TMA4162 Computational algebra, Project 4

Andreas Moe

October 23, 2025

Task 1, Random Squares Factoring

- The random squares method requires factoring smaller numbers via trial division. This implementation returns a list of exponents corresponding to the factors in the factor base

```
def trial_division_factoring(N: int, factor_base: [int]) -> [int]:
    exponents = []
    for prime in factor_base:
        power = 0
        while N % prime == 0:
            N = N//prime
            power += 1
        exponents.append(power)
    if N == 1:
        return exponents
    else:
        return None
```

Listing 1: Trial Division Factoring

- Stage 1 of the algorithm generates relations of the form:

$$x^2 \equiv \prod_{i=1}^{s} {p_i}^{e_i} \pmod{N}$$

The code below return a vector of random x values and a corresponding matrix of e values.

```
def random_squares_stage_1(N: int, factor_base: [int]) -> tuple[np.
    ndarray, np.ndarray]:
    exponent_matrix = []
    xs = []
    while len(exponent_matrix) < len(factor_base) + 1:
        x = randint(1, N - 1)
        a = x ** 2 % N
        exponents = trial_division_factoring(a, factor_base)
        if exponents is not None:
            exponent_matrix.append(exponents)
            xs.append(x)
    return np.array(xs, dtype=object), np.array(exponent_matrix)
```

Listing 2: Stage 1

- The next step is to find a non-trivial linear dependence mod 2 for the exponents. The nullspace of the exponent matrix over $\mathbb{F}_2$ is computed using the galois library. The first element λ of the nullspace is then returned along another vector fs.

  fs $= \lambda \cdot exponents/2$. These two vectors will be used to compute X and Y whose squares will be congruent mod N.

```python
def non_trivial_lin_dep(exponent_matrix: np.ndarray) -> tuple[np.ndarray, np.ndarray]:
    GF = galois.GF(2)
    exp_matrix_mod2 = GF(exponent_matrix % 2)
    null_space = exp_matrix_mod2.T.null_space()
    lambdas = np.array(null_space[0], dtype=int)
    return lambdas, lambdas.dot(exponent_matrix) // 2
```

Listing 3: Non-trivial Linear Dependence

- X and Y are computed as follows:

$$X \equiv \prod_{j=1}^{t} {x_j}^{\lambda_j} \pmod N, \quad Y \equiv \prod_{i=1}^{s} {p_i}^{f_i} \pmod N$$

  where t = number of relations, s = size of factor base

```python
def compute_squares(N: int, xs: np.ndarray, lambdas: np.ndarray, fs: np.ndarray, factor_base: [int]) -> tuple[int, int]:
    X = np.prod([pow(int(x), int(l), N) for x, l in zip(xs, lambdas)], dtype=object) % N
    Y = np.prod([pow(p, int(f), N) for p, f in zip(factor_base, fs)], dtype=object) % N
    return X, Y
```

Listing 4: Computing the squares

- The full algorithm uses the above functions and finally computes $gcd(X - Y, N)$, which will hopefully be a non-trivial factor of N.

```python
def random_squares_factoring(N: int) -> int:
    B = min(10000, N)
    factor_base = list(primerange(0, B))
    xs, exponent_matrix = random_squares_stage_1(N, factor_base)
    lambdas, fs = non_trivial_lin_dep(exponent_matrix)
    X, Y = compute_squares(N, xs, lambdas, fs, factor_base)
    return np.gcd(X-Y, N)
```

Listing 5: Full algorithm

Task 2a, Quadratic Sieve

The quadratic sieve is a faster way to find relations. This implementation takes in a factor base, N and a sieve size. The sieve is generated from square numbers mod N, close to zero. It then uses the sympy.ntheory.residue_ntheory library to calculate the solutions to $x^2 \equiv N \pmod p$ for each p in the factor base. These solutions are then used as the basis for the sieve process, where the sieve element at each multiple of p are divided by p. And the corresponding entries in the exponent matrix are incremented. Unfortunately, this means that prime powers can't be factored, which severely limits the performance of this implementation.

Finally, the algorithm filters out number that still have remaining factors.

```
def quadratic_sieve(factor_base: [int], N: int, sieve_size: int) -> [tuple[
    int, dict[int, int]]]:
    root = math.isqrt(N) + 1
    sieve = [(root + n)**2 - N for n in range(sieve_size)]
    exponents = [[0 for _ in factor_base] for _ in range(sieve_size)]

    for i, p in enumerate(factor_base):
        for solution in sympy.ntheory.residue_ntheory.sqrt_mod_iter(N, p):
            index = solution - root
            if index >= sieve_size:
                break
            if index < 0:
                index += (-index//p + 1)*p
            while index < sieve_size:
                sieve[index] //= p
                exponents[index][i] += 1
                index += p
    return [(root + n, exponents[n]) for n, residue in enumerate(sieve) if
    residue == 1]
```

Listing 6: Quadratic Sieve

This factoring algorithm uses the quadratic sieve and the functions from task 1 to factor an integer N.

```
def quadratic_sieve_factoring(N: int) -> int:
    B = int(math.log(N))
    factor_base = [prime for prime in sympy.primerange(0, B) if pow(N, (
    prime-1)//2, prime) == 1]
    xs, exponent_matrix = zip(*quadratic_sieve(factor_base, N, 100*len(
    factor_base)))
    exponent_matrix = np.array(exponent_matrix)
    lambdas, fs = non_trivial_lin_dep(exponent_matrix)
    X, Y = compute_squares(N, xs, lambdas, fs, factor_base)
    return np.gcd(X - Y, N)
```

Listing 7: Quadratic Sieve Factoring

Task 4

I tried to redo the factorisations from Brillhart and Selfridge[?], but even the first example was too large for my implementation.

