

## Algorithms and Data Structures (ADS) - COMP1819

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

Shared document link: [02\\_04\\_001419181\\_001411145\\_001424560](#)

Group Name: **02\_04**

Team members: with different text colours:

Member	Name	ID	Contribution %
1	Shymanskyi Brendon	001419181	100%
2	Patwardhan Liladhar	001411145	100%
3	Alenezi Abdullah	001424560	100%
4	Hoque Sami	001421408	0%

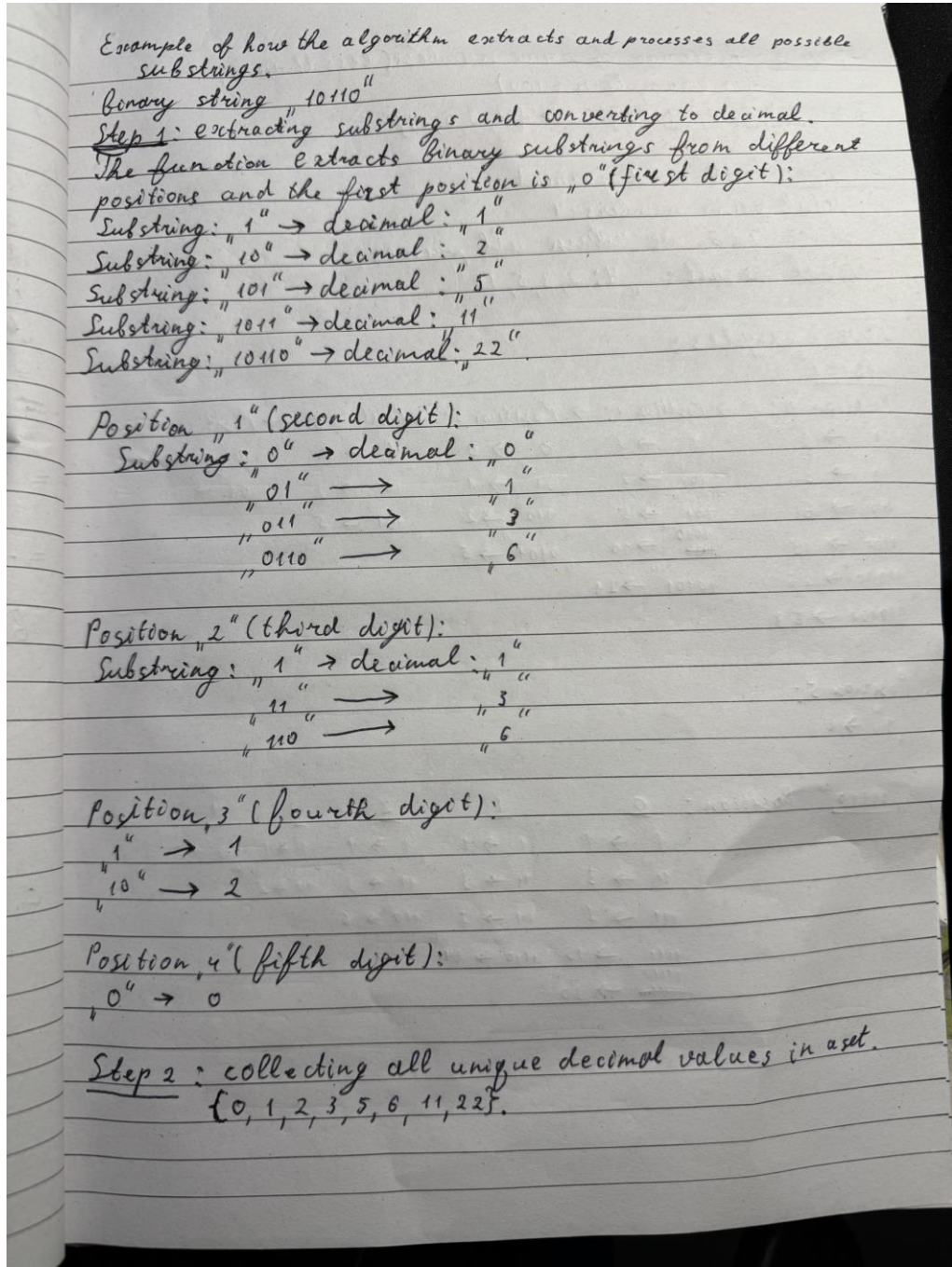
### Contents

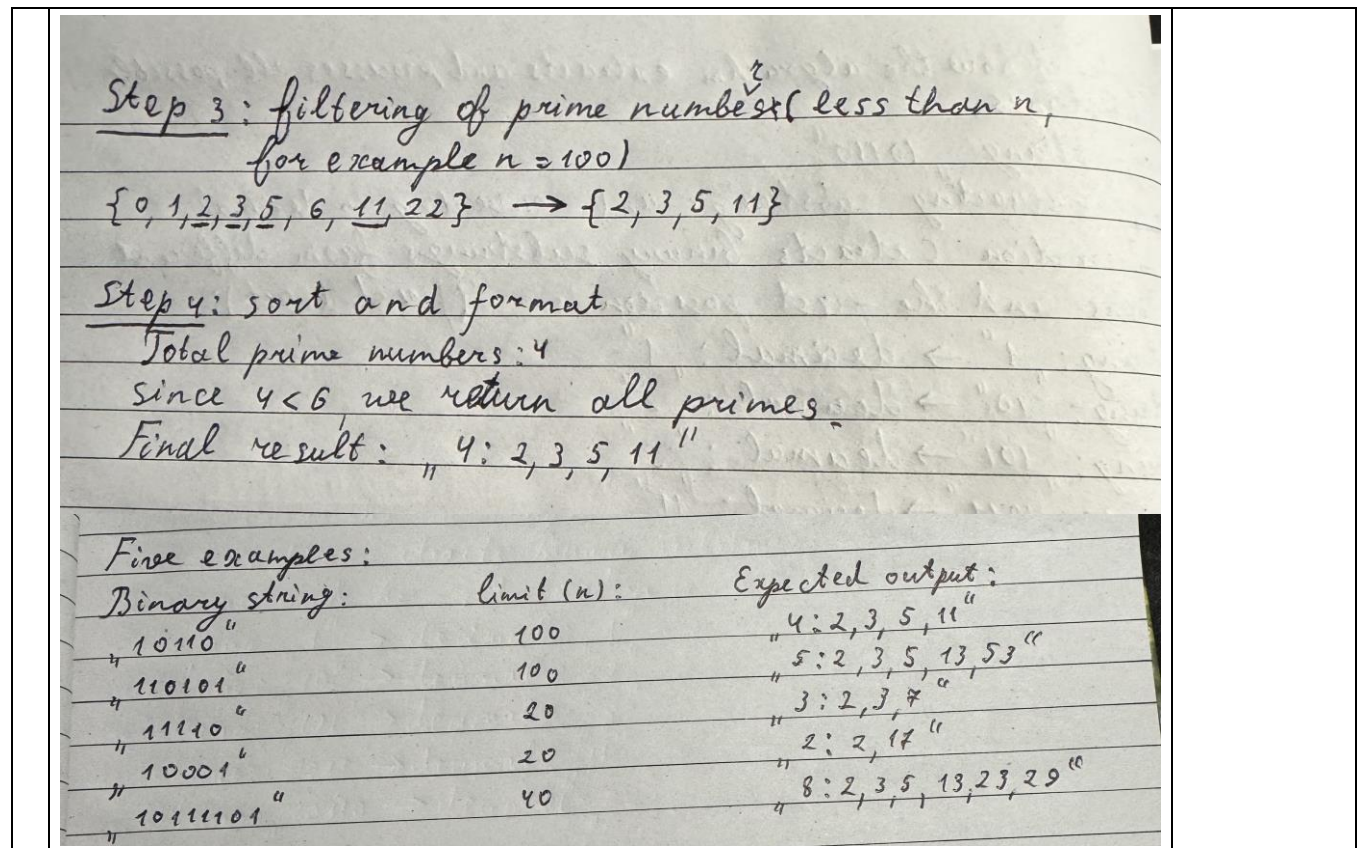
1. Creating Two Basic Working Solutions .....	3
1.1 Understanding the Problem .....	3
1.2 Solution Implementations .....	4
1.3 Code Submission .....	5
2. Testing and Comparing Solutions .....	5
2.1 Test Cases & Validation.....	5
2.2 Running on Given Test Cases .....	5
2.3 Comparison of Solutions .....	6
Running time graph.....	8
3. Optimising Solutions .....	8
3.1 Selecting a Solution for Optimisation .....	8
3.2 Optimisation Steps & Justification .....	8
3.3 Code Submission .....	9
4. Comparing Performance .....	9
4.1 Performance Comparison .....	9
4.2 Running on Given Test Cases .....	11
4.3 Visualising Performance.....	11
5. Reflecting on Teamwork .....	13

5.1 Contribution Marks & Team Agreement .....	13
5.3 Summary of Roles & Contributions .....	14
5.2 Weekly Journal & Communication Logs .....	14
Weekly journal .....	14
5.4 Final Report Quality .....	15
Reference .....	15
Appendix .....	15
Appendix A.1 - Proposed solution 1 (code written by Shymanskyi Brendon) .....	15
Appendix A.2 - Proposed solution 2 (code written by Patwardhan Liladhar .....	17
Appendix A.3 – Optimised solution .....	19
Appendix B - Further evidence of team contribution if necessary.....	20

