Quadratic Sieve Running on Java

Intro

My name is **Ilya Gazman**, I am a dev.
Over the course of the next episodes, I will implement an **efficient** version of the Quadratic Sieve in **Java**.

To follow up you need dev skills, basic Java knowledge, and some basic Math.

If you have any questions, please comment below, or post in the Facebook group.
https://www.facebook.com/groups/Quadratic.Sieve/

Factor the number $N = A \cdot B$ where A and B are big prime numbers.

Episode 1: Fermat's factorization method

The idea is to write N as a difference of squares

$$(C + D)(C - D) = C^2 - D^2 = N$$

Then we can search for square numbers who are square minus N is also a square
$C^2 - N = D^2$

In the worse case, it is a slower then trivial division (testing all the numbers up to $\sqrt{N}$). But as we see later it has tons of optimizations that eventually turners it into the **Quadratic Sieve** algorithm!

Episode 2: Congruence of squares

Recap

In the last episode, we used Fermat's factorization method to factor numbers having factors of 22 bits in seconds. We picked random numbers to satisfy the formula below

$C^2 - N = D^2$

We can make a much weaker condition to look at the Congruence of squares

$C^2 \equiv D^2 \ (mod\ N) \Rightarrow C^2 - D^2 = 0 \ (mod\ N) \Rightarrow (C + D)(C - D) = 0 \ (mod\ N)$

It means that $C + D$ is a multiplication of N or it’s factors. We can efficiently extract it using the Greatest Command Devisor function, gcd. It works in O(log(N)) time, so it’s super fast. And it’s also part of Java native APIs, so it’s a huge plus.
gcd(N , $C + D$) will either give us the factor of N or N or 1, in which case we will try another option.

If you got any questions, please comment below, or post in the Facebook group.
https://www.facebook.com/groups/Quadratic.Sieve/

You can find the source code for those episodes on Github
https://github.com/gazman-sdk/quadratic-sieve-Java

Episode 3: Stress tool

Recap

So far we spoke about three optimizations for the Fermat's factorization method

- Looking for small mods at $\lceil \sqrt{N} \rceil^2$
- Using congress of squares $C^2 \equiv D^2 \ (mod\ N)$
- And multiplying N by a small integer k

We created a small program for each one of those:

- Episode 1
- Episode 2
- Episode 2.1

Now let’s build a stress a tool, to compare the performance of those algorithms. And also add a trivial division as another comparison.

If you got any questions, please comment below, or post in the Facebook group.
https://www.facebook.com/groups/Quadratic.Sieve/

You can find the source code for those episodes on Github
https://github.com/gazman-sdk/quadratic-sieve-Java

Episode 4: Dixon Method - Part 1

Recap:

In the last episode, we built a stress tool for testing the performances, of our algorithms so far. A big lesson that I learned is to not record and run a stress tool at the same time lol.

Here are the true results running offline, speed of factoring 30-bit numbers over 1 sec

- Episode 2.1	speed 1631.198103012559
- Trial Division	speed 680.556549352442
- Episode 2	speed 451.9630294241931
- Episode 1	speed 87.29335158954746

Progress:

- With Fermat's method, we have been looking for a difference of squares using random numbers $C^2 - N = D^2$
- Then we switched looking into sequential numbers from starting from $C = \left|\sqrt{N}\right|$
- Then we found a weeker condition of Congruence of squares, looking into $C^2 \equiv D^2 \ (mod\, N)$
- We added a small multiplier k for $C = \left|\sqrt{kN}\right|$

Now with the Dixon method, we are going to look into the even weaker condition of $C^2 \equiv E \ \ (mod\, N)$

Where E is a product of small factors bounded by some constant B, we are colling for each such combination of $C^2 \equiv E \ \ (mod\, N)$ a “relation” or a B-Smooth value. Once we found $B + 1$ such relations, we can find a combination that will form a square.
Ex:

$$C_1{}^2 \equiv 2^3 11^1 23^8 \ (mod\, N)$$
$$C_2{}^2 \equiv 2^5 11^3 41^6 \ (mod\, N)$$
$$C^2 \equiv 2^8 11^4 23^8 41^6 \ \equiv (2^4 11^2 23^4 41^3)^2 \ (mod\, N)$$

We can also represent the relations as vectors R over the base 2

$$R_1 = 1, 1, 0, 0$$
$$R_2 = 1, 1, 0, 0$$
$$xor(R_1, R_2) = 0, 0, 0, 0$$

In fact, we are going to have multiple such relations, and we can use Gaussian Elimination over base 2, to find the null space, the 0 vectors. Every 0 vector is going to be a solution, and just like with the congruence of squares, it can be a trivial solution of N or 1, or it can be a real factor of N.

