

Peut-on factoriser suffisamment rapidement les
nombres en facteurs premiers?

Tristan Delcourt (15699), Louise Nguyen (28438)

2025

Plan

Introduction et enjeux

La méthode de Dixon

Congruences de carrés

Etapes de la méthode

L'algorithme final

Optimisations

Crible quadratique

Approximation logarithmique

Résultats

Annexe

Les nombres *RSA*

- ▶ Factoriser $N = pq$ où p et q sont premiers et très grands.
- ▶ Dernier nombre non factorisé: RSA-260 (260 chiffres)

$N = 221128255295296664352810852550262309276120895$
0247001539441374831912882294140200198651272972656
9746599085900330031400051170742204560859276357953
7571859542988389587092292384910067030341246205457
8456641366454068421436129301769402084639106587591
4794251435144458199

Plan

Introduction et enjeux

La méthode de Dixon

 Congruences de carrés

 Etapas de la méthode

 L'algorithme final

Optimisations

 Crible quadratique

 Approximation logarithmique

Résultats

Annexe

Plan

Introduction et enjeux

La méthode de Dixon

Congruences de carrés

Etapas de la méthode

L'algorithme final

Optimisations

Crible quadratique

Approximation logarithmique

Résultats

Annexe

Congruence de carrés

$N = pq$, p premier. Supp. $x^2 \equiv y^2 \pmod{N}$ et $x \not\equiv \pm y$.

- ▶ On a $x^2 - y^2 \equiv 0 \pmod{N}$ i.e. $N \mid (x - y)(x + y)$
- ▶ Donc $p \mid (x - y)(x + y)$
- ▶ Lemme d'Euclide: par exemple $p \mid x - y$
- ▶ Alors p divise N et $x - y$: $p \mid N \wedge (x - y)$, ce qui donne $N \wedge (x - y) \neq 1$

Conclusion

$N \wedge (x - y)$ et $N \wedge (x + y)$ sont des facteurs non-triviaux de N

Congruence de carrés

$N = pq$, p premier. Supp. $x^2 \equiv y^2 \pmod{N}$ et $x \not\equiv \pm y$.

- ▶ On a $x^2 - y^2 \equiv 0 \pmod{N}$ i.e. $N \mid (x - y)(x + y)$
- ▶ Donc $p \mid (x - y)(x + y)$
- ▶ Lemme d'Euclide: par exemple $p \mid x - y$
- ▶ Alors p divise N et $x - y$: $p \mid N \wedge (x - y)$, ce qui donne $N \wedge (x - y) \neq 1$

Conclusion

$N \wedge (x - y)$ et $N \wedge (x + y)$ sont des facteurs non-triviaux de N

Congruence de carrés

$N = pq$, p premier. Supp. $x^2 \equiv y^2 \pmod{N}$ et $x \not\equiv \pm y$.

- ▶ On a $x^2 - y^2 \equiv 0 \pmod{N}$ i.e. $N \mid (x - y)(x + y)$
- ▶ Donc $p \mid (x - y)(x + y)$
- ▶ Lemme d'Euclide: par exemple $p \mid x - y$
- ▶ Alors p divise N et $x - y$: $p \mid N \wedge (x - y)$, ce qui donne $N \wedge (x - y) \neq 1$

Conclusion

$N \wedge (x - y)$ et $N \wedge (x + y)$ sont des facteurs non-triviaux de N

Congruence de carrés

$N = pq$, p premier. Supp. $x^2 \equiv y^2 \pmod{N}$ et $x \not\equiv \pm y$.

- ▶ On a $x^2 - y^2 \equiv 0 \pmod{N}$ i.e. $N \mid (x - y)(x + y)$
- ▶ Donc $p \mid (x - y)(x + y)$
- ▶ Lemme d'Euclide: par exemple $p \mid x - y$
- ▶ Alors p divise N et $x - y$: $p \mid N \wedge (x - y)$, ce qui donne $N \wedge (x - y) \neq 1$

Conclusion

$N \wedge (x - y)$ et $N \wedge (x + y)$ sont des facteurs non-triviaux de N

Congruence de carrés

$N = pq$, p premier. Supp. $x^2 \equiv y^2 \pmod{N}$ et $x \not\equiv \pm y$.