# 1. Creating Two Basic Working Solutions

## 1.1 Understanding the Problem

#	Handwritten with explanation (attached)	By (student name)
	 <p>Example of how the algorithm extracts and processes all possible substrings.</p> <p>Binary string "10110"</p> <p>Step 1: extracting substrings and converting to decimal.</p> <p>The function extracts binary substrings from different positions and the first position is "0" (first digit):</p> <p>Substring: "1" → decimal: "1"</p> <p>Substring: "10" → decimal: "2"</p> <p>Substring: "101" → decimal: "5"</p> <p>Substring: "1011" → decimal: "11"</p> <p>Substring: "10110" → decimal: "22"</p> <p>Position "1" (second digit):</p> <p>Substring: "0" → decimal: "0"</p> <p>"01" → "1"</p> <p>"011" → "3"</p> <p>"0110" → "6"</p> <p>Position "2" (third digit):</p> <p>Substring: "1" → decimal: "1"</p> <p>"11" → "3"</p> <p>"110" → "6"</p> <p>Position "3" (fourth digit):</p> <p>"1" → 1</p> <p>"10" → 2</p> <p>Position "4" (fifth digit):</p> <p>"0" → 0</p> <p>Step 2: collecting all unique decimal values in a set.</p> <p>{0, 1, 2, 3, 5, 6, 11, 22}.</p>	By Brendon Shymanskyi



Every team member contributed to the task and understood the essence of the task.

## 1.2 Solution Implementations

### Solution 1:

Short description and highlights of the **use of the data structure in your code**

Using 'set()':

Set help to automatically remove duplicates, making the process faster and less draining on system resources. Using a set() simplifies your work because you don't have to search for new values in the list to see if they have already been added.

When it's time to filter primes, another Set is added to store unique primes so that other operations can be performed more quickly.

By using sets instead of lists, the application avoids unnecessary storage, so it is faster and less expensive for the system.

### Solution 2:

Short description and highlights of the **use of the data structure in your code**

Using List:

This solution uses **list** data structure to store unique prime numbers found in the binary string. It iterates through all substrings of the binary string, converts them to decimal, and checks if they are prime and less than the given limit. The list ensures uniqueness by checking if the number is already in the list before appending it.

To sum up, the list of primes is sorted and formatted according to the requirements.

### Key Features:

**Nested Loops:** The outer loop iterates through the starting index of the substring, and the inner loop iterates through the ending index.

Prime Checking: The `prime()` function checks if a number is prime by testing divisibility up to the square root of the number.

Uniqueness: The list ensures that only unique prime numbers are stored by checking if the number is already in the list.

Sorting: The list of primes is sorted in ascending order before formatting the output.

### 1.3 Code Submission

The full source code is included in the **Appendix**.

## 2. Testing and Comparing Solutions

## 2.1 Test Cases & Validation

### Testing solutions using the handwritten examples from Task 1:

Test case	Input (Binary string and limit(n))	Expected output	Snapshots
1	"10110", 100	"4: 2, 3, 5, 11"	4: 2, 3, 5, 11
2	"110101", 100	"5: 2, 3, 5, 13, 53"	5: 2, 3, 5, 13, 53
3	"11110", 20	"3: 2, 3, 7"	3: 2, 3, 7
4	"10001", 20	"2: 2, 17"	2: 2, 17
5	"10111101", 40	"8: 2, 3, 5, 13, 23, 29"	8: 2, 3, 5, 13, 23, 29

## 2.2 Running on Given Test Cases

Test case No	Input	Output	Pass/Fail for Solution <u>1</u>	Pass/Fail for Solution <u>2</u>	Running time (s) for solution <u>1</u>	Running time (s) for solution <u>2</u>
1	"0100001101001111", 999999	15: 2, 3, 5, 269, 2153, 17231	Pass	Pass	0.000000	0.000000
2	"0100001101001111010 0110101010000", 999999	28: 2, 3, 5, 10729, 17231, 85837	Pass	Pass	0.000998	0.000998
3	"11111111111111111111 11111111111111111111".	7: 3, 7, 31, 8191, 131071, 524287	Pass	Pass	0.000000	0.000997

	999999					
4	"0100001101001111010 01101010100000011000 100111000", 999999999	44: 2, 3, 5, 5571347, 41577089, 55398449	Pass	Pass	0.000999	0.001994
5	"0100001101001111010 01101010100000011000 10011100000110001", 123456789012	52: 2, 3, 5, 84087683, 3920234023, 5625434161	Pass	Pass	0.012962	0.016959
6	"0100001101001111010 01101010100000011000 10011100000110001001 11001", 123456789012 345	64: 2, 3, 5, 2724796139969, 5841981288761, 45810224399911	Pass	Pass	0.729383	0.975870
7	"0100001101001111010 01101010100000011000 10011100000110001001 1100100100001", 123456789012345678	71: 2, 3, 5, 45810224399911, 7435453988964809, 18946016916092977	Pass	Pass	14.988570	28.733273
8	"010000110100111101 0011010101000000110 0010011100000110001 0011100100100001010 00001", 12345678901 23456789	76: 2, 3, 5, 45810224399911, 7435453988964809, 18946016916092977	Pass	Pass	20.946981	35.625465
9	"0100001101001111010 01101010100000011000 10011100000110001001 11001001000010100000 101000100", 12345678 90123456789	81: 2, 3, 5, 7435453988964809, 18946016916092977, 378518838354150661	Pass	Pass	53.943795	112.505318
10	"0100001101001111010 01101010100000011000 10011100000110001001 11001001000010100000 10100010001010011",1 2345678901234567890	89: 2, 3, 5, 7435453988964809, 18946016916092977, 378518838354150661	Pass	Pass	71.572761	211.031828

## 2.3 Comparison of Solutions

### Differences in data structures and their impact on performance:

#### ▪ Data Structure: set()

The set() data structure is used to store unique decimal value extracted from binary numbers

#### ▪ Impact on performance:

- Advantages: Set automatically remove duplicates which makes the process easier to get uniqueness. This reduces the need to check additionally and improves performance with larger inputs.
- Disadvantages: Set require a lot of memory to run than list but in our solution performance gain after avoiding duplicate checks outweighs the memory cost.