Appendix

Code is available at https://github.com/andrmoe/ComputationalAlgebra

```python
from random import randint
import numpy as np
import galois
from sympy import primerange


def trial_division_factoring(N: int, factor_base: [int]) -> [int]:
    exponents = []
    for prime in factor_base:
        power = 0
        while N % prime == 0:
            N = N//prime
            power += 1
        exponents.append(power)
    if N == 1:
        return exponents
    else:
        return None


def random_squares_stage_1(N: int, factor_base: [int]) -> tuple[np.ndarray, np.ndarray]:
    exponent_matrix = []
    xs = []
    while len(exponent_matrix) < len(factor_base) + 1:
        x = randint(1, N - 1)
        a = x ** 2 % N
        exponents = trial_division_factoring(a, factor_base)
        if exponents is not None:
            exponent_matrix.append(exponents)
            xs.append(x)
    return np.array(xs, dtype=object), np.array(exponent_matrix)


def non_trivial_lin_dep(exponent_matrix: np.ndarray) -> tuple[np.ndarray, np.ndarray]:
    GF = galois.GF(2)
    exp_matrix_mod2 = GF(exponent_matrix % 2)
    null_space = exp_matrix_mod2.T.null_space()
    lambdas = np.array(null_space[0], dtype=int)
    return lambdas, lambdas.dot(exponent_matrix) // 2


def compute_squares(N: int, xs: np.ndarray, lambdas: np.ndarray, fs: np.ndarray, factor_base: [int]) -> tuple[int, int]:
    X = np.prod([pow(int(x), int(l), N) for x, l in zip(xs, lambdas)], dtype=object) % N
    Y = np.prod([pow(p, int(f), N) for p, f in zip(factor_base, fs)], dtype=object) % N
    return X, Y


def random_squares_factoring(N: int) -> int:
    B = min(10000, N)
```

```python
    factor_base = list(primerange(0, B))
    xs, exponent_matrix = random_squares_stage_1(N, factor_base)
    lambdas, fs = non_trivial_lin_dep(exponent_matrix)
    X, Y = compute_squares(N, xs, lambdas, fs, factor_base)
    return np.gcd(X-Y, N)
```

Listing 8: random_squares.py

```python
import math

import numpy as np
import sympy
from random_squares import non_trivial_lin_dep, compute_squares


def eulers_criterion(a: int, p: int) -> bool:
    return pow(a, (p-1)//2, p) == 1


def generate_factor_base(smoothness_bound: int, N: int) -> [int]:
    return [prime for prime in sympy.primerange(0, smoothness_bound) if pow(N, (prime-1)//2, prime) == 1]


def quadratic_sieve(factor_base: [int], N: int, sieve_size: int) -> [tuple[int, dict[int, int]]]:
    root = math.isqrt(N) + 1
    sieve = [(root + n)**2 - N for n in range(sieve_size)]
    exponents = [[0 for _ in factor_base] for _ in range(sieve_size)]

    for i, p in enumerate(factor_base):
        for solution in sympy.ntheory.residue_ntheory.sqrt_mod_iter(N, p):
            index = solution - root
            if index >= sieve_size:
                break
            if index < 0:
                index += (-index//p + 1)*p
            while index < sieve_size:
                sieve[index] //= p
                exponents[index][i] += 1
                index += p
    return [(root + n, exponents[n]) for n, residue in enumerate(sieve) if residue == 1]


def quadratic_sieve_factoring(N: int) -> int:
    B = int(math.log(N))
    factor_base = [prime for prime in sympy.primerange(0, B) if pow(N, (prime-1)//2, prime) == 1]
    xs, exponent_matrix = zip(*quadratic_sieve(factor_base, N, 100*len(factor_base)))
    exponent_matrix = np.array(exponent_matrix)
    lambdas, fs = non_trivial_lin_dep(exponent_matrix)
    X, Y = compute_squares(N, xs, lambdas, fs, factor_base)
    return np.gcd(X - Y, N)
```