Gaussian Elimination is an expensive algorithm that runs in $O(N^3)$, but as we move father you will see that the bottleneck is going to be in finding B-smooth numbers,

If you got any questions, please comment below, or post in the Facebook group.
https://www.facebook.com/groups/Quadratic.Sieve/

You can find the source code for those episodes on Github
https://github.com/gazman-sdk/quadratic-sieve-Java

Episode 5: Dixon Method - Part 2

Recap:

In the last episode, we started to implement the Dixon method. We built a prime base, declared a bound B, and found B+1 relations.

Now we are going to build a matrix to extract a solution out of them. We are going to use an optimized version of Gaussian Elimination for that.

- Since we are working in a base two space, we are going to use XOR instead of multiplication
- Also since we are working in base two, we going to work with bits instead of numbers. It both saves space and speeds up the multiplications.
- Cleanup - We will remove relations who only have has one prime, that can't be eliminated.

The algorithm

- Filter out rows who only has one prime
- Build an identity solution matrix with the size of (Rows count) X (Rows count), so each time we are xoring/eliminating rows, we will also make the xor operation on the solution matrix. It will allow us to extract the solution once we find the null space, empty or zero rows.

- Iterate over `i` from 0 to `B`
    - Iterate over all the rows
        - Find a row with an `i` prime and use as an eliminator, to eliminate/xor all the other `i` primes and exclude it from future eliminations(it cannot be used as an eliminator for other primes, but it can and should be eliminated with other eliminators)

