|  |  |  |
| --- | --- | --- |
| Name | ID | Contribution |
| Elias Andrew Ailenei | 00139XXXX | 100% |
| Lloyd Vichitsophaphan | 00138XXXX | 75% |
| Caleb Nimo | 00139XXXX | 75% |
| Safin Chowdhury | 00137XXXX | 50% |
| Uzair Jamal | 00140XXXX | 0% |

Table of Contents

[1) Creating two basic solutions 3](#_Toc193141362)

[1.1) Understanding the Problem 3](#_Toc193141363)

[Elias Andrew Ailenei 3](#_Toc193141364)

[Lloyd Vichitsophaphan 3](#_Toc193141365)

[Caleb Nimo 3](#_Toc193141366)

[Safin Chowdhury 3](#_Toc193141367)

[1.2) Solution implementation 4](#_Toc193141368)

[2) Testing and comparing solutions 5](#_Toc193141369)

[2.1) Test Cases & Validation 5](#_Toc193141370)

[2.2) Running on Given Test Cases (check appendix for high quality) 5](#_Toc193141371)

[2.3) Comparison of Solutions 6](#_Toc193141372)

[3) Optimising solutions 7](#_Toc193141373)

[3.1) Selecting a Solution for Optimisation (check appendix for high quality) 7](#_Toc193141374)

[3.2) Optimisation Steps & Justification 7](#_Toc193141375)

[4) Comparing performance 9](#_Toc193141376)

[4.1) Performance Comparison 9](#_Toc193141377)

[4.2) Running on Given Test Cases (check appendix for high quality) 9](#_Toc193141378)

[4.3) Visualising Performance 10](#_Toc193141379)

[5) Reflecting on teamwork 11](#_Toc193141380)

[Appendix 12](#_Toc193141381)

[Solution 1 12](#_Toc193141382)

[Solution 2 14](#_Toc193141383)

[Solution 3 16](#_Toc193141384)

[Images for handwritten examples 19](#_Toc193141385)

[Elias 19](#_Toc193141386)

[Safin 21](#_Toc193141387)

[Lloyd 22](#_Toc193141388)

[Caleb 23](#_Toc193141389)

[Results of images upscaled for 2.2 24](#_Toc193141390)

[Results of images upscaled for 3.1 26](#_Toc193141391)

[Results of images upscaled for 4.2 27](#_Toc193141392)

[References 28](#_Toc193141393)

[Bibliography 28](#_Toc193141394)

[Non-academic sources (mostly YouTube videos) 28](#_Toc193141395)

# Creating two basic solutions

## Understanding the Problem

### Elias Andrew Ailenei

We get the binary value, and we keep splitting it from one number, and then adding the next number and so on until we reach the last position of the original string. We then work backwards to get any left out numbers. We convert these binary values into actual denary values. All results are stored in a set meaning that we do not get any duplicates. We then use an algorithm to generate some prime numbers up to N such as trial or sieve to do some comparisons. We then only keep the numbers from the set if they are prime numbers by saying is number seen in the prime list? If so, keep it otherwise discard it.

### Lloyd Vichitsophaphan

There are hidden binary values in each binary. We use a set of algorithms to find every possible binary value and then remove that possible value. The program repeats itself until there are no more values. Once that is done, I will convert it into denary. If there are any repetition it will be removed. Then we use another algorithm to filter out non prime numbers. A variable called “N” will ensure that any prime number value would be smaller. If there are more than six values, it will generate the last 3 values(biggest) and the first 3 values(smallest). Otherwise, it will just generate the values.

### Caleb Nimo

The issue is to find the implicit binary values of a given binary string by continuously dividing and adding numbers up to the end position. The process is reversed for finding any value that has been overlooked. The binary values are then converted to denary and added to a set to remove duplicates. To remove unwanted outcomes, an algorithm like the trial division or the sieve method generates primes up to N. The numbers that are present in the prime list are retained. If more than six values remain, the three lowest and three highest are returned. Otherwise, they are all returned. This maintains the efficiency with different input sizes.

