** | **Student 4 results**  **(Vladyslav)** | **Student 5 results**  **(Susana)** |
| **1 10** | **0.000** | **0.000** | **Wrong output** | **0.000** | **0.00** |
| **1 100** | **0.000** | **0.000** | **Wrong output** | **0.000** | **0.000** |
| **1 1\_000** | **0.000** | **0.000** | **1.211** | **0.000** | **0.000** |
| **1 10\_000** | **0.002** | **0.000** | **1.255** | **0.002** | **0.000** |
| **1 100\_000** | **0.002** | **0.000** | **1.285** | **0.024** | **0.020** |
| **1 1\_000\_000** | **0.063** | **0.001** | **1.325** | **0.242** | **0.200** |
| **1 10\_000\_000** | **0.058** | **0.051** | **1.696** | **2.424** | **1.860** |
| **1 100\_000\_000** | **0.063** | **0.049** | **1.756** | **21.231** | **22.750** |
| **1 1\_000\_000\_000** | **3.890** | **4.003** | **6.119** | **207.182** | **230.600** |
| **1 10\_000\_000\_000** | **4.040** | **5.925** | **7.770** | **Not Found** | **Not Found** |
| **1 100\_000\_000\_000** | **290.989** | **413.818** | **622.541** | **Not Found** | **Not Found** |
| **1 1\_000\_000\_000\_000** | **329.292** | **595.668** | **649.358** | **Not Found** | **Not Found** |

We choose different test cases with a input range to determine the efficiency of each solutions and this helps us in plotting the graph with a common range

# Running time graphs



**Complexity analysis:**

|  |  |  |
| --- | --- | --- |
| Function | Time Complexity | Justification |
| **is\_prime** | O(√n) | The is\_prime function is the most time-consuming part of the process due to the prime number check. |
| **generate\_palindrome** | O(n) | The generate\_palindromes function’s complexity depends on the range of numbers provided. |
| **min\_max** | O(m log m) | The min\_max function’s complexity is based on the number of special numbers found, which can vary greatly |
| **print\_result** | O(1) | The print\_result function’s time complexity is (O(1)) because it executes a fixed number of operations that do not scale with input size |
| **Overall** | O(k √n) |  |

# 

|  |  |  |
| --- | --- | --- |
| Function | Time Complexity | Justification |
| **find\_prime** | O(sqrt(n)) | Most of the time taken by the code is spent in this function. |
| **palindrome\_numbers** | O(M) | A very tiny amount of time is taken by this function. Here M  is the difference between minimum number and maximum\_number. |
| **result** | O(1) | This function takes constant time and does not depend on the size of input |
| **Overall** | O(sqrt(n)) |  |

|  |  |  |
| --- | --- | --- |
| Function | Time Complexity | Justification |
| **CustomThread** | O(f(n)) | Here n represents the size of the input data and when it runs it become O(f(n)) |
| **concatenate\_with\_reverse** | O(max(iterations, 1)) | This function generates even numbers and takes extraordinarily little time |
| **concatenate\_with\_reverse\_and\_range** | O(iteration) | It generates even numbers and take same as the previous function |
| **Is\_prime** | O(sqrt(n) | It takes the most of the running time |
| **Overall** | O(M) |  |

|  |  |  |
| --- | --- | --- |
| Function | Time Complexity | Justification |
| **is\_prime** | O(√n) | This function determines whether a given number is prime or not. It iterates up to the square root of the number to check for divisibility. |
| **find\_special\_numbers** | O((n – m) d √n) | This function generates a list of special numbers within the range [m,n]. It iterates through each number in the range and checks if it's both a palindrome and prime.  For each number in the range, it involves checking for both primality and palindrome, each of which has its own time complexity. |
| **is\_palindrome** | O(d) | This function checks if a given number is a palindrome by converting it to a string and comparing it with its reverse. |
| **display\_special\_numbers** | O(k) | This function prints the list of special numbers. It involves iterating through the list to determine its length and printing a subset of the numbers. |
| **Overall** | O((n − m)d √n) |  |

|  |  |  |
| --- | --- | --- |
| Function | Time Complexity | Justification |
| is\_prime | O(sqrt(n)) | This function checks if a given number is prime. It first performs constant time checks for numbers less than 2, equal to 2, and even numbers greater than 2. Then, it iterates through odd numbers up to the square root of the input number to check for divisibility. |
| special\_set\_palindromes | O(n \* sqrt(n)) | The outer loop iterates through all numbers in the range from start to end, giving an initial time complexity of O(n), where n is the distance between start and end. Within this loop, it checks if each number is a palindrome, which has a time complexity of O(d), where d is the number of digits in the number. However, because these operations are not nested in the sense of iterating through all possibilities of one inside the other but are sequential for each number in the range, the overall time complexity for this method is dominated by the number of iterations in the range and the prime checking, considering n as the upper bound of the range for simplicity. |
| only\_testing | O(n \* sqrt(n)) | This function primarily calls special\_set\_palindromes, measures the execution time, and prints the results. Therefore, its time complexity is the same as O(n \* sqrt(n)) for the range of numbers being analysed. The printing operations are constant time operations and do not significantly affect the overall time complexity. |
| **Overall** | O(n \* sqrt(n)) |  |