#### ▪ Data Structure: list()

The list() data structure is used to store unique prime numbers found in the binary string.

#### ▪ Impact on performance:

- Advantages: List is simple and straightforward to use they are memory efficient for smaller datasets.
- Disadvantages: Lists requires manual checks for uniqueness. This increases time complexity for larger inputs, as the list grows more checks are required

### Theoretical analysis of time complexity for both approaches:

#### ▪ Solution 1 set() Data Structure:

Time Complexity: Getting all substrings  $O(n^2)$  length of the binary string is  $n$

Checking primes:  $O(m\sqrt{m})$  max decimal value extracted from the binary string is  $m$ .

Overall Complexity:  $O(n^2+m\sqrt{m})$

The set() data structure make sure that duplicate values automatically gets removed decreasing the number of prime checks required this makes the solution more efficient for large inputs.

#### ▪ Solution 2 list() Data Structure:

Time Complexity: Getting all substrings  $O(n^2)$   $n$  is the length of binary string

Checking for prime numbers and uniqueness:  $O(m\sqrt{m}+m)$   $m$  is max decimal value extracted from binary string.

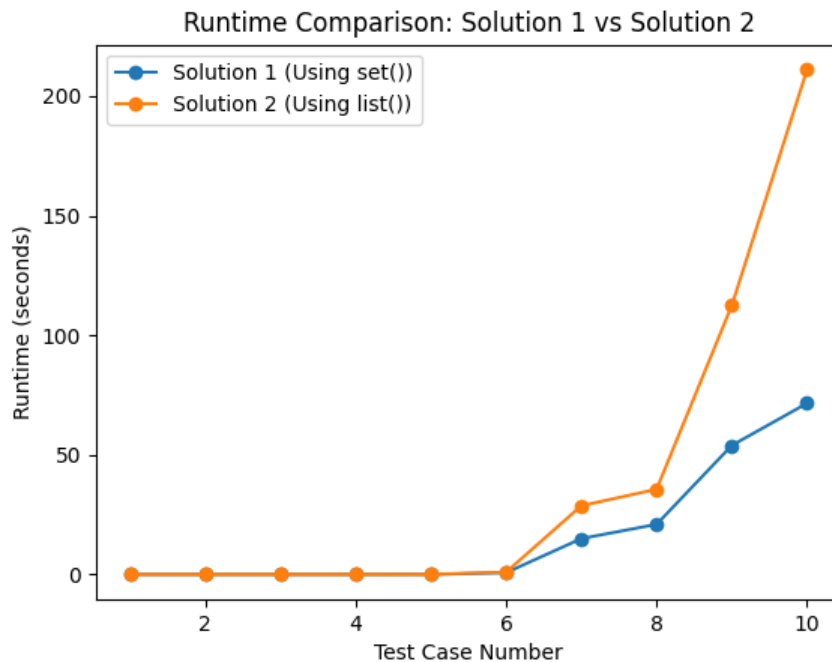
Overall complexity:  $O(n^2+m\sqrt{m}+m)$

The list() requires manual checks for uniqueness which adds an unnecessary  $O(m)$  complexity which makes the solution slow for larger inputs.

### The graph showing how runtime changes across 10 test inputs:



## Running time graph



### 3. Optimising Solutions

#### 3.1 Selecting a Solution for Optimisation

We decided to use the basic solution with the set data structure for optimisation because it had better performance than the list solution. The set-based solution was more scalable because it automatically handled duplicates, thereby reducing the need for additional checks. This made it more suitable for optimisation, as it was already much more suited to efficiently processing large data sets.

After reviewing the possible results and testing the entire code, we determined that the is\_prime() function takes most of the execution time and decided that it would be good to improve this particular function, which will result in better results and faster execution of test cases.

Our goal was to implement optimisation detection to improve the basic solution by at least half. We would then test the optimised solution against the original test cases to measure the performance improvements.

#### 3.2 Optimisation Steps & Justification

First optimised version.

The code has been optimised so that the function that calculates primes does not go through absolutely all numbers. After reviewing it, we realised that it would be much better to optimise the function by reducing the number of steps the code will go through, so it skips every second element, that is, an even number, and the function automatically starts the path from the number 5. With this function our solution completed last test case in 35 seconds.



```
# O(sqrt(n))
def is_prime(n):
    if n < 2:
        return False
    if n in (2, 3):
        return True
    # every number divisible by 2 or 3 is not prime
    if n % 2 == 0 or n % 3 == 0:
        return False
    # finding a divisor from 5 to sqrt(n), and skipping every 2nd number.
    for i in range(5, int(n ** 0.5) + 1, 2):
        if n % i == 0: # if n is divisible by any number, then it is not prime
            return False
    return True
```

Second and final optimised version.