### Safin Chowdhury

We begin with a binary number and make it smaller by spitting it by the digits each time, then it proceeds to add to the next once each time, until the end of the binary string is reached. Then it will begin to go backwards catching any numbers that might have missed out. Then it converts all the binary strings into decimal substrings, the results are stored in a set to avoid duplicates. Then an algorithm is used like a sieve method or trial division to generate prime numbers up to N. finally the sets are checked and if the numbers are in the list prime numbers the number will stay and if not, it will be discarded.

|  |  |
| --- | --- |
| Handwritten example – see appendix for high quality screenshot | By |
| A close-up of a white paper  AI-generated content may be incorrect. | Elias |
| A close-up of a whiteboard  AI-generated content may be incorrect. | Elias – NOTE MY 9’s looks like 4’s |
| A close-up of a notepad  AI-generated content may be incorrect. | Safin |
|  | Caleb |
|  | Lloyd |

## Solution implementation

We have been tasked with creating two solutions. The first solution will first convert the hidden binary strings into denary values and only use trail division to find the primes which has the worst times of solutions (Cormen, 2009). We also have solution 2 which does the same hidden binary extraction to denary but instead of only using trial (Tarafder, 2019), solution 2 now uses a hybrid approach of Sieve of Eratosthenes and trial which speeds up the results for bigger numbers and helps the solution to pass by reducing the execution time by 2/4 seconds with big N numbers (Luo, 1989) .

# Testing and comparing solutions

## 2.1) Test Cases & Validation

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **#** | **Input** | **Output** | **Pass/Fail** | **Running time (s)** | **Screenshot** |
| 1 | 11011 N = 30 | 5: 2, 3, 5, 11, 13 | PASS | 86700 ns | A black background with white text  AI-generated content may be incorrect. |
| 2 | 11101101 N = 100 | 8: 2, 3, 5, ..., 13, 29, 59 | PASS | 77400 ns |  |
| 3 | 101011 N = 88 | 5: 2, 3, 5, 11, 43 | PASS | 82100 ns | A black background with white text  AI-generated content may be incorrect. |
| 4 | 10101 N = 40 | 2: 2, 5 | PASS | 32.200 µs |  |
| 5 | 1001 N = 30 | 1: 2 | PASS | 35.800 µs |  |

## 2.2) Running on Given Test Cases (check appendix for high quality)

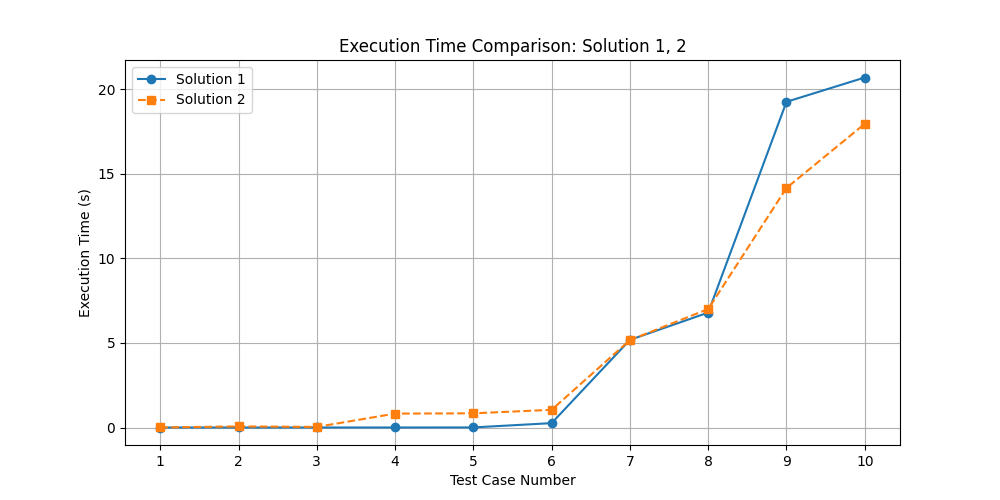
|  |  |  |  |
| --- | --- | --- | --- |
| **#** | **Sol1** | **Sol2** | **PASS/FAIL** |
| 1 |  |  | PASS |
| 2 |  |  | PASS |
| 3 |  |  | PASS |
| 4 |  |  | PASS |
| 5 |  |  | PASS |
| 6 |  |  | PASS |
| 7 |  |  | PASS |
| 8 |  |  | PASS |
| 9 |  |  | PASS |
| 10 |  |  | PASS |

## 2.3) Comparison of Solutions

In the coursework, we have developed two solutions to extract unique prime numbers hidden within a binary string which each uses different data structures and different time complexities. The first solution uses sets and lists, where the set helps to prevent duplication in the list and the list help to create the prefix sums to extract the decimal values from the binary substrings. Trial division is used in the first solution (Cormen, 2009).

The second solution makes uses of lists and Sieve of Eratosthenes for its prime checking (Tarafder, 2019). Sieve of Eratosthenes is a list-based sieve algorithm that efficiently precomputes prime numbers up to a constant limit such as 10 million (Luo, 1989) (this number is picked for its balance between accuracy and speed). This approach also reduces the need for repeatedly checking for primes individually which helps for the time complexity. However, even though the time complexity is better than trial division, it comes with the trade-off which is taking up a lot of memory (Ghidarcea, 2024). This might lead to some systems crashing due to a Memory Overflow error if they have low RAM and was given a large N number.