- ▶ On a $x^2 - y^2 \equiv 0 \pmod{N}$ i.e. $N \mid (x - y)(x + y)$
- ▶ Donc $p \mid (x - y)(x + y)$
- ▶ Lemme d'Euclide: par exemple $p \mid x - y$
- ▶ Alors p divise N et $x - y$: $p \mid N \wedge (x - y)$, ce qui donne **$N \wedge (x - y) \neq 1$**

Conclusion

$N \wedge (x - y)$ et $N \wedge (x + y)$ sont des facteurs non-triviaux de N

Congruence de carrés

$N = pq$, p premier. Supp. $x^2 \equiv y^2 \pmod{N}$ et $x \neq \pm y$.

- ▶ On a $x^2 - y^2 \equiv 0 \pmod{N}$ i.e. $N \mid (x - y)(x + y)$
- ▶ Donc $p \mid (x - y)(x + y)$
- ▶ Lemme d'Euclide: par exemple $p \mid x - y$
- ▶ Alors p divise N et $x - y$: $p \mid N \wedge (x - y)$, ce qui donne $N \wedge (x - y) \neq 1$

Conclusion

$N \wedge (x - y)$ et $N \wedge (x + y)$ sont des facteurs non-triviaux de N

Plan

Introduction et enjeux

La méthode de Dixon

Congruences de carrés

Etapes de la méthode

L'algorithme final

Optimisations

Crible quadratique

Approximation logarithmique

Résultats

Annexe

$$b \in \mathbb{N}$$

2

3

5

•

•

•

p_b

$$b \in \mathbb{N}$$

$$(x_1, \quad x_2, \quad x_3, \quad \dots, \quad x_{b+1})$$

2

3

5

.

.

.

p_b

$$b \in \mathbb{N}$$

$$(x_1, \quad x_2, \quad x_3, \quad \dots, \quad x_{b+1})$$

$$x_1^2 \pmod{N}$$

2

3

5

.

.

.

p_b

$$b \in \mathbb{N}$$

$$(x_1, \quad x_2, \quad x_3, \quad \dots, \quad x_{b+1})$$

$$x_1^2 \pmod{N}$$

$$2 \quad v_1^{(1)}$$

$$3 \quad v_1^{(2)}$$

$$5 \quad v_1^{(3)}$$

$$\cdot \quad \cdot$$

$$\cdot \quad \cdot$$

$$\cdot \quad \cdot$$

$$p_b \quad v_1^{(b)}$$

$$b \in \mathbb{N}$$

$$(x_1, \quad x_2, \quad x_3, \quad \dots, \quad x_{b+1})$$

$$x_1^2 \pmod{N}$$

$$2 \qquad v_1^{(1)}$$

$$3 \qquad v_1^{(2)}$$

$$5 \qquad v_1^{(3)}$$

$$\cdot \qquad \cdot$$

$$\cdot \qquad \cdot$$

$$\cdot \qquad \cdot$$

$$p_b \qquad v_1^{(b)}$$

$$\left. \begin{array}{l} 2 \qquad v_1^{(1)} \\ 3 \qquad v_1^{(2)} \\ 5 \qquad v_1^{(3)} \\ \cdot \qquad \cdot \\ \cdot \qquad \cdot \\ \cdot \qquad \cdot \\ p_b \qquad v_1^{(b)} \end{array} \right\} x_1^2 \pmod{N} = \prod_{i=1}^b p_i^{v_1^{(i)}} = 2^{v_1^{(1)}} 3^{v_1^{(2)}} \dots p_b^{v_1^{(b)}}$$

$$b \in \mathbb{N}$$

$$(x_1, \quad x_2, \quad x_3, \quad \dots, \quad x_{b+1})$$

$$x_1^2 \pmod{N}$$

2

$$v_1^{(1)}$$

3

$$v_1^{(2)}$$

5

$$v_1^{(3)}$$

.