Example

0> 2^8, 3^3, 5^1, 7^2, 11^2, 13^5	->	0,1,1,0,0,1
1> 2^1, 3^2, 5^2, 7^1, 11^0, 13^3	->	1,0,0,1,0,1
2> 2^3, 3^0, 5^4, 7^0, 11^3, 13^2	->	1,0,0,0,1,0 filter out, it only has one prime
3> 2^7, 3^7, 5^3, 7^2, 11^8, 13^4	->	1,1,1,0,0,0
4> 2^0, 3^0, 5^5, 7^0, 11^0 13^7	->	0,0,1,0,0,1
5> 2^0, 3^4, 5^2, 7^3, 11^0, 13^8	->	0,0,0,1,0,0

```
6> 2^0, 3^5, 5^4, 7^2, 11^2, 13^1     ->        0,1,0,0,0,1
7> 2^9, 3^2, 5^0, 7^1, 11^4, 13^0     ->        1,0,0,1,0,0


  0> 0,1,1,0,0,1           0,1,1,0,0,1     |           1,0,0,0,0,0,0      1,0,0,0,0,0,0
  1> 1,0,0,1,0,1  -        1,0,0,1,0,1     |           0,1,0,0,0,0,0      0,1,0,0,0,0,0
  3> 1,1,1,0,0,0  +        0,1,1,1,0,1     |           0,0,1,0,0,0,0      0,1,1,0,0,0,0
  4> 0,0,1,0,0,1  ->       0,0,1,0,0,1     |           0,0,0,1,0,0,0  ->  0,0,0,1,0,0,0
  5> 0,0,0,1,0,0           0,0,0,1,0,0     |           0,0,0,0,1,0,0      0,0,0,0,1,0,0
  6> 0,1,0,0,0,1           0,1,0,0,0,1     |           0,0,0,0,0,1,0      0,0,0,0,0,1,0
  7> 1,0,0,1,0,0  +        0,0,0,0,0,1     |           0,0,0,0,0,0,1      0,1,0,0,0,0,1

  0> 0,1,1,0,0,1  -        0,1,1,0,0,1     |           1,0,0,0,0,0,0      1,0,0,0,0,0,0
* 1> 1,0,0,1,0,1           1,0,0,1,0,1     |           0,1,0,0,0,0,0      0,1,0,0,0,0,0
  3> 0,1,1,1,0,1  +        0,0,0,1,0,0     |           0,1,1,0,0,0,0      1,1,1,0,0,0,0
  4> 0,0,1,0,0,1  ->       0,0,1,0,0,1     |           0,0,0,1,0,0,0  ->  0,0,0,1,0,0,0
  5> 0,0,0,1,0,0           0,0,0,1,0,0     |           0,0,0,0,1,0,0      0,0,0,0,1,0,0
  6> 0,1,0,0,0,1  +        0,0,1,0,0,0     |           0,0,0,0,0,1,0      1,0,0,0,0,1,0
  7> 0,0,0,0,0,1           0,0,0,0,0,1     |           0,1,0,0,0,0,1      0,1,0,0,0,0,1

* 0> 0,1,1,0,0,1  +        0,1,0,0,0,0     |           1,0,0,0,0,0,0      1,0,0,1,0,0,0
* 1> 1,0,0,1,0,1           1,0,0,1,0,1     |           0,1,0,0,0,0,0      0,1,0,0,0,0,0
  3> 0,0,0,1,0,0  ->       0,0,0,1,0,0     |           1,1,1,0,0,0,0      1,1,1,0,0,0,0
  4> 0,0,1,0,0,1  -        0,0,1,0,0,1     |           0,0,0,1,0,0,0  ->  0,0,0,1,0,0,0
  5> 0,0,0,1,0,0           0,0,0,1,0,0     |           0,0,0,0,1,0,0      0,0,0,0,1,0,0
  6> 0,0,1,0,0,0  +        0,0,0,0,0,1     |           1,0,0,0,0,1,0      1,0,0,1,0,1,0
  7> 0,0,0,0,0,1           0,0,0,0,0,1     |           0,1,0,0,0,0,1      0,1,0,0,0,0,1

* 0> 0,1,0,0,0,0           0,1,0,0,0,0     |           1,0,0,1,0,0,0      1,0,0,1,0,0,0
* 1> 1,0,0,1,0,1  +        1,0,0,0,0,1     |           0,1,0,0,0,0,0      1,0,1,0,0,0,0
  3> 0,0,0,1,0,0  -        0,0,0,1,0,0     |           1,1,1,0,0,0,0      1,1,1,0,0,0,0
* 4> 0,0,1,0,0,1  ->       0,0,1,0,0,1     |           0,0,0,1,0,0,0  ->  0,0,0,1,0,0,0
  5> 0,0,0,1,0,0  +        0,0,0,0,0,0     |           0,0,0,0,1,0,0      1,1,1,0,1,0,0
  6> 0,0,0,0,0,1           0,0,0,0,0,1     |           1,0,0,1,0,1,0      1,0,0,1,0,1,0
  7> 0,0,0,0,0,1           0,0,0,0,0,1     |           0,1,0,0,0,0,1      0,1,0,0,0,0,1

* 0> 0,1,0,0,0,0           0,1,0,0,0,0     |           1,0,0,1,0,0,0      1,0,0,1,0,0,0
* 1> 1,0,0,0,0,1 +         1,0,0,0,0,0     |           1,0,1,0,0,0,0      0,0,1,1,0,1,0
* 3> 0,0,0,1,0,0           0,0,0,1,0,0     |           1,1,1,0,0,0,0      1,1,1,0,0,0,0
* 4> 0,0,1,0,0,1 + ->      0,0,1,0,0,0     |           0,0,0,1,0,0,0  ->  1,0,0,0,0,1,0
  5> 0,0,0,0,0,0           0,0,0,0,0,0     |           1,1,1,0,1,0,0      1,1,1,0,1,0,0    solution 0,1,2,4
  6> 0,0,0,0,0,1  -        0,0,0,0,0,1     |           1,0,0,1,0,1,0      1,0,0,1,0,1,0
  7> 0,0,0,0,0,1  +        0,0,0,0,0,0     |           0,1,0,0,0,0,1      1,1,0,1,0,1,1    solution 0,1,3,5,6
```

0> 2^8, 3^3, 5^1, 7^2, 11^2, 13^5
1> 2^1, 3^2, 5^2, 7^1, 11^0, 13^3
2> 2^3, 3^0, 5^4, 7^0, 11^3, 13^2
3> 2^7, 3^7, 5^3, 7^2, 11^8, 13^4
4> 2^0, 3^0, 5^5, 7^0, 11^0 13^7
5> 2^0, 3^4, 5^2, 7^3, 11^0, 13^8
6> 2^0, 3^5, 5^4, 7^2, 11^2, 13^1
7> 2^9, 3^2, 5^0, 7^1, 11^4, 13^0

