Cryptography Project 1

João Pedro Lourenço (jo1360lo-s)
Group 12

November 2020

Working hours

The time taken to solve this project was about 12 hours.

Exercise 1

The largest 25 digit number is smaller than 10^{25}, and the largest 12 digit number (max size of a prime factor) is smaller than 10^{12}. If we can test 10 million numbers per second, that means that we will take at most $\frac{10^{12}}{10^7} = 10^5$ seconds, or around 2 hours and 46 minutes, to factor a 25 digit number.

Exercise 2

According to the prime-number theorem, an estimation of the number of primes smaller than n can be given by the following formula:

$$\pi(n) = n/\ln(n) \tag{1}$$

If we consider $n = 10^{12}$, then $\pi(n) = 3,62 \times 10^{10}$. Following the same logic as before, the factorization will take at most around $\frac{3,62 \times 10^{10}}{10^7} = 3619$ seconds, which is about 1 hour.

If we consider that each integer requires 8 bytes to represent, then we would need about $3,62 \times 10^{10} \times 8 = 2,896 \times 10^{11}$ bytes, or around 290GB. Such a capacity is definitely within student budget, as a 512GB HDD can be bought for 329 SEK [1].

Exercise 3

The exercise was solved with Python and all standard libraries. It's worth noting, however, that it makes use of the provided `GaussBin.exe` program, so it only runs on Windows.

Structure

The program can be separated into 4 different stages:

1. **Generate the factor base list**. Given the desired factor base size as input, create a list with that size containing the first factor base size primes. The primes come from the provided .txt file.

   Done by the `generate_factor_base_list` method.

2. **Find r values and create the binary matrix**. With the primes previously found, use the technique explained in the project description to find at most factor base size + 2 r values and their factorization, as well as build the binary matrix with that information.

   Done by the `generate_r_values_and_matrix` method.

[1] Western Digital 500GB Sata III on amazon.se

3. **Perform the gaussian elimination**. The matrix is written to an input file properly formatted, the `GaussBin` program is run, and then the results are gathered from the output of `GaussBin`.

   Done by the `gaussian_elimination` method.

4. **Find the gcd, if it exists**. Walk by all the solutions of `GaussBin`, and try to find one that satisfies the required constraints.

   Done by the `find_gcd` method.

Results

Using a factor base of 750, these are the obtained results:

Input	Time (secs)	Result
323	21.706	17, 19
307561	2.695	457, 673
31741649	2.695	4621, 6869
3205837387	2.792	46819, 68473
392742364277	3.31801	534571, 734687
235616869893895625763911	48.0058	453131078611, 519975082301

Table 1: Results of running the program with the different inputs

The last input is the unique value to each group (in my case, it's the value for group 12).