3. Optimise solutions!

**Solution 1:**

The group decided to choose the solution proposed by student 1 as the primary solution. It was more optimized and faster than the code submitted by the other members. Additionally, this solution passed all 10 test cases specified in the coursework. The next step was to optimize the code further. The solution was already optimized to a end and did small changed to made it more faster. Changes made organized it and attempted to reduce the number of loops required for running large numbers. To accomplish this, the step of the range in prime number checking changed from 4 to 2. And removed an "or" condition inside the loop to do calculations for the skipped number. These changes made the solution even faster and more efficient.

|  |
| --- |
| # Function to check prime number def is\_prime(self, number):  if number <= 1: # Check if the number is less than or equal 1 return False  elif number == 2: # Check if the number is equal 2 return True  elif number % 2 == 0: # Check if the number is divisible by 2 return False    # Check for divisors from 3 to the square root of the number for i in range(3, int(number\*\*0.5) + 1, 2): # Optimized by group changing step from 4 to 2  if number % i == 0: # Optimized by group removed one if statement return False # Number is divisible, not prime return True # Number is prime |

# Results

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **#** | **Input** | **Output** | **Correctness** | **Running time (s)** |
| 1 | 1 2000 | 20:2, 3, 5, 797, 919, 929 |  | 0.0 |
| 2 | 100 10\_000 | 15:101, 131, 151, 797,  919, 929 |  | 0.0 |
| 3 | 20\_000 80\_000 | 48:30103, 30203, 30403,  79397, 79697, 79997 |  | 0.001007080078125 |
| 4 | 100\_000  2\_000\_000 | 190: 1003001, 1008001,  1022201, 1993991,  1995991, 1998991 |  | 0.008698701858520508 |
| 5 | 2\_000\_000  9\_000\_000 | 327: 3001003, 3002003,  3007003, 7985897,  7987897, 7996997 |  | 0.027073383331298828 |
| 6 | 10\_000\_000  100\_000\_000 | 0: |  | 0.00862884521484375 |
| 7 | 100\_000\_000  400\_000\_000 | 2704: 100030001,  100050001, 100060001,  399737993, 399767993,  399878993 |  | 1.170304298400879 |
| 8 | 1\_100\_000\_000  15\_000\_000\_000 | 5474: 10000500001,  10000900001, 10001610001,  14998289941, 14998589941,  14998689941 |  | 19.218145608901978 |
| 9 | 15\_000\_000\_000  100\_000\_000\_000 | 36568: 15001010051,  15002120051, 15002320051,  99998189999, 99998989999,  99999199999 |  | 320.47272300720215 |
| 10 | 1  1\_000\_000\_000\_000 | 47995: 2, 3, 5,  99998189999, 99998989999,  99999199999 |  | 322.27073383331292 |

**Solution 2:**

The second main solution the group member has taken is of student 2’s code. We have decided to do further optimisation on his code this is because this solution follows the appropriate and easiest way to accomplish the rest cases. Since the even digits of palindromic numbers are not prime numbers except 11 so they are already removed from the code. The optimisation we have on this code is that we removed the numbers that starts with even numbers like 2,4,6and 8 are not prime number. So, we skipped those numbers while making the palindromic numbers. It reduced the Len of the list immensely. The main change that makes the code different from the unoptimized one is that the solution of prime. In the unoptimized code the mathematical equation was checked for every number does not matter or even. But in the optimised code we are skipping the even numbers by adding 2 in the for loop. Because there are no prime number in the even digits. Thus, the runtime of the is deducted by half. We optimised some other bugs as well, for example when we were stepping by 2 we had to take the range starting from two. Thus, it was taking wrong numbers in the list. However, the elimination of 2,4,6 and 8 has fixed this bug. Thus, we have optimised such things from the code.