Solution 0,1,2,4:
0> 2^8, 3^3, 5^1, 7^2, 11^2, 13^5
1> 2^1, 3^2, 5^2, 7^1, 11^0, 13^3
3> 2^7, 3^7, 5^3, 7^2, 11^8, 13^4
5> 2^0, 3^4, 5^2, 7^3, 11^0, 13^8

2^16, 3^16, 5^8, 7^8, 11^10, 13^20 = sqrt(2^8, 3^8, 5^4, 7^4, 11^5, 13^10)^2

Solution 0,1,3,5,6:
0> 2^8, 3^3, 5^1, 7^2, 11^2, 13^5
1> 2^1, 3^2, 5^2, 7^1, 11^0, 13^3
4> 2^0, 3^0, 5^5, 7^0, 11^0 13^7
6> 2^0, 3^5, 5^4, 7^2, 11^2, 13^1
7> 2^9, 3^2, 5^0, 7^1, 11^4, 13^0

2^18, 3^12, 5^12, 7^6, 11^8, 13^16

Episode 6: Hello Quadratic Sieve

During the last two episodes, we have been implementing the Dixon method.
But as it turns out it's not that fast at all.

Now we will see how two additional optimizations will turn it into a blazing fast Quadratic Sieve algorithm!

Also, this is not the last episode and more optimizations are on the way, but today we will start feeling this speed.

Sieving

Example: $292564681 = 7669 \cdot 38149$

$(\left|\sqrt{292564681}\right| + i)^2$ mod N mod $5, 7, 13, 17$

Logging approximation

$a \cdot b = log(a) + log(b)$

To find if a number is bSmooth we can test if the sum of logs is bigger than some limit, ex: $log(\left|\sqrt{N}\right|)$

Sieving process

Pick a range and starting from $\left|\sqrt{N}\right|$ iterate over "range" values at a time.
Build wheels that will help tracking the next divisor of each prime, and compute the logs over the range using all the wheels.

Than each log who is above $log(\left|\sqrt{N}\right|)$, must be a bSmooth value

If you have any questions, please comment below, or post in the Facebook group.
https://www.facebook.com/groups/Quadratic.Sieve/
You can find the source code for those episodes on Github
https://github.com/gazman-sdk/quadratic-sieve-Java

Episode 7: Sieve Of Eratosthenes

In the last episode we sow the true power of quadratic sieve.

During this episode, we will do some cleanup and make a minor optimization of Sieve Of Eratosthenes.

If you have any questions, please comment below, or post in the Facebook group.
https://www.facebook.com/groups/Quadratic.Sieve/
You can find the source code for those episodes on Github
https://github.com/gazman-sdk/quadratic-sieve-Java

Episode 8: Continued Factorization

In the last episode we did some cleanup, improved the prime base construction time, and increased the allowed reminder size that we are willing to take into our BSmooth values collections.

We now have several magic numbers in place that we need to optimize to get the best performances.

- B - The B-smooth bound that determines the largest prime size in our prime base.
- Range - The size of our logs array and the size of our sieving cycle.
- Reminder Size - The largest reminder size that we allow, to include a number in our B-Smooth values. We didn't give it a name yet so 2 for now, it's a multiplier of the biggest prime value in our prime base.
- 1.5 - The multiplier of the prime base size that determines the number of the values we are willing to collect before moving into extracting the vectors and solving the matrix.

During this episode, I try to get rid of the last magic number, the prime base multiplier. Last time we factored the below number:
4374856124133371935830349970701053157453 09

This how it looks while I am not recording:

```
17> A 55233095102462979 2017
24> B 79207151364912128 0077
24> N 4374856124133371935830349970701053157453 09
76> built prime base of 3774/80000. Max prime 79999
2181> found 5662 bSmooth values
3303> Extracted 4573/5662 vectors
4304> Found 545
4497> 0 bad luck 1
4672> 1 bad luck 1
4855> 2 bad luck 4374856124133371935830349970701053157453 09
5039> 3 Oh yeah 55233095102462979 2017
```

It takes 5.039 seconds.