Finding prime numbers using the  $6k+1$  method. We learnt about this method during a research on the “GeeksforGeeks” website. This method uses  $6k+1$  method to check divisibility and adding 6 every time to check the pair of divisors. With this function code completed last test case in 27 seconds.

This method, like the previous ones, corresponds to  $O(\sqrt{n})$  in terms of Big O Performance.

```
def is_prime(n):
    """
    prime-checking function using  $6k \pm 1$  method
    """
    if n < 2:
        return False
    if n in (2, 3):
        return True
    if n % 2 == 0 or n % 3 == 0: # eliminating multiples of 2 and 3
        return False
    if n < 9:
        return True
    k, limit = 5, int(n ** 0.5) + 1 # starting to check from 5 up to the square root of n
    while k <= limit:
        if n % k == 0 or n % (k + 2) == 0: #  $6k + 1$  method to check divisibility
            return False
        k += 6 # adding 6 to check the pair of divisors
    return True # if no divisors
```

### 3.3 Code Submission

The final **optimised code** is included in the **Appendix**.

## 4. Comparing Performance

### 4.1 Performance Comparison

Comparison of solutions by execution time on the example of 10 given test cases:

Test Case	Running Time (s), Basic Solution	Running Time (s), Optimised Solution	Time difference (Basic Solution– Optimised)(s)
1	0.000000	0.000000	0

2	0.000998	0.000000	0.000998
3	0.000000	0.000000	0
4	0.000999	0.000997	0.000002
5	0.012962	0.004987	0.007975
6	0.729383	0.293243	0.43614
7	14.988570	5.891723	9.096847
8	20.946981	8.271387	12.675594
9	53.943795	21.515236	32.428559
10	71.572761	27.671312	43.901449

## Difference in approaches to implementation:

Basic solution:

```
def is_prime(n): # Big O - O(sqrt(n))
    if n < 2:
        return False
    for i in range(2, int(n ** 0.5) + 1):
        if n % i == 0:
            return False
    return True
```

All the differences were in the change of the 'is\_prime' function. The code of the first solution is passed through each number after 2, which is completely non-optimised.

Optimised solution:

```
def is_prime(n):
    """
    prime-checking function using 6k ± 1 method
    """
    if n < 2:
        return False
    if n in (2, 3):
        return True
    if n % 2 == 0 or n % 3 == 0: # eliminating multiples of 2 and 3
        return False
    if n < 9:
        return True
    k, limit = 5, int(n ** 0.5) + 1 # starting to check from 5 up to the square root of n
```

```

while k <= limit:
    if n % k == 0 or n % (k + 2) == 0: # 6k + 1 method to check divisibility
        return False
    k += 6 # adding 6 to check the pair of divisors
return True # if no divisors

```

The code of the optimised solution uses the  $6k \pm 1$  method, which optimises prime number checking by skipping all even numbers and multiples of 3. Also, it first goes through the specified steps that are located before the check from 5 to  $\sqrt{n}$ , which also allows to skip some verification steps.