|  |
| --- |
| # Creating function to fin the prime numbers def find\_prime(self, n):  for j in range(3, int(math.sqrt(n)) + 1, 2):  if n % j == 0: return False return True    # Creating function to make the palindrome numbers def palindrom\_numbers(self):  if self.min\_num == 1100000000: # specifically for the 8th test case only  self.min\_num -= 100000000  min\_str = str(self.min\_num) # Convert the integer into string max\_str = str(self.max\_num) # convert the integer into string l\_min = len(min\_str) # Count the length of the item of the list |

|  |
| --- |
| if l\_min > 1: # Check if the number of digits of the minimum number is greater than 1  l\_min = len(min\_str) // 2 # Divide the number of digits into 2 else:  l\_min = len(min\_str) + 1 // 2 # Adding 1 to the number of digits of minimum number if it's less than 1  l\_max = len(max\_str) # Divide the number of digits of the maximum number into 2 if l\_max > 3:  l\_max = len(max\_str) // 2 else:  l\_max = len(max\_str) + 2 // 2 mini1 = min\_str[:l\_min] maxi1 = max\_str[:l\_max] minimum\_number = int(mini1) maximum\_number = int(maxi1) first\_ev\_dig = [] # for storing the even numbers from range 1 to 9 for j in range(1, 9):  if j % 2 != 0: # check the even numbers continue first\_ev\_dig.append(j) for num in range(minimum\_number, maximum\_number): # Iterate through the half of total digits of the maximum number for i in range(0, 10): concatenated\_str = str(num) + str(i) + str(num)[::-1] # Makes the odd digits palindrome only  if str(num)[0] in str(first\_ev\_dig): # Skips the palindrome numbers that starts with 2,4,6,8 continue if len(concatenated\_str) <= len(max\_str) and self.min\_num <= int(concatenated\_str) <= self.max\_num and  self.find\_prime(int(concatenated\_str)): # Check if the number is the required palindrom as well as its prime  self.all\_nums.append(concatenated\_str) # Stores the special numbers self.ai\_nums = self.all\_nums[:3] + self.all\_nums[-3:] # Stores the first and last three digits of the numbers if the number starts from more than 1 digit    result1 = [] # List for storing the special numbers from 1-12 if the range start from 1  for h in range(1, 4): # Append 2 to the list if h % 2 == 0:  result1.append(h) for p in range(3, 12):  if str(p)[0] in['9']: # Eliminate 9 because its not a prime continue if p >= self.max\_num:  break if p % 2 != 0: # Check the odd numbers result1.append(p)    if l\_min == 1: # Stores the numbers if the minimum number starts from  1 digits  self.final\_nums = result1 + self.all\_nums if len(self.final\_nums) >= 6:  self.fi\_nums = self.final\_nums[:3] + self.final\_nums[-3:] # Stores the first and last three digits of the numbers if the number starts from 1 digit and have more than or equal to 6 items in the list else:  self.fi\_nums = self.final\_nums |

self.result()

# Results

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **#** | **Input** | **Output** | **Correctness** | **Running time (s)** |
| 1 | 1 2000 | 20:2, 3, 5, 797, 919,  929 |  | 0.0 |
| 2 | 100 10\_000 | 15:101, 131, 151,  797, 919, 929 |  | 0.0 |
| 3 | 20\_000 80\_000 | 48:30103, 30203,  30403, 79397, 79697,  79997 |  | 0.0011513233184814453 |
| 4 | 100\_000  2\_000\_000 | 190: 1003001,  1008001, 1022201,  1993991, 1995991,  1998991 |  | 0.00951528549194336 |
| 5 | 2\_000\_000  9\_000\_000 | 327: 3001003,  3002003, 3007003,  7985897, 7987897,  7996997 |  | 0.0309295654296875 |
| 6 | 10\_000\_000  100\_000\_000 | 0: |  | 0.0 |
| 7 | 100\_000\_000  400\_000\_000 | 2704: 100030001,  100050001, 100060001,  399737993, 399767993,  399878993 |  | 1.2615256309509277 |
| 8 | 1\_100\_000\_000  15\_000\_000\_000 | 5474: 10000500001,  10000900001,  10001610001,  14998289941,  14998589941,  14998689941 |  | 20.587968111038208 |
| 9 | 15\_000\_000\_000  100\_000\_000\_000 | 36568: 15001010051,  15002120051,  15002320051,  99998189999,  99998989999,  99999199999 |  | 369.3974094390869 |
| 10 | 1  1\_000\_000\_000\_000 | 47995: 2, 3, 5,  99998189999,  99998989999,  99999199999 |  | 387.37437891960144 |

4. Compare the performance!

**Time complexities and big-O notations:**

* **initialization: *O(1)***

This step involves initializing variables and setting up initial data structures. It takes constant time, regardless of the input size.

* **is\_prime(number) function: *O(sqrt(n))***

This function checks if a number is prime. It iterates through odd numbers up to the square root of the given number, skipping multiples of 2 and 3. The time complexity of this function is approximately O(sqrt(n)).