Another point with the second solution is that when N exceeds the sieve limit, the second solution defaults back to trial division for numbers beyond the precomputed range (Harahap, 2019), meaning that while smaller primes are found much faster, the larger numbers still require trial division just like in solution 1.



When it comes to time complexity, solution 1 runs at , where is the length of the binary string and is the largest number extracted (Cormen, 2009). Solution 2 (when only using Sieve) has a complexity of , where L is substring extraction, is the sieve limit, and is just some constant overhead (Luo, 1989). But if the sieve needs to switch back to trial division, the complexity shifts to , which shows how it balances speed and memory usage (Harahap, 2019).

At the end of the day, both solutions have their strengths and weaknesses—solution 1 is simpler and uses less memory but takes longer, while solution 2 is way faster but can be heavy on system resources (Ghidarcea, 2024). The best choice depends on how large is and how much memory the system must work with.

# Optimising solutions

## 3.1) Selecting a Solution for Optimisation (check appendix for high quality)

|  |  |  |  |
| --- | --- | --- | --- |
| **#** | **Input** | **Output with time** | **Pass/Fail** |
| 1 | "1010101011" \* 20 N: 123456789012345678901 |  | Pass |
| 2 | "1" \* 200 N: 1234567890123456789012345678901 |  | Pass |
| 3 | "1100110111010001110100010101100010111011" \* 20  N: 12345678901234567890123456789 |  | Pass |
| 4 | "000000000000000000001" + "110100101001" \* 20 N: 123456789012345678901 |  | Pass |
| 5 | "1001100110011001100110011001100110011001" \* 20 N: 12345678901234567890123456789 |  | Pass |

## 3.2) Optimisation Steps & Justification

In this improved version, we build upon our basic solution 2 from Task 1 but keep the core flow of extracting numbers from the binary string using a Sieve approach. We make the Sieve itself more efficient by only segmenting and using it when it’s needed (Luo, 1989), plus we bring in a stronger factorization method (Pollard’s Rho) and a Fermat based primality test (Rabin, 1980) (Pollard, 1975). Together, these additions let the program handle a wide range of numeric sizes far more quickly often within sub second times. First off, numbers under 10,000 still go through trial division. As we learned from our original solutions, trial division is perfectly fine when the values are that small. However, once we jump past 10,000 and up to 10 million, we turn to a segmented Sieve of Eratosthenes (Atkin, 2004), which sieves only the low and high range of candidate numbers. This approach cuts down on memory and avoids time lost sieving from 1 all the way up to the highest candidate.

For numbers that go beyond 10 million, we do a quick Fermat primality test first. Fermat’s little theorem essentially tells us that if is prime, then for any integer not divisible by . We check this with a few random ‘a’ value if one fails, we know is composite right away (Rabin, 1980). While this test can occasionally be fooled by “Carmichael numbers,” those cases are rare, and it’s super fast in practice.

If the number still isn’t ruled out by the Fermat test, we move on to Pollard’s Rho, which is a randomized factorization algorithm. Instead of dividing by every number up to ​, Pollard’s Rho uses a polynomial function (commonly and some advanced math to detect nontrivial factors more quickly (Pollard, 1975). It’s much faster than naive trial division for larger inputs often cutting the expected runtime dramatically.

Thanks to this combination (segmented Sieve for medium ranges, Fermat + Pollard’s Rho for large ranges), we avoid the worst-case scenarios of traditional approaches. In Big O terms, we no longer rely purely on for large . Pollard’s Rho typically finds factors in around or better under average conditions, while the segmented Sieve runs in roughly for an interval size (Ghidarcea, 2024). These are big improvements over naive methods, especially for large inputs.

We’ve also preserved the prefix array from our previous solution, so each binary substring can be converted to a decimal in time. This avoids re checking and parsing the same substring repeatedly. Overall, these updates give us a solution that’s both more scalable and consistently faster than our original Solution 2 while still building on its foundation rather than scrapping it entirely. The only slight downside is the tiny probabilistic risk of Fermat’s test being tricked, but that’s rare enough compared to the major speed gains we get in return.