.

.

.

.

.

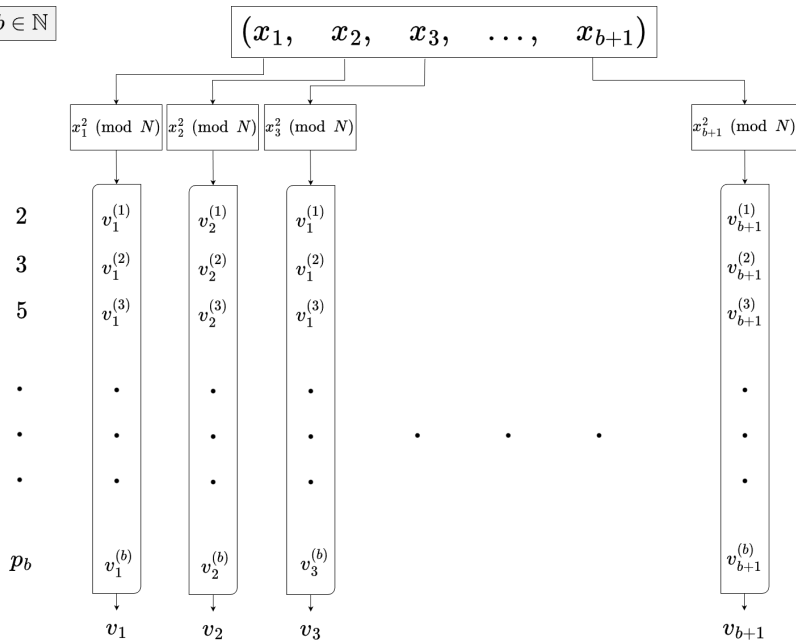
p_b

$$v_1^{(b)}$$

$$v_1$$

$$x_1^2 \pmod{N} = \prod_{i=1}^b p_i^{v_1^{(i)}} = 2^{v_1^{(1)}} 3^{v_1^{(2)}} \dots p_b^{v_1^{(b)}}$$

$$b \in \mathbb{N}$$



Construction de y - Pivot de Gauss

- $b + 1$ vecteurs de \mathbb{F}_2^b , système lié:

$$\exists (\lambda_i)_{i \in \llbracket 1, b+1 \rrbracket} \in \{0, 1\}^{b+1} \mid \sum_{i=1}^{b+1} \lambda_i v_i = 0_{\mathbb{F}_2^b} = (2\alpha_1, \dots, 2\alpha_b)$$

- On pose $y = \prod_{j=1}^b p_j^{\alpha_j}$ et $x = \prod_{j=1}^{b+1} x_j^{\lambda_j}$

Résultat admis (calcul en annexe)

$$x^2 \equiv y^2 \pmod{N}$$

Construction de y - Pivot de Gauss

- ▶ $b + 1$ vecteurs de \mathbb{F}_2^b , système lié:

$$\exists (\lambda_i)_{i \in \llbracket 1, b+1 \rrbracket} \in \{0, 1\}^{b+1} \mid \sum_{i=1}^{b+1} \lambda_i v_i = 0_{\mathbb{F}_2^b} = (2\alpha_1, \dots, 2\alpha_b)$$

- ▶ On pose $y = \prod_{j=1}^b p_j^{\alpha_j}$ et $x = \prod_{j=1}^{b+1} x_j^{\lambda_j}$

Résultat admis (calcul en annexe)

$$x^2 \equiv y^2 \pmod{N}$$

Construction de y - Pivot de Gauss

- $b + 1$ vecteurs de \mathbb{F}_2^b , système lié:

$$\exists (\lambda_i)_{i \in \llbracket 1, b+1 \rrbracket} \in \{0, 1\}^{b+1} \mid \sum_{i=1}^{b+1} \lambda_i v_i = 0_{\mathbb{F}_2^b} = (2\alpha_1, \dots, 2\alpha_b)$$

- On pose $y = \prod_{j=1}^b p_j^{\alpha_j}$ et $x = \prod_{j=1}^{b+1} x_j^{\lambda_j}$