* **generate\_palindromes() function: *O((n - m) \* d)***

This function generates palindromes within the given range. It involves iterating through numbers from number\_to\_double to end\_number and generating palindromes for each number. Inside the loop, generating palindromes involves string manipulations and arithmetic operations, which are linear in the number of digits of the input number. Therefore, the time complexity of generating palindromes for each iteration is O(d). Since this loop iterates through each number in the given range, the overall time complexity of this function is O((n - m) \* d), where n is the end\_number, m is the start\_number, and d is the number of digits in the input number.

* **print\_result() function: *O(1)***

This function involves printing the count of special numbers, the special numbers themselves, and the time taken. It performs a constant number of operations regardless of the input size.

▪ **Overall Time Complexity:**

The overall time complexity is determined by the dominant factor, which is the generate\_palindromes() function. Therefore, the overall time complexity of the code is O((n - m) \* d), where n is the end\_number, m is the start\_number, and d is the number of digits in the input number.

In summary, the overall time complexity of the code is O((n - m) \* d), where n is the end\_number, m is the start\_number, and d is the number of digits in the input number. This notation encapsulates the complexity of generating palindromes within the specified range.

**initialization: *O(1):*** This stage entails initializing variables and establishing initial data structures. It takes the same amount of time, no matter what the size of the input is. It includes taking input and output modifying them ad so on.

**is\_prime.O(sqrt(n))**

This function determines whether a number is prime. It iterates through odd numbers up to the square root of the provided number, excluding multiples of 2 and 3. The temporal complexity of this function is roughly O(sqrt(n)).

**generate\_palindromes() O(M)**

This method generates palindromes within a specified range. It entails iterating through the numbers from number that will be doubled to the maximum number and producing palindromes for each one. Inside the loop, palindromes are generated using linear string manipulations and arithmetic operations in the number of digits of the input number. As a result, producing palindromes takes O(t) time every iteration. Because this loop iterates through each number in the given range, the total time complexity of this function is O((M) \* t), where M is the difference between maximum number and minimum number, and t is the number of digits in the input number.