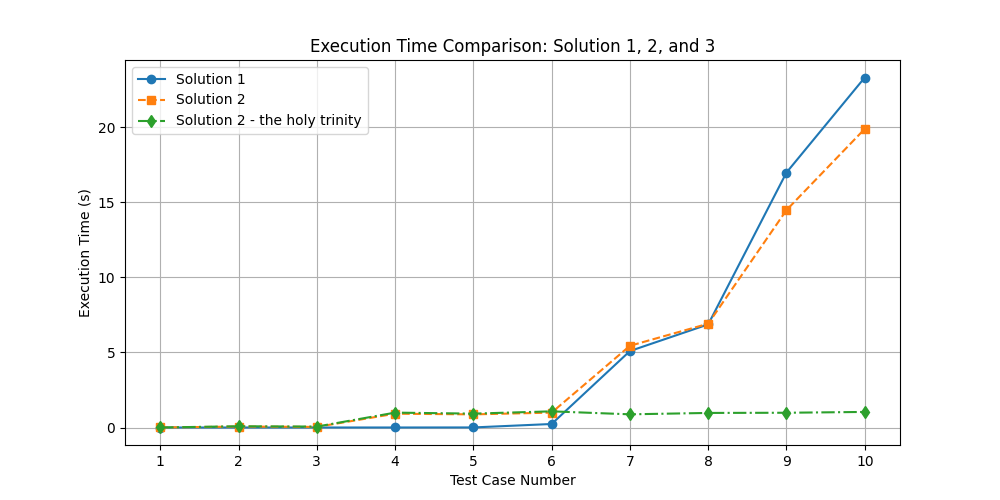
# Comparing performance

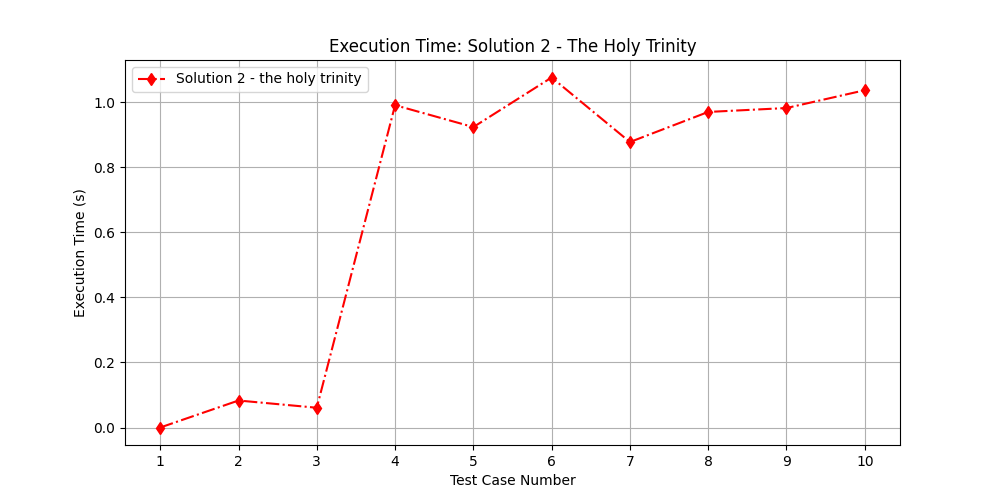
## 4.1) Performance Comparison

This new solution is more modular and efficient than the solutions seen in Task 1 as its focus is to tackle memory usage and reduce excess calculations for prime checking. Both solutions use sets to store unique prime numbers and prefix sum arrays to efficiently convert binary substrings into useable numbers for Python. The main difference between these solutions, reguarding their data structure, is that this new solution optimizes performance by using a Boolean array in its segmented sieve implementation which helps to reduce memory usage and improve lookup times (Luo, 1989). As a result, this makes the new solution more memory efficient when handling large numbers (or even extreme numbers) unlike the previous solution 2 which relies on a more basic approach on implementing Sieve which thus leads to it becoming more inefficient when given large inputs (Atkin, 2004). Furthermore, reguarding algorithmic differences, the main difference between these solutions is in their primality testing strategies (see if it’s a prime or not algorithms). This new solution has three stages that it can go through to get the fastest numbers: brute force stage, look-up stage and random generation stage (Rabin, 1980). Unlike the previous solution which used trail division for larger numbers, this new solution uses trail division to brute force small numbers since its less expensive for our resources to do that and it always comes up faster, we usually do this with numbers that are less than or equal to 10,000 . If we still have more numbers above the trail threshold, we move onto using segmented sieve which means we only sieve from where trail has ended to where we need which greatly reduces memory overhead (Ghidarcea, 2024), we usually give Sieve a 10 million limit . In the previous solution, Sieve was used for small numbers and always started from 1 to which was inefficient. Lastly, if the numbers are still greater than the Sieve limit, we will use our combination of Fermat’s primality test with Polland Rho’s algorithm to quickly find the remaining (Pollard, 1975) (Rabin, 1980). Lastly the main difference between these approaches, reguarding implementation, is that is new solution follows a more modular design which helps to improve maintainability and scalability whereas the previous solution was straightforward but was lacking modularity thus not being able to handle large cases like this new solution (Harahap, 2019).