Résultat admis (calcul en annexe)

$$x^2 \equiv y^2 \pmod{N}$$

Résolution du système

```
typedef struct system {  
    int** m; // matrice du systeme  
    int* perm; // permutation de la solution  
    int* sol; // le vecteur lambda  
    bool done; // iteration des solutions fini?  
    int n1, n2, arb;  
} system_s;  
  
typedef system_s* system_t;
```

```
// initialise le systeme et donne une premiere solution dans 'sol'  
    avec un pivot de Gauss
```

```
system_t init_gauss(int** v, int n1, int n2);
```

```
// itere la solution trouvee, la stocke dans 'sol' et met 'done' a  
    true si c'est la derniere
```

```
void gaussien_step(system_t s);
```

```
// libere la memoire du systeme
```

```
void free_system(system_t s);
```


► $N = 20382493 = 3467 \times 5879$ et $b = 4$.

► $x_j^2 \bmod N = 2^{v_j^{(1)}} \cdots 7^{v_j^{(4)}}$ pour $j \in \llbracket 1, 5 \rrbracket$

x_j	v_j
16853	(6, 5, 2, 2)
32877	(3, 0, 7, 0)
35261	(5, 3, 0, 1)
56569	(3, 2, 1, 0)
48834	(0, 2, 3, 1)

► On résout dans \mathbb{F}_2^5

$$\begin{cases} 6\lambda_1 + 3\lambda_2 + 5\lambda_3 + 3\lambda_4 + 0\lambda_5 = 0_{\mathbb{F}_2} \\ 5\lambda_1 + 0\lambda_2 + 3\lambda_3 + 2\lambda_4 + 2\lambda_5 = 0_{\mathbb{F}_2} \\ 2\lambda_1 + 7\lambda_2 + 0\lambda_3 + 1\lambda_4 + 3\lambda_5 = 0_{\mathbb{F}_2} \\ 2\lambda_1 + 0\lambda_2 + 1\lambda_3 + 0\lambda_4 + 1\lambda_5 = 0_{\mathbb{F}_2} \end{cases}$$

$\lambda = (1, 1, 1, 0, 1)$ solution.

► $x = \prod_{j=1}^{b+1} x_j^{\lambda_j} = 7248176$
 $y = \prod_{j=1}^b p_j^{\alpha_j} = 4837786$

► On a $x^2 \equiv y^2 \pmod{N}$

► $N \wedge (x - y) = 5879$ et
 $N \wedge (x + y) = 3467$.

► $N = 20382493 = 3467 \times 5879$ et $b = 4$.

► $x_j^2 \bmod N = 2^{v_j^{(1)}} \cdots 7^{v_j^{(4)}}$ pour $j \in \llbracket 1, 5 \rrbracket$

x_j	v_j
16853	(6, 5, 2, 2)
32877	(3, 0, 7, 0)
35261	(5, 3, 0, 1)
56569	(3, 2, 1, 0)
48834	(0, 2, 3, 1)

► On résout dans \mathbb{F}_2^5

$$\begin{cases} 6\lambda_1 + 3\lambda_2 + 5\lambda_3 + 3\lambda_4 + 0\lambda_5 = 0_{\mathbb{F}_2} \\ 5\lambda_1 + 0\lambda_2 + 3\lambda_3 + 2\lambda_4 + 2\lambda_5 = 0_{\mathbb{F}_2} \\ 2\lambda_1 + 7\lambda_2 + 0\lambda_3 + 1\lambda_4 + 3\lambda_5 = 0_{\mathbb{F}_2} \\ 2\lambda_1 + 0\lambda_2 + 1\lambda_3 + 0\lambda_4 + 1\lambda_5 = 0_{\mathbb{F}_2} \end{cases}$$

$\lambda = (1, 1, 1, 0, 1)$ solution.

► $x = \prod_{j=1}^{b+1} x_j^{\lambda_j} = 7248176$
 $y = \prod_{j=1}^b p_j^{\alpha_j} = 4837786$

► On a $x^2 \equiv y^2 \pmod{N}$

► $N \wedge (x - y) = 5879$ et
 $N \wedge (x + y) = 3467$.