After playing that magic number, I found that the optimal value for this number is 4860(the total BSmooth values we wish to collect). It looks like this:

```
20> A 552330951024629792017
28> B 792071513649121280077
28> N 437485612413337193583034997070105315745309
89> built prime base of 3774/80000. Max prime 79999
1808> found 4860 bSmooth values
2708> Extracted 3926/4860 vectors
3457> Found 2
3634> 0 bad luck 437485612413337193583034997070105315745309
3815> 1 Oh yeah 792071513649121280077
```

Continued Factorization

This is my original optimization idea. It doesn’t mean that I was the first to discover it, a research is needed to answer that question, but it means that I am not aware of anyplace else doing anything similar.

Instead of calculating the matrix at the end, let’s calculate it for each new prime

The algorithm

```
Continues input: bSmooth value
Output: A factor of N

    - Save bSmooth into bSmoothList
    - Add new row for the solutions matrix
    - For each bitIndex in bSmooth.vector
        - Check in eliminators map, if we have an eliminator that can eliminate that bit
            - eliminate/xor it if we do
    - Now the bSmooth is either a solution or a new eliminator

Input: eliminator
    - if eliminator vector is empty/NULL
        - Then extract a solution from the solutions matrix of the eliminator row.
    - Otherwise find the index of the first 1 bit in the eliminator.vector
    - Add the eliminator to the eliminators map by that index
    - For each bSmooth in bSmoothList
        - Check if it has the eliminator index on
```

        - Eliminate it if it does
- Recursively execute this method for each bSmooth that were eliminated

If you have any questions, please comment below, or post in the Facebook group.
https://www.facebook.com/groups/Quadratic.Sieve/
You can find the source code for those episodes on Github
https://github.com/gazman-sdk/quadratic-sieve-Java

Episode 9: Improving logs

In the last episode we improved our matrix to work in real time. Now we have a multithreaded system that factors our numbers.

Today we are going to update the logging, to get a better picture about how it works and fix some small bugs.

Updated log format

If you have any questions, please comment below, or post in the Facebook group.
https://www.facebook.com/groups/Quadratic.Sieve/
You can find the source code for those episodes on Github
https://github.com/gazman-sdk/quadratic-sieve-Java

Episode 10: Multi Polynomial

In the last episode we worked on getting better logs. Now let’s get back for optimizations!

So far we been working on the below polinomial, searching for bSmooth values of y

$$(x + \left\lceil \sqrt{N} \right\rceil)^2 - N = y$$

The problem with this approach is that y will keep growing forever and so the chances for it to be smooth will decrease, and so is our sieving speed.

Peter Montgomery, offered another polynomials set of the form

$$(ax + b)^2 - N = y$$

If we choose $b^2 - N = ac$, then we can rewrite the entire thing as

$$a^2x^2 + 2ab \ + \ b^2 - N = a^2x^2 + 2ab \ + \ ac = a(a\,x^2 + 2b \ + \ c) = y$$

If we also say that a is a square, then we can ignore it while sieving and just consider $a\,x^2 + 2b \ + \ c$

To find a solution for $b^2 - N = ac$ we can pick a prime q near to $\sqrt[4]{N}$, then define a as $a = q^2$. Also $b^2 - N = ac$ will only have a solution if N is a quadratic residue of q .

To calculate b we need to compute the modulare square root of N mod a . We do that by first calculating the modulare square root of N mod q and then “lifting” the root mod q^2 using Hensel’s Lemma.

The lifting is calculated as explained by Carl Schildkraut.
https://math.stackexchange.com/a/3779351/101178

The idea of a multi polynomial is to periodically pick a new q and sive over a new polynomial. It also makes it easy to sieve in parallel.

In fact q needs to be picked up near $\sqrt{\frac{\sqrt{2N}}{m}}$ where m is the sieving range, each time after sieving m values, we switch a polynomial. This way it is promised that y will always be around $m\sqrt{2n}$ for $-m < x < m$

The only problem with this optimization is that it requires rebuilding the wheels each time we change a polynomial, and that is expensive.

If you have any questions, please comment below, or post in the Facebook group.
https://www.facebook.com/groups/Quadratic.Sieve/
You can find the source code for those episodes on Github
https://github.com/gazman-sdk/quadratic-sieve-Java