## 4.2) Running on Given Test Cases (check appendix for high quality)

|  |  |  |
| --- | --- | --- |
| **#** | **Sol3 (improved Sol2)** | **PASS/FAIL** |
| 1 |  | PASS |
| 2 |  | PASS |
| 3 |  | PASS |
| 4 |  | PASS |
| 5 |  | PASS |
| 6 |  | PASS |
| 7 |  | PASS |
| 8 |  | PASS |
| 9 |  | PASS |
| 10 |  | PASS |





## 4.3) Visualising Performance

Firstly, we do our binary conversion by extracting all possible substrings and converting them into numbers. Since we use prefix sums, we can assume that each substring can be computed at and the extraction being at for an input of length (Cormen, 2009). We then move onto trial division, which is used for small numbers under or equal to 10,000 giving us the worst-case scenario of per number (Harahap, 2019). We also have segmented sieve which starts from 10,001 to 10,000,000 which gives the complexity of for numbers up to (Luo, 1989). If the numbers are still greater than 10,000,000, we then move to our duo of Polland Rho and Fermat’s test which gives us a complexity of (Pollard, 1975) (Rabin, 1980). When the N is extremely large, as in the area, most candidates will be checked with Polland Rho (Ghidarcea, 2024). The final worse case scenario for this solution is:

# Reflecting on teamwork

|  |  |  |
| --- | --- | --- |
| **Name with ID** | **Contribution** | **Description of work** |
| Elias Andrew Ailenei  001392302 | 100% | Completed Solution 2 and Solution 3 and handled most of the work, including testing, optimization, and performance comparison. |
| Lloyd Vichitsophaphan  001380982 | 75% | Developed Solution 1 with Caleb, wrote a paragraph for creating two basic solutions, and contributed a handwritten example. |
| Caleb Nimo  001399850 | 75% | Developed Solution 1 with Lloyd, wrote a paragraph for creating two basic solutions, and contributed a handwritten example. |
| Safin Chowdhury  001379463 | 50% | Wrote a paragraph and provided a handwritten example for creating two basic solutions. |
| Uzair Jamal  001403511 | 0% | Didn’t provide their contact details to join group so considered MIA. |

|  |  |  |
| --- | --- | --- |
| **Week** | **Task** | **Contribution** |
| Week 1 | MVPs for tasks 1 and 2 | Lloyd and Caleb on Task 1, Elias and Safin on Task 2 (Safin gave no input) |
| Week 2 | Improving tasks 1 and 2 | Lloyd and Caleb on Task 1, Elias on Task 2 |
| Week 3 | Tasks 1 and 2 finished, research on task 3 | Lloyd and Caleb on Task 1, Elias on Task 2 and 3 |
| Week 4 | All tasks finished, getting ready for documentation | Lloyd and Caleb on Task 1, Elias on Task 2 and 3 |
| Week 5 | Documenting everything | Majority Elias, rest of team gave their required input |

# Appendix

## Solution 1

def check(binary):

    return all(index in "01" for index in binary) # this will check if the binary string is valid

def hidden(binary):

    data = set() # no dupes allowed

    length = len(binary)

    if check(binary): # if the binary string is valid, then it will convert it to a decimal number

        prefix = [0] \* (length + 1)

        for i in range(length):

            prefix[i + 1] = (prefix[i] << 1) + int(binary[i])

        for i in range(length):

            for j in range(i + 1, length + 1):

                val = prefix[j] - (prefix[i] << (j - i))

                if val > 1:

                    data.add(val)

    return sorted(data)

def is\_prime(n): # only using trial division here, very simple and fast

    if n < 2: # even numbers are not prime

        return False

    if n in {2, 3, 5, 7, 11}: # these are the only prime numbers that are not divisible by 2 or 3

        return True

    if n % 2 == 0 or n % 3 == 0: # if the number is divisible by 2 or 3, then it is not prime

        return False

    i = 5

    while i \* i <= n: # this will iterate through the number and check if it is prime

        if n % i == 0 or n % (i + 2) == 0:

            return False

        i += 6

    return True

def sort(array, less): # this will sort the array and check if the number is prime

    array = [num for num in array if num <= less and is\_prime(num)]

    array = sorted(set(array))

    if len(array) == 0:

        return "No primes found."