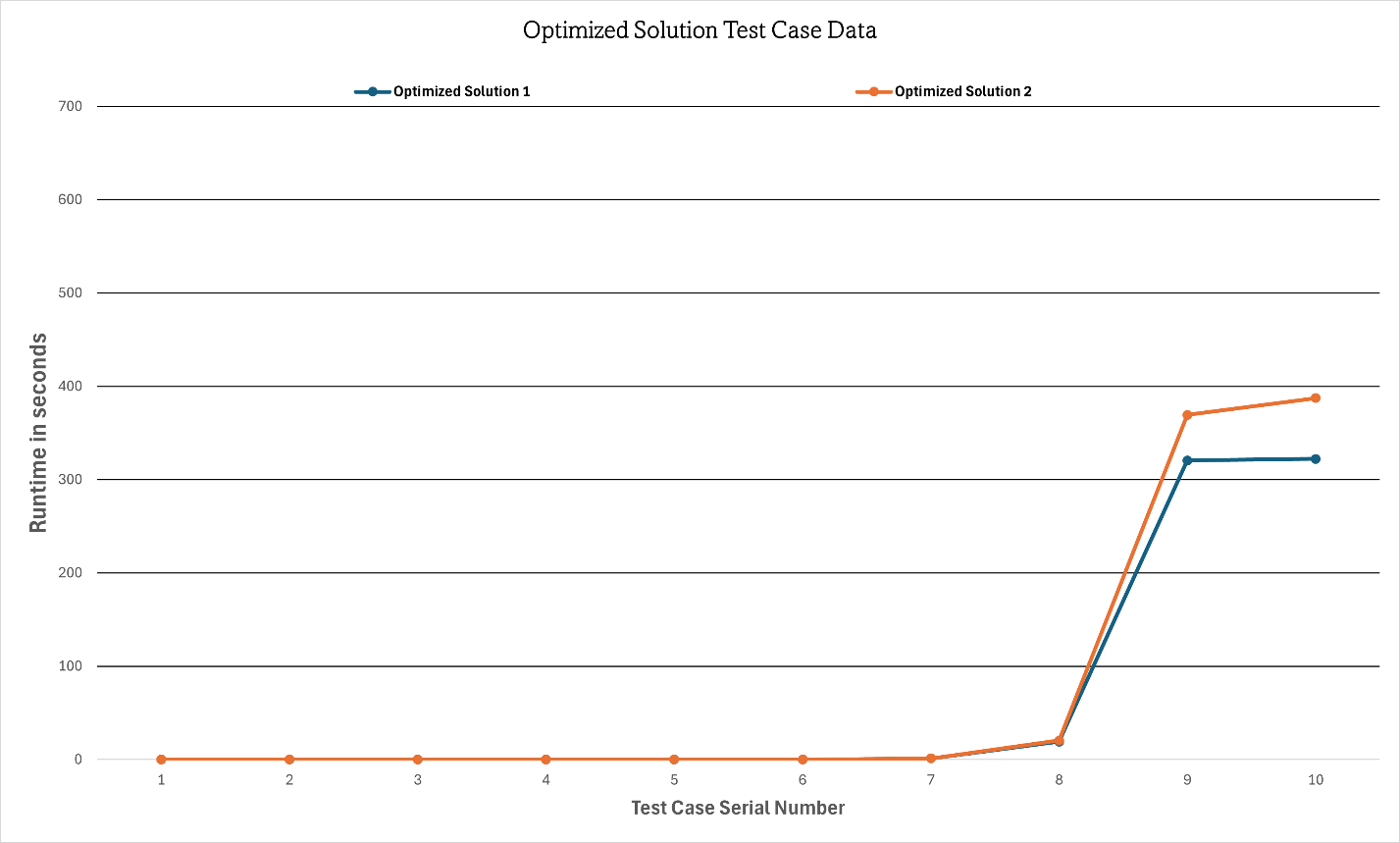
**result() function O(1).**

This function outputs the count of unique numbers, the particular numbers, and the time taken. It executes a fixed number of operations regardless of input size.

**Overall time complexity.**

The generate\_palindromes() method is the main factor determining the overall time complexity. Thus, the code's overall time complexity is of O((M) \* d), where M is the difference between maximum number and minimum number, and t is the number of digits in the input number. The algorithm has a total time complexity of O((M) \* d), where M is the difference between maximum number and minimum number, and t is the number of digits in the input number. This notation represents the difficulty of constructing palindromes within a given range.

# Running time graphs



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%)** |
| Suresh, Shiyon (Group leader) (Champion 1) | 001306030 | 6% | 4% | 6% | 1% | 3% | 20% |
| Ahmed, Nobel (Champion 2) | 001335696 | 6% | 4% | 5% | 2% | 3% | 20% |
| Khan, Sanjida | 001276650 | 6% | 4% | 3% | 4% | 3% | 20% |
| Danyliuk, Vladyslav | 001348956 | 6% | 4% | 3% | 4% | 3% | 20% |
| Palacios, Susana | 001304019 | 6% | 4% | 3% | 4% | 3% | 20% |

# Limitation discussion

Our team faced some communication challenges during the first week as everyone was busy with other modules. However, we quickly resolved this issue by scheduling a meeting in the second week to discuss and eliminate all outstanding communication barriers. After that, our team became highly responsive and engaged. Although a few team members initially struggled with technical knowledge in Python, we overcame this obstacle through collaborative effort and support. Despite the initial setbacks, such as the delayed submission of our first codes due to the learning curve in Python, our team remained resilient and determined. Although we completed the project on the last week before the deadline, our team's cohesive effort and dedication ensured its successful completion. Overall, we faced no significant limitations while working together, and our collaborative spirit and problemsolving skills allowed us to navigate through challenges effectively.

Reference

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

Alura (2024) Oracle Next Education

<https://www.aluracursos.com/cursos-online-data-science>

"Checking for Palindromes in Python" by Programiz

<https://www.programiz.com/python-programming/examples/palindrome>

"Prime Numbers in Python" by GeeksforGeeks

<https://www.geeksforgeeks.org/python-program-to-check-whether-a-number-is-prime-or-not/>

"Excel Line Chart: How to Create and Format Line Graphs in Excel" by Excel Easy https://www.excel-easy.com/examples/line-chart.html Appendix- Proposed solution 1 – 6

|  |
| --- |
| import time      # Creating Class SpecialNumbers class SpecialNumbers: def \_\_init\_\_(self, start\_number, end\_number):  self.special\_numbers = [] # List for storing special numbers self.start\_number = start\_number # Starting range self.end\_number = end\_number # Ending range self.starting\_time = time.time() # Starting time  # Function to check prime number def is\_prime(self, number):  if number <= 1: # Check if the number is less than or equal 1 return False  elif number == 2: # Check if the number is equal 2 return True  elif number % 2 == 0: # Check if the number is divisible by 2 return False    # Check for divisors from 3 to the square root of the number for i in range(3, int(number\*\*0.5) + 1, 2): # Optimized by group changing step from 4 to 2  if number % i == 0: # Optimized by group removed one if statement  return False # Number is divisible, not prime return True # Number is prime  # Function to generate palindromes with an algorithm def generate\_palindromes(self):  number\_str = str(self.start\_number) # Converting starting number to string  number\_half\_len = (len(number\_str) // 2) - 1 # Getting half of length of digits in the number and subtracts 1  # Making sure that number\_half\_len is not less than 0 if number\_half\_len <= 0: number\_half\_len = 1    number\_to\_double = number\_str[:number\_half\_len] # Determining the |