Listing 9: quadratic_sieve.py

```python

```

```
# This was extracted from John Brillhart and J. L. Selfridge. Some
    factorizations of 2n  1  and related results. Mathematics of Computation
    , 21(97):87  96  , 1967.
# using Claude 3.7 Sonnet (Reasoning). There might be errors
factorizations = {
    "2^103 + 1": "3 : 4151416301938142767081771726171 1",
    "2^109 - 1": "5 : 74323515777853174651885214034553",
    "2^119 + 1": "3  4343691   : 8236796831431625531655609592 9",
    "2^121 + 1": "3 683  : 11737111054184582797800455736061107 7",
    "2^124 + 1": "17 : 290657377020264111416291804019768958773",
    "2^125 - 1": "31  6011801   : 269089806001471088316887950601",
    "2^125 + 1": "3   112514051    : 229668251551948541833628830325",
    "2^127 + 1": "3 : 56713727820156410577229101238628035243 3",
    "2^131 + 1": "3 : 1049474429718233112868120778178439181361 1",
    "2^136 + 1": "257 383521  : 2368179743873373200722470799764577",
    "2^137 + 1": "3 : 109715619321279636243568110549821202759297 7",
    "2^139 - 1": ": 5625767248687123876132205208335762278423601",
    "2^139 + 1": "3 : 45069375154263952466179530007417425036569",
    "2^140 + 1": "17   6168115790321    : 8417984207765786201186788968 1",
    "2^143 - 1": "23  898191   : 7241531588229514315782172113400990 73",
    "2^143 + 1": "3  6832731   : 20036156182033104252854431550048777533 23",
    "2^145 - 1": "31   23311032089    : 2679895157783862814690027494144991"
    ,
    "2^145 + 1": "3   11593033169    : 7553921999802854724715300883845411",
    "2^149 + 1": "3 : 119365083338369587798455957350425985635912465 7",
    "2^155 + 1": "3  1171582883   : 1116159476032213978912912605604316852 1"
    ,
    "2^157 - 1": ": 852133201607264441671654058017289213438736861041 77",
    "2^160 + 1": "641 6700417  : 36025619445568495348456305599183855808 11",
    "2^161 - 1": "47  127178481   :
    12893188767450760445531480860771531578248 11",
    "2^161 + 1": "3  432796203   :
    8103467492759792327149800361564410265219 9",
    "2^167 - 1": ": 2349023 prime ",
    "2^167 + 1": "3 : prime",
    "2^175 + 1": "3      1143251281405186171      :
    10511102513478332784513401003231525251 1",
    "2^183 - 1": "13   3456749667055378149    :
    5080081420950858994191255565199180 81",
    "2^185 + 1": "3   11177725781083    :
    1481281366517784293653978876085406183308732811",
    "2^191 + 1": "3 : prime",
    "2^197 - 1": ": 7487 prime ",
    "2^199 + 1": "3 : prime",
    "2^205 + 1": "3   11838831418697    : prime",
    "2^217 - 1": "113  5581384773   : prime",
    "2^220 + 1": "17    35361681293154241 7    :
    10912114872134046760011103546570808125467173176816 8",
    "2^233 - 1": ": 1399135607622577 prime ",
    "2^239 - 1": ": 479     1913573717638313400060 9prime      ",
    "2^241 - 1": ": 22000409 prime ",
    "2^255 + 1": "3      1130733128576529436 91      :
    12241418562986357561513661494554947539 31",
    "2^272 - 1": "97 673  : prime",
    "2^313 + 1": "3 : prime",
    "2^356 + 1": "17 : prime"
```

```
}
```

Listing 10: examples_brillhart_and_selfridge.py

```python
from random_squares import random_squares_factoring
from quadratic_sieve import quadratic_sieve_factoring
from examples_brillhart_and_selfridge import factorizations


def experiment():
    for expression in factorizations.keys():
        expression = expression.replace('^', '**')
        N = eval(expression)
        print(expression, '=', N)
        factor = quadratic_sieve_factoring(N)
        print(factor, N % factor)



if __name__ == '__main__':
    experiment()
```

Listing 11: experiment.py

```python
from random_squares import trial_division_factoring
from quadratic_sieve import quadratic_sieve, eulers_criterion,
    generate_factor_base

import numpy as np

def test_trial_division_factoring():
    factor_base = [2, 3, 5, 7, 11, 13]
    assert trial_division_factoring(5, factor_base) == [0, 0, 1, 0, 0, 0]
    assert trial_division_factoring(2*3**3*7, factor_base) == [1, 3, 0, 1,
    0, 0]
    assert trial_division_factoring(7**100, factor_base) == [0, 0, 0, 100,
    0, 0]
    assert trial_division_factoring(17, factor_base) is None


def test_quadratic_sieve_factor_base():
    N = 5657*7757
    factor_base = generate_factor_base(100, N)
    for p in factor_base:
        assert eulers_criterion(N, p)


def test_quadratic_sieve():
    N = 5657 * 7757
    factor_base = generate_factor_base(1000, N)
    result = quadratic_sieve(factor_base, N, 10*len(factor_base))
    for x, factorisation in result:
        assert x**2 - N == np.prod([p**r for p, r in zip(factor_base,
    factorisation)], dtype=object)
```

Listing 12: test.py

References

[1] John Brillhart, J. L. Selfridge, "Some factorizations of 2n ±1 and related results," *Mathematics of Computation*, vol. 21, no. 97, pp. 87–96, 1967.