    elif len(array) < 6:

        return f"{len(array)}: {', '.join(map(str, array))}"

    else:

        last\_three = [f"{n}" for n in array[-3:]]

        return f"{len(array)}: {', '.join(map(str, array[:3]))}, ..., {', '.join(last\_three)}"

def final(array, less):

    return (sort(array, less))

def extract\_primes(array, less):

    if not all(index in "01" for index in array):

        return "0: Invalid binary strings "

    if type(less) is not int:

        return "0: Invalid N given "

    values = hidden(array)

    return final(values, less)

if \_\_name\_\_ == "\_\_main\_\_":

    print(extract\_primes(input("Enter a binary string: "), int(input("Enter N: "))))

# Code made by Lloyd and Caleb. Reviewed and commented by Elias Andrew Ailenei (github.com/eliasailenei)

## Solution 2

import math # built-in libs, allowed by spec

def sieve\_upto(n): # this is a limit that the sieve will go up to since it has a memory limit and has a bad diminishing marginal return when the N is really big

    is\_prime = [True] \* (n + 1)

    is\_prime[0] = is\_prime[1] = False # 0 and 1 are not prime numbers

    for i in range(2, int(n\*\*0.5) + 1):

        if is\_prime[i]:

            for j in range(i\*i, n + 1, i): # this will iterate through the list of numbers and determine if they are prime or not

                is\_prime[j] = False

    return is\_prime, [x for x in range(2, n + 1) if is\_prime[x]] # this will return the list of prime numbers

def is\_prime\_trial(n): # this is a simple trial division method to check if the number is prime

    if n < 2:

        return False

    if n < 4:

        return True

    if n % 2 == 0 or n % 3 == 0:

        return False

    limit = int(math.isqrt(n)) # this will get the square root of the number and set it as a limit

    i = 5

    while i <= limit:

        if n % i == 0 or n % (i + 2) == 0:

            return False

        i += 6

    return True

def extract\_primes(binary\_str, N): # main function

    if not all(index in "01" for index in binary\_str): # edge case : only 0 and 1 are allowed

        return "0: Invalid binary strings "

    if type(N) is not int: # edge case : N must be an integer

        return "0: Invalid N given "

    candidates = set() # no dupes allowed

    length = len(binary\_str)

    prefix = [0] \* (length + 1)

    for i in range(length): # this will convert the binary string to a decimal number

        prefix[i+1] = (prefix[i] << 1) + (binary\_str[i] == '1')

    for start in range(length): # this will iterate through the binary string

        for end in range(start + 1, length + 1): # this will iterate through the binary string

            val = prefix[end] - (prefix[start] << (end - start))

            if val >= N:

                break

            if val > 1:

                candidates.add(val) # this will add the number to the set

    if not candidates:

        return []

    max\_candidate = max(candidates) # this will get the max number from the set

    SIEVE\_THRESHOLD = 10\_000\_000  # this constant value is not so high on the curve but is good enought to produce the speed needed

    primes\_found = []

    if max\_candidate <= SIEVE\_THRESHOLD: # if the max number is less than the threshold, then it will use the sieve

        is\_prime\_small, \_ = sieve\_upto(max\_candidate) # this will get the prime numbers

        for val in sorted(candidates):

            if is\_prime\_small[val]: # if the number is prime, then it will add it to the list

                primes\_found.append(val)

    else: # if the number is greater than the threshold, then it will use the hybrid approach

        is\_prime\_small, small\_primes = sieve\_upto(SIEVE\_THRESHOLD)

        for val in sorted(candidates):

            if val <= SIEVE\_THRESHOLD:

                if is\_prime\_small[val]:

                    primes\_found.append(val)

            else:

                if is\_prime\_trial(val):

                    primes\_found.append(val)

    if not primes\_found:

        return "No primes found."

    count = len(primes\_found)

    if count < 6:

        return f"{count}: {', '.join(map(str, primes\_found))}"

    else:

        return (f"{len(primes\_found)}: {primes\_found[0]}, {primes\_found[1]}, {primes\_found[2]}, ..., "

                f"{primes\_found[-3]}, {primes\_found[-2]}, {primes\_found[-1]}")

if \_\_name\_\_ == "\_\_main\_\_":

   print(extract\_primes(input("Enter binary string: "), int(input("Enter N: "))))