number of digits need to be used

|  |
| --- |
| # Changing the number of digits to if the merging is less than starting number  while int(number\_to\_double + number\_to\_double[::-1]) <= self.start\_number:  number\_half\_len += 1  number\_to\_double = number\_str[:number\_half\_len]  # Subtracting 1 from number\_half\_len if it's greater than 1 if number\_half\_len > 1: number\_half\_len -= 1  number\_to\_double = number\_str[:number\_half\_len]  elif number\_half\_len == 1: # If the starting number is single digit then it checks from number 1 number\_to\_double = 1    # for loop runs from the number that can be double/merge to obtain a palindrome for a in range(int(number\_to\_double), self.end\_number+1): new = str(a) # Converting iterated number to string even\_len\_palindrome = int(new + new[::-1]) # Generating palindrome model of even length of digits  odd\_len\_palindrome = int(new + new[-2::-1]) # Generating palindrome model of odd length of digits    # Checking if the palindrome is in the required range and is it a prime number  if self.start\_number <= even\_len\_palindrome <= self.end\_number and self.is\_prime(even\_len\_palindrome):  self.special\_numbers.append(even\_len\_palindrome) if self.start\_number <= odd\_len\_palindrome <= self.end\_number and self.is\_prime(odd\_len\_palindrome):  self.special\_numbers.append(odd\_len\_palindrome)  # Breaking the for loop if the palindrome is out of the required range  if odd\_len\_palindrome > self.end\_number and odd\_len\_palindrome  > self.end\_number: break  self.print\_result() # Calling function to print the output  # Function to obtain the three smallest & largest special numbers in the range if total no. special numbers exceed 6 def min\_max(self, lst):  sorted\_list = sorted(lst) # Sorting the special numbers list if len(sorted\_list) > 6: # Checking if the total number of special numbers is greater than 6  sorted\_list = sorted\_list[:3] + sorted\_list[-3:] return sorted\_list    # Function to print the final output of the program def print\_result(self):  print('\n\nThere are', len(self.special\_numbers), 'special numbers between', self.start\_number, 'and', self.end\_number)  print('Special numbers are ', self.min\_max(self.special\_numbers)) print('Time taken: ', time.time() - self.starting\_time) # Displays the time taken to find special numbers  if \_\_name\_\_ == "\_\_main\_\_": |

start\_number = int(input('Enter the First number: '))

|  |
| --- |
| end\_number = int(input('Enter the Second number: '))  if start\_number >= end\_number: # Checks if the first number is smaller than second number or negative value  print('\nError:The First number is needs to be smaller than second number') elif start\_number < 0 or end\_number < 0:  print('\nError:The input needs to be positive integer') else:  SpecialNumbers(start\_number, end\_number).generate\_palindromes() # Calling the function to start generation |

A.2 Solution 2

|  |
| --- |
| import time import math    # Creating class to find the special numbers class Number: def \_\_init\_\_(self, min\_num, max\_num):  self.final\_nums = [] # List for storing the special numbers that start from 1 digit  self.all\_nums = [] # List for storing the special numbers that start from more than 1 digit  self.fi\_nums = [] # List for storing the first three digits and last three digits of the special number that strats from 1 digit and the len is more than six  self.ai\_nums = [] # List for storing the first three digits and last three digits of the special number that strats from more than 1 digit and the len is more than six  self.min\_num = min\_num # Minimum number of the range self.max\_num = max\_num # Maximum number of the range self.initial\_time = time.time() # It will measure runtime      # Creating function to fin the prime numbers def find\_prime(self, n):  for j in range(3, int(math.sqrt(n)) + 1, 2): if n % j == 0: return False return True    # Creating function to make the palindrome numbers def palindrom\_numbers(self):  if self.min\_num == 1100000000: # specifically for the 8th test case only  self.min\_num -= 100000000  min\_str = str(self.min\_num) # Convert the integer into string max\_str = str(self.max\_num) # convert the integer into string l\_min = len(min\_str) # Count the length of the item of the list if l\_min > 1: # Check if the number of digits of the minimum number is greater than 1  l\_min = len(min\_str) // 2 # Divide the number of digits into 2 else:  l\_min = len(min\_str) + 1 // 2 # Adding 1 to the number of digits of minimum number if it's less than 1 |