► $N = 20382493 = 3467 \times 5879$ et $b = 4$.

► $x_j^2 \bmod N = 2^{v_j^{(1)}} \cdots 7^{v_j^{(4)}}$ pour $j \in \llbracket 1, 5 \rrbracket$

x_j	v_j
16853	(6, 5, 2, 2)
32877	(3, 0, 7, 0)
35261	(5, 3, 0, 1)
56569	(3, 2, 1, 0)
48834	(0, 2, 3, 1)

► On résout dans \mathbb{F}_2^5

$$\begin{cases} 6\lambda_1 + 3\lambda_2 + 5\lambda_3 + 3\lambda_4 + 0\lambda_5 = 0_{\mathbb{F}_2} \\ 5\lambda_1 + 0\lambda_2 + 3\lambda_3 + 2\lambda_4 + 2\lambda_5 = 0_{\mathbb{F}_2} \\ 2\lambda_1 + 7\lambda_2 + 0\lambda_3 + 1\lambda_4 + 3\lambda_5 = 0_{\mathbb{F}_2} \\ 2\lambda_1 + 0\lambda_2 + 1\lambda_3 + 0\lambda_4 + 1\lambda_5 = 0_{\mathbb{F}_2} \end{cases}$$

$\lambda = (1, 1, 1, 0, 1)$ solution.

► $x = \prod_{j=1}^{b+1} x_j^{\lambda_j} = 7248176$
 $y = \prod_{j=1}^b p_j^{\alpha_j} = 4837786$

► On a $x^2 \equiv y^2 \pmod{N}$

► $N \nmid (x - y) = 5879$ et
 $N \nmid (x + y) = 3467$.

► $N = 20382493 = 3467 \times 5879$ et $b = 4$.

► $x_j^2 \bmod N = 2^{v_j^{(1)}} \cdots 7^{v_j^{(4)}}$ pour $j \in \llbracket 1, 5 \rrbracket$

x_j	v_j
16853	(6, 5, 2, 2)
32877	(3, 0, 7, 0)
35261	(5, 3, 0, 1)
56569	(3, 2, 1, 0)
48834	(0, 2, 3, 1)

► On résout dans \mathbb{F}_2^5

$$\begin{cases} 6\lambda_1 + 3\lambda_2 + 5\lambda_3 + 3\lambda_4 + 0\lambda_5 = 0_{\mathbb{F}_2} \\ 5\lambda_1 + 0\lambda_2 + 3\lambda_3 + 2\lambda_4 + 2\lambda_5 = 0_{\mathbb{F}_2} \\ 2\lambda_1 + 7\lambda_2 + 0\lambda_3 + 1\lambda_4 + 3\lambda_5 = 0_{\mathbb{F}_2} \\ 2\lambda_1 + 0\lambda_2 + 1\lambda_3 + 0\lambda_4 + 1\lambda_5 = 0_{\mathbb{F}_2} \end{cases}$$

$\lambda = (1, 1, 1, 0, 1)$ solution.

► $x = \prod_{j=1}^{b+1} x_j^{\lambda_j} = 7248176$
 $y = \prod_{j=1}^b p_j^{\alpha_j} = 4837786$

► On a $x^2 \equiv y^2 \pmod{N}$

► $N \wedge (x - y) = 5879$ et
 $N \wedge (x + y) = 3467$.

► $N = 20382493 = 3467 \times 5879$ et $b = 4$.