The program

```python
import math, os, time, subprocess, decimal


def prime_factorization(n, prime_list):
    """
    Factorizes n with the primes found in prime_list
    """
    result = {}

    for index, prime in enumerate(prime_list):
        prime_counter = 0
        while n % prime == 0:
            n /= prime
            prime_counter += 1

        if prime_counter != 0:
            result[prime] = prime_counter

        if n == 1:
            break

    if n != 1:
        return {}

    return result


def gcd(a,b):
```

```python
    while b:
        b, a = a % b, b
    return a


def generate_matrix_row(factorization, factor_base_list):
    """
    Creates a row for the binary matrix, with a given factorization
    """
    new_row = []

    # have to iterate over the whole factor base, as that's the length of the row
    for prime in factor_base_list:
        if prime in factorization:
            new_row.append(
                factorization[prime] % 2
            )  # append 1 or 0, depending on the exponent being odd or even
        else:
            new_row.append(0)
    return new_row


def find_gcd(solutions, r_values, given_number):
    """
    Attempts to find the gcd based on the solutions of the binary matrix
    """
    decimal.getcontext().prec = 1024  # Start with a decent precision

    for index, solution in enumerate(solutions):
        calculated = False
        while not calculated:
            try:
                x = decimal.Decimal(1)
                y = decimal.Decimal(1)

                # Iterate over the current solution to find and accumulate the
                # appropriate the r_values
                for equation_number, matrix_value in enumerate(solution):
                    if matrix_value == 1:
                        x *= r_values[equation_number][0] % given_number
                        for prime, count in r_values[equation_number][1].items():
                            y *= prime ** count

                y = int(decimal.Decimal(y.sqrt())) % given_number

                # Given x and y, get gcd
                gcd_ = int(gcd(abs(x - y), given_number))

                if gcd_ != 1 and gcd_ != given_number:  # Acceptable, is solution
                    other_factor = int(given_number / gcd_)
                    if other_factor > gcd_:
                        return gcd_, other_factor
                    else:
                        return other_factor, gcd_
```

```
                calculated = True  # no problems, can proceed to try the next solution

            except decimal.InvalidOperation:
                # Could not perform an operation at the current level of precision,
                # so double it and try again
                decimal.getcontext().prec *= 2

    return (-1, -1)


def gaussian_elimination(matrix):
    """
    Writes matrix to a file to be used as input to the provided program,
    returns all the solutions
    """
    m = len(matrix)
    if m == 0:
        return []
    n = len(matrix[0])

    # Write input file
    with open("matrix_input.txt", "w") as file:
        file.write("{} {}\n".format(m, n))
        for row in matrix:
            stringed_row = [str(i) for i in row]
            file.write(" ".join(stringed_row))
            file.write("\n")

    # Execute program
    with open(os.devnull, "wb") as devnull:
        subprocess.check_call(
            ["GaussBin.exe", "matrix_input.txt", "result.txt"],
            stdout=devnull,
            stderr=subprocess.STDOUT,
        )

    # Retrieve solutions
    with open("result.txt", "r") as output:
        raw_result = output.readlines()[
            1:
        ]  # first result is number of solutions, ignore

    # Convert into integers and a list of lists and for easier processing
    separated = [result.split() for result in raw_result]
    final_result = []
    for solution in separated:
        final_result.append([int(x) for x in solution])

    return final_result


def generate_r_values_and_matrix(
    k_start, k_stop, factor_base_size, factor_base_list, given_number
```

```python
):
    """
    Generates and returns the r_values, with their respective factorization,
    as well as the binary matrix.
    """
    r_values = []
    matrix = []
    r_values_seen = []

    for k in range(k_start, k_stop):
        for j in range(2, k + 1):
            r = math.floor(math.sqrt(k * given_number)) + j
            r_modulo = r ** 2 % given_number

            if r_modulo > 1 and r not in r_values_seen:
                factorization = prime_factorization(r_modulo, factor_base_list)

                # It might not be factorizable with the given factor base,
                # in which case we ignore this r value
                if factorization != {}:
                    new_row = generate_matrix_row(factorization, factor_base_list)

                    # Only accept the value in case it does not result in
                    # a repeated row
                    if new_row not in matrix:
                        matrix.append(new_row)
                        r_values.append((r, factorization))
                        r_values_seen.append(r)

            if len(r_values) >= factor_base_size + 2:
                return r_values, matrix

    return r_values, matrix


def generate_factor_base_list(factor_base):
    """
    Creates a list of primes with size factor_base.
    """
    primes = []
    with open("prim_2_24.txt", "r") as primes:

        # Primes are organized in lists of 10 primes, so the following list
        # comprehension returns all the lists of 10 primes until the list that
        # contains the factor_base prime.
        factor_base_list = [
            next(primes).split() for x in range(math.ceil(factor_base / 10))
        ]

        # Collapse all lists into one, and convert each number to integer
        flattened = [
            int(prime)
            for factor_base_sublist in factor_base_list
            for prime in factor_base_sublist
```

```
        ]

        # Only return until the size given by the argument
        return flattened[:factor_base]


def quadratic_sieve(given_number, factor_base_size):
    """
    Runs the simplified quadratic sieve algorithm
    """

    factor_base_list = generate_factor_base_list(factor_base_size)
    r_values, matrix = generate_r_values_and_matrix(
        2, factor_base_size + 1, factor_base_size, factor_base_list, given_number
    )
    solutions = gaussian_elimination(matrix)
    factor_1, factor_2 = find_gcd(solutions, r_values, given_number)

    if factor_1 != -1 and factor_2 != -1:
        return (factor_1, factor_2)
    else:
        return "Not found"
```