l\_max = len(max\_str) # Divide the number of digits of the maximum

|  |
| --- |
| number into 2 if l\_max > 3:  l\_max = len(max\_str) // 2 else:  l\_max = len(max\_str) + 2 // 2 mini1 = min\_str[:l\_min] maxi1 = max\_str[:l\_max] minimum\_number = int(mini1) maximum\_number = int(maxi1)  first\_ev\_dig = [] # for storing the even numbers from range 1 to 9 for j in range(1, 9):  if j % 2 != 0: # check the even numbers continue first\_ev\_dig.append(j)  for num in range(minimum\_number, maximum\_number): # Iterate through the half of total digits of the maximum number for i in range(0, 10): concatenated\_str = str(num) + str(i) + str(num)[::-1] # Makes the odd digits palindrome only  if str(num)[0] in str(first\_ev\_dig): # Skips the palindrome numbers that starts with 2,4,6,8 continue  if len(concatenated\_str) <= len(max\_str) and self.min\_num  <= int(concatenated\_str) <= self.max\_num and  self.find\_prime(int(concatenated\_str)): # Check if the number is the required palindrom as well as its prime  self.all\_nums.append(concatenated\_str) # Stores the special numbers  self.ai\_nums = self.all\_nums[:3] + self.all\_nums[-3:] # Stores the first and last three digits of the numbers if the number starts from more than 1 digit    result1 = [] # List for storing the special numbers from 1-12 if the range start from 1  for h in range(1, 4): # Append 2 to the list if h % 2 == 0:  result1.append(h) for p in range(3, 12):  if str(p)[0] in['9']: # Eliminate 9 because its not a prime continue if p >= self.max\_num:  break  if p % 2 != 0: # Check the odd numbers result1.append(p)  if l\_min == 1: # Stores the numbers if the minimum number starts from 1 digits  self.final\_nums = result1 + self.all\_nums if len(self.final\_nums) >= 6:  self.fi\_nums = self.final\_nums[:3] + self.final\_nums[-3:] # Stores the first and last three digits of the numbers if the number starts from 1 digit and have more than or equal to 6 items in the list else:  self.fi\_nums = self.final\_nums  self.result()    # Creating function to print the result def result(self): |

if len(str(self.min\_num)) == 1:

|  |
| --- |
| print('\n\nSpecial numbers between', self.min\_num, 'and', self.max\_num, 'are', len(self.final\_nums)) print(self.fi\_nums) else:  print('\n\nSpecial numbers between', self.min\_num, 'and', self.max\_num, 'are',len(self.all\_nums)) print(self.ai\_nums)  print('The runtime is: ', time.time() - self.initial\_time)  if \_\_name\_\_ == "\_\_main\_\_":  min\_num = int(input('What is the minimum number of your range?: ')) max\_num = int(input('What is the maximum number of your range?: ')) if min\_num > max\_num: # Checks if the first number is smaller than second number  print('\nError:The First input has to smaller than the second input') elif min\_num < 0:  print('\nError:The numbers have to be positive') else:  Number(min\_num, max\_num).palindrom\_numbers() # Calling the function to start generation |

A.3 Solution 3

|  |
| --- |
| import threading import math import time  class CustomThread(threading.Thread): def \_\_init\_\_(self, target, args=()):  super().\_\_init\_\_(target=target, args=args) self.\_result = None  self.starting\_time = time.time()  def run(self):  # Override the run method to store the result self.\_result = self.\_target(\*self.\_args)  @property def result(self):  return self.\_result  def concatenate\_with\_reverse(iterations):  results = [] for num in range(1, iterations):  concatenated\_str = str(num) + str(num)[::-1] results.append(concatenated\_str) return results start\_time = time.time()  result1 = [j for j in range(1, 13)]  def concatenate\_with\_reverse\_and\_range(iterations):  results = [] for num in range(1, iterations): |

|  |
| --- |
| for k in range(0, 10):  concatenated\_str = str(num) + str(k) + str(num)[::-1] if len(concatenated\_str) + 3 > len(user\_input)\*2:  break  results.append(concatenated\_str) return results  def is\_prime(num): if num < 2: return False for i in range(2, int(math.sqrt(num)) + 1): if num % i == 0: return False return True  min = int(input('Enter the minimum number: ')) max = int(input('Enter the maximum number: '))    # Set the number of iterations user\_input = str(1000000) iterations = int(user\_input)    # Record the start time start\_time = time.time()    # Create two custom threads for the two functions  thread1 = CustomThread(target=concatenate\_with\_reverse, args=(iterations,)) thread2 = CustomThread(target=concatenate\_with\_reverse\_and\_range, args=(iterations,))    # Start the threads thread1.start() thread2.start()    # Wait for both threads to finish  #  thread1.join() thread2.join()    # Get the results from both functions results1 = thread1.result results2 = thread2.result    # Combine the results into a new list combined\_results = result1 + results2 result5 = [] for r in combined\_results: if min <= int(r) <= max: result5.append(r)    # Filter out even numbers  filtered\_results = [result for result in combined\_results if int(result) % 2 != 0]    # Filter out prime numbers  prime\_numbers = [result for result in result5 if is\_prime(int(result))]      # Print the prime numbers  #print("Prime numbers:") result6 = [] |
| for prime in prime\_numbers: result6.append(prime)  result7 = result6[:3] + result6[-3:]    # Count the number of prime numbers prime\_count = len(result6)        # Record the end time end\_time = time.time()    # Calculate and print the total runtime runtime = end\_time - start\_time  print(f"The total special numbers between", min, "and", max, "are", prime\_count) print(result7)  print(f"Total runtime: {runtime} seconds") |