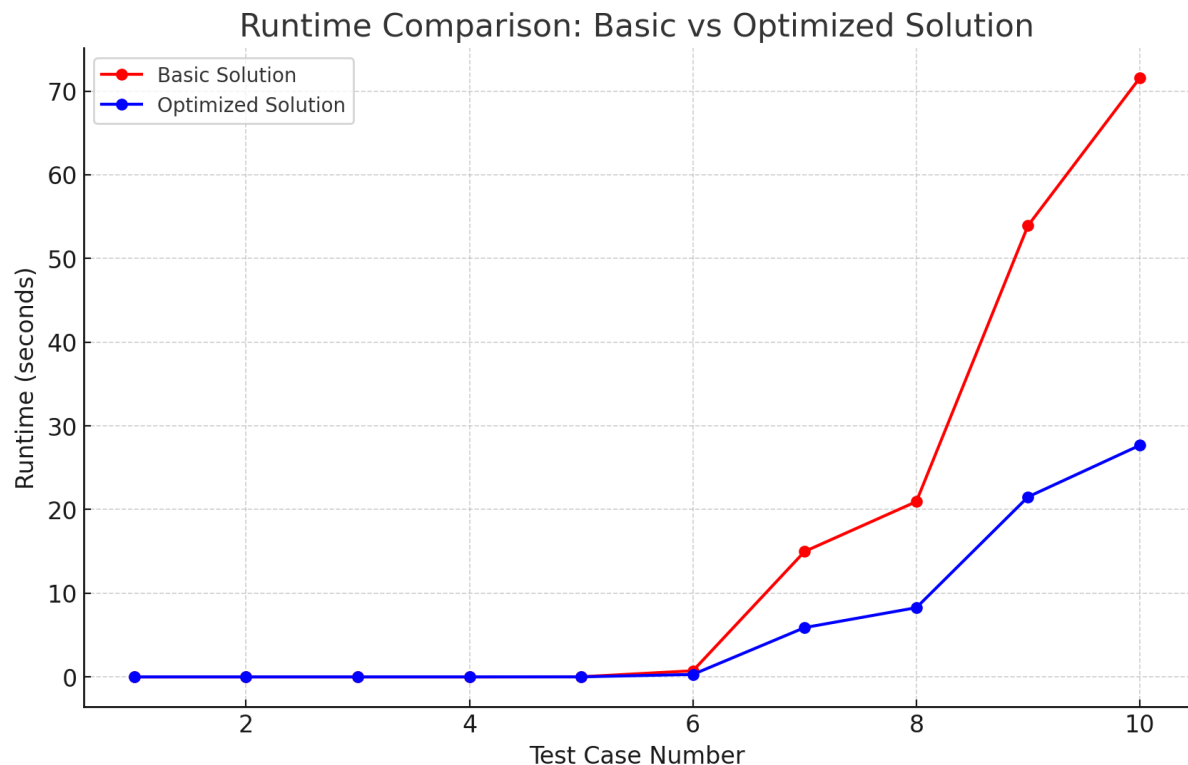
## 4.2 Running on Given Test Cases

Running the optimised solution on the 10 given test cases:

Test Case	Output	Running Time (s)	Pass/Fail
1	15: 2, 3, 5, 269, 2153, 17231	0.000000	Pass
2	28: 2, 3, 5, 10729, 17231, 85837	0.000000	Pass
3	7: 3, 7, 31, 8191, 131071, 524287	0.000000	Pass
4	44: 2, 3, 5, 5571347, 41577089, 55398449	0.000997	Pass
5	52: 2, 3, 5, 84087683, 3920234023, 5625434161	0.004987	Pass
6	64: 2, 3, 5, 2724796139969, 5841981288761, 45810224399911	0.293243	Pass
7	71: 2, 3, 5, 45810224399911, 7435453988964809, 18946016916092977	5.891723	Pass
8	76: 2, 3, 5, 45810224399911, 7435453988964809, 18946016916092977	8.271387	Pass
9	81: 2, 3, 5, 7435453988964809, 18946016916092977, 378518838354150661	21.515236	Pass
10	89: 2, 3, 5, 7435453988964809, 18946016916092977, 378518838354150661	27.671312	Pass

## 4.3 Visualising Performance

Runtime comparison graph based on ten given test cases:



### Time complexity and Big-O notation for the optimised solution.

The optimized version has Big O performance of  $O(n^2 * \sqrt{m})$ .

### Big O notation in "is\_prime" function.

Base cases( $O(1)$  operations):

1. If  $n < 2$ :  
return False;
2. If  $n$  in (2, 3):  
return True;
3. If  $n \% 2 == 0$  or  $n \% 3 == 0$ :  
return False;
4. If  $n < 9$ :  
return True;

$O(\sqrt{n})$  operations:

1. The while loop runs from 5 to  $\sqrt{n}$  in steps of 6.
  - The number of iterations is  $(\sqrt{n})/6$ ;
  - The loop runs until  $k$  exceeds the square root of  $n$  (limit =  $\text{int}(n^{**0.5}) + 1$ ).

### Time complexity analysis: $O(\sqrt{n})$

Best case:

- n must be an even number or divisible by 3 for best case scenario, the function will return False immediately.

Worst case:

- The worst case occurs when n is a prime number, and the loop executes until k exceeds the square root of n.

- When the number of iterations is proportional to the square root of  $n$  divided by 6, since  $k$  increases by 6 in each iteration.

Average case:

- The loop can terminate early if a divisor is found.
- However, the worst case dominates the time complexity, so the average case is still  $O(\sqrt{n})$ .

Big O notation:

The time complexity is  $O(\sqrt{n})$  in the worst and average cases, and  $O(1)$  if it is best case.

### **Big O notation in “extract\_binary\_substrings(binary\_string)” function.**

Time complexity analysis:  $O(n^2)$

Best case ( $O(n)$ ):

- The best case occurs when the input string is invalid (when contains other characters than just “0” and “1”), so this is.

Worst case ( $O(n^2)$ ):

- The worst case occurs when the input string is valid and the nested loops run fully.

Average case ( $O(n^2)$ ):

- Same as the worst case.

Big O notation:

The time complexity is  $O(n^2)$  in the worst and average cases, and  $O(n)$  if it is best case.

### **Big O notation in “main\_primes(binary\_str, n)” function.**

Time complexity analysis:  $O(n^2 * \sqrt{m})$

1. Calls “extract\_binary\_substrings” function first, which is  $O(n^2)$ .
2. For extracted numbers the “is\_prime” ( $O(\sqrt{n})$ ) function is called, so the time complexity of this function is  $O(\sqrt{m})$ , where  $m$  is declared as the maximum value of the numbers generated from the binary substrings.

### **Overall time complexity:**

The term in the overall time complexity is  $O(n^2 * \sqrt{m})$ :

- $n$  is a length of binary string;
- $m$  is the maximum value of the numbers from the binary substrings.