# Code made and maintained by Elias Andrew Ailenei (github.com/eliasailenei)

## Solution 3

import math,random # built-in libs, allowed by spec

def sieve\_segment(low, high): # sieve is going to gen numbers from x to y instead of 1 to n

    sieve\_range = high - low + 1

    is\_prime = [True] \* sieve\_range # this will set all numbers to prime

    if low == 1: is\_prime[0] = False

    limit = int(math.sqrt(high))

    small\_primes = sieve\_upto(limit)[1] # this will get the prime numbers

    for prime in small\_primes: # this will iterate through the prime numbers

        start = max(prime \* prime, low + (prime - low % prime) % prime) # this will get the start of the prime number

        for j in range(start, high + 1, prime):

            is\_prime[j - low] = False

    return is\_prime, [low + i for i in range(sieve\_range) if is\_prime[i]] # this will return the prime numbers

def sieve\_upto(n): # the main sieve function

    is\_prime = [True] \* (n + 1)

    is\_prime[0] = is\_prime[1] = False

    for i in range(2, int(n \*\* 0.5) + 1):

        if is\_prime[i]:

            for j in range(i \* i, n + 1, i):

                is\_prime[j] = False

    return is\_prime, [x for x in range(2, n + 1) if is\_prime[x]]

def is\_prime\_trial(n): # this is a simple trial division method to check if the number is prime

    if n < 2: return False

    if n in (2, 3): return True

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

    limit = int(math.sqrt(n)) # this will get the square root of the number and set the limit

    i = 5

    while i <= limit:

        if n % i == 0 or n % (i + 2) == 0: return False

        i += 6

    return True

def is\_prime\_fermat(n, k=5): # this is the fermat primality test to check if the number is prime

    if n < 2: return False

    if n in (2, 3): return True

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

    for \_ in range(k): # this will iterate through the number

        a = random.randint(2, n - 2)

        if pow(a, n - 1, n) != 1: return False

    return True

def pollards\_rho(n): # this is the pollards rho algorithm to check if the number is prime

    if n % 2 == 0: return 2

    x = random.randint(2, n - 1); y = x; c = random.randint(1, n - 1); d = 1

    def f(x): return (x \* x + c) % n

    while d == 1: x = f(x); y = f(f(y)); d = math.gcd(abs(x - y), n) # using the greatest common divisor to check if the number is prime

    return d if d != n else None

def is\_prime\_large(n): # this is the main function to check if the number is prime for a large N

    if is\_prime\_fermat(n): return True

    factor = pollards\_rho(n)

    if factor and factor != n: return False

    return True

def extract\_primes(binary\_str, N): # the main function

    if not all(index in "01" for index in binary\_str): return "0: Invalid binary strings" # edge case : only 0 and 1 are allowed

    if type(N) is not int: return "0: Invalid N given" # edge case : N must be an integer

    candidates = set() # no dupes allowed

    length = len(binary\_str)

    prefix = [0] \* (length + 1)

    for i in range(length): # this will convert the binary string to a decimal number

        prefix[i+1] = (prefix[i] << 1) + (binary\_str[i] == '1')

    for start in range(length):

        for end in range(start + 1, length + 1):

            val = prefix[end] - (prefix[start] << (end - start))

            if val >= N: break

            if val > 1: candidates.add(val)

    if not candidates: return "No primes found."

    max\_candidate = max(candidates)

    TRIAL\_THRESHOLD = 10\_000 # this is the threshold for the trial division (balanced)

    SIEVE\_THRESHOLD = 10\_000\_000 # this is the threshold for the sieve (balanced)

    primes\_found = []

    trial\_checked = set()

    for val in sorted(candidates): # this will iterate through the candidates

        if val <= TRIAL\_THRESHOLD: # if the number is less than the threshold, then it will use the trial division

            if is\_prime\_trial(val):  # if the number is prime, then it will add it to the list

                primes\_found.append(val)

            trial\_checked.add(val)

    medium\_candidates = sorted(n for n in candidates if TRIAL\_THRESHOLD < n <= SIEVE\_THRESHOLD) # this will get the medium candidates

    if medium\_candidates: # if there are medium candidates

        low, high = min(medium\_candidates), max(medium\_candidates) # get the min and max of the medium candidates

        is\_prime\_segmented, primes\_from\_sieve = sieve\_segment(low, high)  # this will get the prime numbers from a limit

        for val in medium\_candidates:

            if val - low >= 0 and is\_prime\_segmented[val - low]: # if the number is prime, then it will add it to the list

                primes\_found.append(val)

    for val in sorted(n for n in candidates if n > SIEVE\_THRESHOLD): # if the number is greater than the threshold, then it will use the hybrid approach

        if is\_prime\_large(val):

            primes\_found.append(val) # if the number is prime, then it will add it to the list

    if not primes\_found:

        return "No primes found."

    count = len(primes\_found)

    if count < 6:

        return f"{count}: {', '.join(map(str, primes\_found))}"

    else:

        return (f"{len(primes\_found)}: {primes\_found[0]}, {primes\_found[1]}, {primes\_found[2]}, ..., "

                f"{primes\_found[-3]}, {primes\_found[-2]}, {primes\_found[-1]}")

if \_\_name\_\_ == "\_\_main\_\_":

    print(extract\_primes(input("Enter binary string: "), int(input("Enter N: "))))

    # Code made and maintained by Elias Andrew Ailenei (github.com/eliasailenei)

## Images for handwritten examples

### Elias

A close-up of a whiteboard

AI-generated content may be incorrect.