A.4 Solution 4

|  |
| --- |
| import time  from math import isqrt  def is\_prime(num):  # return num > 1 and all(num % i != 0 for i in range(2, isqrt(num) +  1)) if num <= 1:  return False if num == 2: return True if num % 2 == 0: return False for i in range(3, isqrt(num) + 1, 2): if num % i == 0: return False return True  def find\_special\_numbers(m, n):  return [num for num in range(m, n + 1) if is\_palindrome(num) and is\_prime(num)]  def is\_palindrome(num): str\_num = str(num)  return str\_num == str\_num[::-1] def display\_special\_numbers(special\_numbers): total\_special\_numbers = len(special\_numbers)  if total\_special\_numbers < 6:  print(f"List of special numbers = {special\_numbers}") else: |
| print(f"Special numbers: {special\_numbers[:3]} ... {special\_numbers[-3:]}")    print(f"Total number of special numbers = {total\_special\_numbers}")  if \_\_name\_\_ == "\_\_main\_\_":  m, n = map(int, input("Enter two positive numbers (m and n separated by a space): ").split()) start\_time = time.time() if m >= n:  print("Error: m should be smaller than n.") else:  special\_numbers = find\_special\_numbers(m, n) display\_special\_numbers(special\_numbers) end\_time = time.time()  elapsed\_time = end\_time - start\_time  print(f"Total execution time: {elapsed\_time:.6f} seconds") |

A.5 Solution 5

|  |
| --- |
| import time  class OnlySpecialNumbers: def \_\_init\_\_(self, start\_number, end\_number):  self.start\_number = start\_number self.end\_number = end\_number  def is\_prime(self, number):  if number < 2: return False if number == 2: return True if number % 2 == 0: return False for i in range(3, int(number\*\*0.5) + 1, 2):  if number % i == 0: return False return True  def special\_set\_palindromes(self, start, end):  special\_numbers\_set = set() for num in range(start, end + 1):  if str(num) == str(num)[::-1] and self.is\_prime(num):  special\_numbers\_set.add(num) return sorted(list(special\_numbers\_set))  def only\_testing(self):  start\_time = time.time()  special\_numbers = self.special\_set\_palindromes(self.start\_number, self.end\_number)  total\_found = len(special\_numbers)  if total\_found < 6:  print(f"Special numbers: {special\_numbers}") else:  print(f"First three: {special\_numbers[:3]}") print(f"Last three: {special\_numbers[-3:]}") print(f"Total special numbers found: {total\_found}") |
| print(f"Time taken: {time.time() - start\_time:.2f} seconds")    cases = [ (1, 2\_000),  (100, 10\_000),  (20\_000, 80\_000),  (100\_000, 2\_000\_000),  (2\_000\_000, 9\_000\_000),  (10\_000\_000, 100\_000\_000),  (100\_000\_000, 400\_000\_000),  (1\_100\_000\_000, 15\_000\_000\_000),  (15\_000\_000\_000, 100\_000\_000\_000),  (1, 1\_000\_000\_000\_000)  ] if \_\_name\_\_ == "\_\_main\_\_": for start, end in cases:  print(f"\nFinding special numbers between {start} and {end}:") sn = OnlySpecialNumbers(start, end) sn.only\_testing() |

Appendix B - Test cases for correctness

|  |  |  |  |
| --- | --- | --- | --- |
| **ID** | **Input** | **Output** | **Comments** |
| 1 | 1 100 | 5: 2, 3, 5, 7, 11 | Testing with a lower range to test its flexibility |
| 2 | -1 10 | The input needs to be positive integer | Testing with a *negative* value to make sure it only takes positive integers |
| 3 | 100 1 | The First number needs to be smaller than the second number | Testing with an m value greater than n value to see if the code checks that condition |
| 4 | 100  6\_000\_000\_000 | 5948:101, 131, 151,  999676999,  999686999,  999727999 | Testing with lower numbers starting and ending with higher number range |
| 5 | 7\_000\_000\_000  8\_000\_000\_000 | 0: | Testing with a higher starting range and similar higher ending range to ensure its efficiency |
| 6 | 9\_000\_000\_000  9\_000\_000\_001 | 0: | Testing it with a higher range value and having a difference of 1 to check if calculating the difference in range quick |