## 5. Reflecting on Teamwork

### 5.1 Contribution Marks & Team Agreement

Name	ID	Task 1 (30%)	Task 2 (20%)	Task 3 (20%)	Task 4 (15%)	Task 5 (15%)	Contribution mark (100%)
Shymanskyi, Brendon (Group leader)	001419181	30%	20%	20%	15%	15%	100%
Patwardhan, Liladhar (Developer 1)	001411145	30%	20%	20%	15%	15%	100%
Last, First (Developer 2)		0%	0%	0%	0%	0%	0%
Alenezi, Abdullah (Analyst)	001424560	30%	20%	20%	15%	15%	100%

### 5.3 Summary of Roles & Contributions

**Shymanskyi Brendon (Group Leader)** – Managed deadlines, organised meetings and ensured report clarity. Created first basic solution(1) using the set data structure. Participated in task 1 and wrote an answer to this task. Tested output in task 2 and created tables. Participated in task 3, wrote down the decisions that was made by all team members after discussing the task. Helped with task 4. Filled in the weekly journal.

**Patwardhan Liladhar (Developer 1)** – Implemented second basic solution(2), ensured correctness, and documents runtime measurements. Participated in task 1. Participated in task 2 and completed task 2.3. Helped to make decisions on optimisation in task 3.

**Alenezi Abdullah (Analyst)** – Took charge of testing, created graphs for task 4 and 2, recorded outputs for ran test cases. Participated in the implementation of task 1, 3 and 4.

### 5.2 Weekly Journal & Communication Logs

#### Weekly journal

	Task note	Status
<b>Week 1: from date-date</b>	17/02/2025 - 23/02/2025	
Shymanskyi, Brendon (Group leader)	Reviewed the specification, carefully divided the tasks between the participants and created the first basic solution. Filled weekly journal.	Done
Patwardhan, Liladhar (Developer 1)	Participated in the first task. Created the second basic solution.	Done
Alenezi, Abdullah (Analyst )	Participated in the first task.	Done

<b>Week 2: from date-date</b>	23/02/2025 - 02/03/2025	
<b>Shymanskyi, Brendon (Group leader)</b>	Divided the tasks between team members. Reviewed all the tasks again and helped the participants to understand them. Participated in task 2 and 3.	Done
<b>Patwardhan, Liladhar (Developer 1)</b>	Participated in task 2, helped to make decisions on optimisation in task 3, and completed assignment 2.3.	Done
<b>Alenezi, Abdullah (Analyst )</b>	Participated in tasks. Checked the correctness of compared solutions, graphs.	Done
<b>Week 3: from date-date</b>	02/03/2025 - 09/03/2025	
<b>Shymanskyi, Brendon (Group leader)</b>	Set up a meeting, explained the details and discussed further action on task 4. Directed the team to further work. Participated in task 4.	Done
<b>Patwardhan, Liladhar (Developer 1)</b>		Done
<b>Alenezi, Abdullah (Analyst )</b>	Participated in task 4 and tested the optimised solution for creating a table.	Done
<b>Week 4: from date-date</b>	09/03/2025 - 16/03/2025	
<b>Shymanskyi, Brendon (Group leader)</b>	Checked how the work went, corrected some issues. Prepared a document for submission.	Done
<b>Patwardhan, Liladhar (Developer 1)</b>	Fixed some issues in several tasks in the document.	Done
<b>Alenezi, Abdullah (Analyst )</b>	Reviewed his tasks and those of other team members, checked them.	Done

## 5.4 Final Report Quality

### Reference

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

GeeksforGeeks, Primality Test, 6k method,  
<https://www.geeksforgeeks.org/introduction-to-primality-test-and-school-method/>

Wikipedia. (n.d.). Primality test,  
[https://en.wikipedia.org/wiki/Primality\\_test](https://en.wikipedia.org/wiki/Primality_test)

### Appendix

(Include all solutions as clear text, not screenshots)

## Appendix A.1 - Proposed solution 1 (code written by Shymanskyi Brendon)

Using `set()` as Data Structure.



```

"""
This program takes an input of binary string and extracting substrings,
checks if a number is prime and then sorting the result.
The final result will be fewer than 6 prime numbers.
Using set() as a data structure.
"""

def is_prime(n): # Big O - O(sqrt(n))
    if n < 2:
        return False
    for i in range(2, int(n ** 0.5) + 1):
        if n % i == 0:
            return False
    return True

def extract_binary_substrings(binary_string):
    """ extracts all possible decimal values from given binary substrings """
    if any(c not in '01' for c in binary_string): # checks if the string contains only '0'
or '1'
        return None

    substrings = set() # store decimal values
    length = len(binary_string)
    for i in range(length):
        decimal = 0 # Decimal equivalent
        for j in range(i, length):
            # converting binary -> decimal
            decimal = (decimal << 1) | int(binary_string[j]) # binary-to-decimal
conversion by moving bits to the left
            substrings.add(decimal) # adding the decimal value to the set
    return substrings

def main_primes(binary_str, n):
    """ main function to find and format unique prime numbers in binary substrings """
    binary_substrings = extract_binary_substrings(binary_str)
    if binary_substrings is None:
        return "0: Invalid binary strings"

    # extracting prime numbers that are less than 'n' and are prime.
    prime_numbers = {num for num in binary_substrings if num < n and is_prime(num)}
    if not prime_numbers:
        return "0: No prime numbers found"

    sorted_primes = sorted(set(prime_numbers)) # sorting the prime numbers
    total_primes = len(sorted_primes) # counting the total number of prime numbers
    if total_primes < 6: # if it's less than 6 - return ALL prime numbers
        return f"{total_primes}: {' '.join(map(str, sorted_primes))}"
    else: # if it's greater than 6, return the first 3 and the last 3
        return f"{total_primes}: {' '.join(map(str, sorted_primes[:3] + sorted_primes[-
3:]))}"

def main(binary_string, n):
    import time
    print("Running main function...")
    start = time.time()
    print(main_primes(binary_string, n))
    end = time.time()