A whiteboard with writing on it

AI-generated content may be incorrect.

### Safin

A close-up of a math problem

AI-generated content may be incorrect.

### Lloyd

A close-up of a paper

AI-generated content may be incorrect.

### Caleb

A notebook with writing on it

AI-generated content may be incorrect.

## Results of images upscaled for 2.2

|  |  |
| --- | --- |
| **#** | **Sol1** |
| 1 | A black background with white text  AI-generated content may be incorrect. |
| 2 | A black background with white text  AI-generated content may be incorrect. |
| 3 | A black background with white text  AI-generated content may be incorrect. |
| 4 | A black background with white numbers  AI-generated content may be incorrect. |
| 5 | A black background with white numbers  AI-generated content may be incorrect. |
| 6 |  |
| 7 |  |
| 8 |  |
| 9 |  |
| 10 |  |

|  |  |
| --- | --- |
| **#** | **Sol2** |
| 1 | A black background with white text  AI-generated content may be incorrect. |
| 2 | A black background with white text  AI-generated content may be incorrect. |
| 3 | A black background with white text  AI-generated content may be incorrect. |
| 4 | A black background with white numbers  AI-generated content may be incorrect. |
| 5 | A black background with white numbers  AI-generated content may be incorrect. |
| 6 |  |
| 7 |  |
| 8 |  |
| 9 |  |
| 10 |  |

## Results of images upscaled for 3.1

|  |  |
| --- | --- |
| **#** | **Output with time** |
| 1 |  |
| 2 |  |
| 3 |  |
| 4 |  |
| 5 |  |

## Results of images upscaled for 4.2

|  |  |
| --- | --- |
| **#** | **Sol3 (improved Sol2)** |
| 1 | A black background with white text  AI-generated content may be incorrect. |
| 2 | A black background with white text  AI-generated content may be incorrect. |
| 3 | A black background with white text  AI-generated content may be incorrect. |
| 4 | A black background with white text  AI-generated content may be incorrect. |
| 5 | A black background with white numbers  AI-generated content may be incorrect. |
| 6 |  |
| 7 |  |
| 8 |  |
| 9 |  |
| 10 |  |

## References

# Bibliography

Atkin, A. a. B. D., 2004. Prime sieves using binary quadratic forms. *Mathematics of Computation.*

Cormen, T. L. C. R. R. a. S. C., 2009. *Introduction to Algorithms.* 3rd ed. Cambridge, MA: MIT Press.

Ghidarcea, M. a. P. D., 2024. Prime number sieving — A systematic review with performance analysis. *Algorithms.*

Harahap, M. a. K. N., 2019. The comparison of methods for generating prime numbers between the Sieve of Eratosthenes, Atkins, and Sundaram. *Sinkron: Jurnal dan Penelitian Teknik Informatika.*

Luo, X., 1989. A practical sieve algorithm finding prime numbers. *Communications of the ACM.*

Pollard, J., 1975. A Monte Carlo method for factorization. *BIT Numerical Mathematics.*

Rabin, M., 1980. Probabilistic algorithm for testing primality. *Journal of Number Theory.*

Tarafder, A. a. C. T., 2019. *A comparative analysis of general, sieve-of-Eratosthenes and Rabin-Miller approach for prime number generation.* Cox’s Bazar, Bangladesh, Proceedings of the 2019 International Conference on Electrical, Computer and Communication Engineering (ECCE), IEEE.

### Non-academic sources (mostly YouTube videos)

[How to Find VERY BIG Prime Numbers?](https://www.youtube.com/watch?v=X9kL2fbEC9U)

[How To Quickly Factor a Number: Pollard's Rho Algorithm](https://www.youtube.com/watch?v=7lhlJTtCsiw)

[Fermat's Little Theorem ← Number Theory](https://www.youtube.com/watch?v=w0ZQvZLx2KA)

[Sieve of Eratosthenes | Journey into cryptography | Computer Science | Khan Academy](https://www.youtube.com/watch?v=klcIklsWzrY)

[Primality test](https://en.wikipedia.org/wiki/Primality_test)