► $x_j^2 \bmod N = 2^{v_j^{(1)}} \cdots 7^{v_j^{(4)}}$ pour $j \in \llbracket 1, 5 \rrbracket$

x_j	v_j
16853	(6, 5, 2, 2)
32877	(3, 0, 7, 0)
35261	(5, 3, 0, 1)
56569	(3, 2, 1, 0)
48834	(0, 2, 3, 1)

► On résout dans \mathbb{F}_2^5

$$\begin{cases} 6\lambda_1 + 3\lambda_2 + 5\lambda_3 + 3\lambda_4 + 0\lambda_5 = 0_{\mathbb{F}_2} \\ 5\lambda_1 + 0\lambda_2 + 3\lambda_3 + 2\lambda_4 + 2\lambda_5 = 0_{\mathbb{F}_2} \\ 2\lambda_1 + 7\lambda_2 + 0\lambda_3 + 1\lambda_4 + 3\lambda_5 = 0_{\mathbb{F}_2} \\ 2\lambda_1 + 0\lambda_2 + 1\lambda_3 + 0\lambda_4 + 1\lambda_5 = 0_{\mathbb{F}_2} \end{cases}$$

$\lambda = (1, 1, 1, 0, 1)$ solution.

► $x = \prod_{j=1}^{b+1} x_j^{\lambda_j} = 7248176$
 $y = \prod_{j=1}^b p_j^{\alpha_j} = 4837786$

► On a $x^2 \equiv y^2 \pmod{N}$

► $N \wedge (x - y) = 5879$ et
 $N \wedge (x + y) = 3467$.

► $N = 20382493 = 3467 \times 5879$ et $b = 4$.

► $x_j^2 \bmod N = 2^{v_j^{(1)}} \cdots 7^{v_j^{(4)}}$ pour $j \in \llbracket 1, 5 \rrbracket$

x_j	v_j
16853	(6, 5, 2, 2)
32877	(3, 0, 7, 0)
35261	(5, 3, 0, 1)
56569	(3, 2, 1, 0)
48834	(0, 2, 3, 1)

► On résout dans \mathbb{F}_2^5

$$\begin{cases} 6\lambda_1 + 3\lambda_2 + 5\lambda_3 + 3\lambda_4 + 0\lambda_5 = 0_{\mathbb{F}_2} \\ 5\lambda_1 + 0\lambda_2 + 3\lambda_3 + 2\lambda_4 + 2\lambda_5 = 0_{\mathbb{F}_2} \\ 2\lambda_1 + 7\lambda_2 + 0\lambda_3 + 1\lambda_4 + 3\lambda_5 = 0_{\mathbb{F}_2} \\ 2\lambda_1 + 0\lambda_2 + 1\lambda_3 + 0\lambda_4 + 1\lambda_5 = 0_{\mathbb{F}_2} \end{cases}$$

$\lambda = (1, 1, 1, 0, 1)$ solution.

► $x = \prod_{j=1}^{b+1} x_j^{\lambda_j} = 7248176$
 $y = \prod_{j=1}^b p_j^{\alpha_j} = 4837786$

► On a $x^2 \equiv y^2 \pmod{N}$

► $N \wedge (x - y) = 5879$ et
 $N \wedge (x + y) = 3467$.

Ce qu'il faut retenir

L'enjeu principal

Étant donné $b \in \mathbb{N}$, trouver $b + 1$ nombres tels que
 $\forall j \in \llbracket 1, b + 1 \rrbracket, x_j^2 \bmod N$ a ses facteurs premiers inférieurs à p_b

Plan

Introduction et enjeux

La méthode de Dixon

- Congruences de carrés

- Etapes de la méthode

- L'algorithme final

Optimisations

- Crible quadratique

- Approximation logarithmique

Résultats

Annexe

L'algorithme final

Algorithme 1 Factorisation par la méthode de Dixon

Entrée: $N \in \mathbb{N}$ composé, $b \in \mathbb{N}$

Sortie: p et q , tels que $p \mid N$ et $q \mid N$

1: $(v_i)_{i \in \llbracket 1, b+1 \rrbracket}, (x_i)_{i \in \llbracket 1, b+1 \rrbracket} \leftarrow RechercheNombres(N, b)$

2: $(\lambda_i)_{i \in \llbracket 1, b+1 \rrbracket} \leftarrow PivoteGauss((v_i)_{i \in \llbracket 1, b+1 \rrbracket})$