```

```

    print("Finished!")
    print(f"Time taken: {end - start:.6f} seconds\n")

# Test Cases:
print("Test 1:") # Time taken: 0.000000 seconds
main("0100001101001111",999999)
print("Test 2:") # Time taken: 0.000998 seconds
main("010000110100111101001101010000",999999)
print("Test 3:") # Time taken: 0.000000 seconds
main("1111111111111111111111111111111111",999999)
print("Test 4:") # Time taken: 0.000999 seconds
main("0100001101001111010011010100000011000100111000",999999999)
print("Test 5:") # Time taken: 0.012962 seconds
main("01000011010011110100110101010000001100010011100000110001",123456789012)
print("Test 6:") # Time taken: 0.729383 seconds
main("0100001101001111010011010101000000110001001110000011000100111001",123456789012345)
print("Test 7:") # Time taken: 14.988570 seconds
main("010000110100111101001101010100000011000100111000001100010011100100100001",123456789012345678)
print("Test 8:") # Time taken: 20.946981 seconds
main("01000011010011110100110101010000001100010011100000110001001110010010000101000001",1234567890123456789)
print("Test 9:") # Time taken: 53.943795 seconds
main("0100001101001111010011010101000000110001001110000011000100111001001000010100000101000100",1234567890123456789)
print("Test 10:") # Time taken: 71.572761 seconds
main("010000110100111101001101010100000011000100111000001100010011100100100001010000010100010001010011",12345678901234567890)

```

## Appendix A.2 - Proposed solution 2 (code written by Patwardhan Liladhar)

### Using 'list' as Data Structure.

```

import time # Imported time module to measure the execution time of the code

def prime(number):
    # Check if a number is prime
    if number < 2:
        return False # Number smaller than 2 is not prime
    for i in range(2, int(number ** 0.5) + 1):
        if number % i == 0:
            return False
    return True

def find_prime(binary_string, maximum_number):
    # Find all prime numbers hidden in binary numbers
    prime_found = [] # List of prime numbers found
    string_length = len(binary_string)

    for i in range(string_length):
        for j in range(i + 1, string_length + 1):
            substring = binary_string[i:j] # Extract the substring
            decimal = int(substring, 2) # Convert binary to decimal
            # Check if the decimal is less than input limit and is a prime number
            if decimal < maximum_number and prime(decimal) and decimal not in prime_found:
                # Ensured if the prime number is unique before adding to the list
                prime_found.append(decimal)
    prime_found.sort() # Sorting the list of prime numbers for better readability

```

[illegible]

```

end_time = time.time()

# Calculate and print the running time
running_time = end_time - start_time
print(f"Running time: {running_time:.6f}seconds")

```

## Appendix A.3 – Optimised solution

### Using `set()` as Data Structure.

```

"""
This program takes an input of binary string and extracting substrings,
checks if a number is prime and then sorting the result.
The final result will be fewer than 6 prime numbers.
Using set() as a data structure.
"""

def is_prime(n):
    """
    prime-checking function using 6k ± 1 method
    """
    if n < 2:
        return False
    if n in (2, 3):
        return True
    if n % 2 == 0 or n % 3 == 0: # eliminating multiples of 2 and 3
        return False
    if n < 9:
        return True
    k, limit = 5, int(n ** 0.5) + 1 # starting to check from 5 up to the square root of n
    while k <= limit:
        if n % k == 0 or n % (k + 2) == 0: # 6k + 1 method to check divisibility
            return False
        k += 6 # adding 6 to check the pair of divisors
    return True # if no divisors

def extract_binary_substrings(binary_string):
    """ extracts all possible decimal values from given binary substrings """
    if any(c not in '01' for c in binary_string): # checks if the string contains only '0'
or '1'
        return None

    substrings = set() # store decimal values
    length = len(binary_string)
    for i in range(length):
        decimal = 0 # Decimal equivalent
        for j in range(i, length):
            # converting binary -> decimal
            decimal = (decimal << 1) | int(binary_string[j]) # binary-to-decimal
conversion by moving bits to the left
            substrings.add(decimal) # adding the decimal value to the set
    return substrings

def main_primes(binary_str, n):
    """ main function to find and format prime numbers in binary substrings """

```

[illegible]

## Appendix B - Further evidence of team contribution if necessary

Such as chat log, meeting log ...