3: $x \leftarrow \prod_{j=1}^{b+1} x_j^{\lambda_j}$

4: $y \leftarrow \prod_{j=1}^b p_j^{\alpha_j}$

renvoyer $N \wedge (x - y), N \wedge (x + y)$

Etude théorique (Louise Nguyen)

Une minoration de la densité des B -friables

Soit $B : \mathbb{N}^* \rightarrow \mathbb{N}^*$ une fonction telle que $\ln n = o(B(n))$ et $\ln B(n) = o(\ln n)$. Alors on a, pour $n \rightarrow +\infty$,

$$\Psi(B(n), n) \geq n \exp \left(\left(\frac{\ln n}{\ln B(n)} \ln \ln n \right) (-1 + o(1)) \right)$$

Une complexité sous-exponentielle

$$\exp \left((1 + o(1)) 2\sqrt{2} (\ln n \ln \ln n)^{1/2} \right)$$

lorsque $B = \exp \left(\frac{1}{\sqrt{2}} (\ln n \ln \ln n)^{1/2} \right)$

Plan

Introduction et enjeux

La méthode de Dixon

 Congruences de carrés

 Etapas de la méthode

 L'algorithme final

Optimisations

 Crible quadratique

 Approximation logarithmique

Résultats

Annexe

Plan

Introduction et enjeux

La méthode de Dixon

 Congruences de carrés

 Etapas de la méthode

 L'algorithme final

Optimisations

 Crible quadratique

 Approximation logarithmique

Résultats

Annexe

Principe

- Utilisation d'un polynôme $Q = (\underbrace{\lfloor \sqrt{N} \rfloor + X}_{x_i})^2 - N$
- Résolution de $Q(x) \equiv 0 \pmod{p}$ grâce à Tonelli-Shanks, 2 solutions x_p et \tilde{x}_p dans $\llbracket 1, p \rrbracket$.
- $p \mid Q(x) \implies \forall k \in \mathbb{N}, p \mid Q(x + kp)$ (Démonstration en annexe)
- Cribler sur $T = [Q(1), Q(2), Q(3), \dots, Q(S)]$

Principe

- Utilisation d'un polynôme $Q = (\underbrace{\lfloor \sqrt{N} \rfloor + X}_{x_i})^2 - N$
- Résolution de $Q(x) \equiv 0 \pmod{p}$ grâce à Tonelli-Shanks, 2 solutions x_p et \tilde{x}_p dans $\llbracket 1, p \rrbracket$.
- $p \mid Q(x) \implies \forall k \in \mathbb{N}, p \mid Q(x + kp)$ (Démonstration en annexe)
- Cribler sur $T = [Q(1), Q(2), Q(3), \dots, Q(S)]$

Principe

- Utilisation d'un polynôme $Q = (\underbrace{\lfloor \sqrt{N} \rfloor + X}_{x_i})^2 - N$
- Résolution de $Q(x) \equiv 0 \pmod{p}$ grâce à Tonelli-Shanks, 2 solutions x_p et \tilde{x}_p dans $\llbracket 1, p \rrbracket$.
- $p \mid Q(x) \implies \forall k \in \mathbb{N}, p \mid Q(x + kp)$ (Démonstration en annexe)
- Cribler sur $T = [Q(1), Q(2), Q(3), \dots, Q(S)]$

Principe

- Utilisation d'un polynôme $Q = (\underbrace{\lfloor \sqrt{N} \rfloor + X}_{x_i})^2 - N$
- Résolution de $Q(x) \equiv 0 \pmod{p}$ grâce à Tonelli-Shanks, 2 solutions x_p et \tilde{x}_p dans $\llbracket 1, p \rrbracket$.
- $p \mid Q(x) \implies \forall k \in \mathbb{N}, p \mid Q(x + kp)$ (Démonstration en annexe)
- Cribler sur $T = [Q(1), Q(2), Q(3), \dots, Q(S)]$

$$S = 10$$

$$N = 20382493$$

$$T = [Q(1), Q(2), Q(3), Q(4), Q(5), Q(6), Q(7), Q(8), Q(9), Q(10)]$$

$$S = 10$$

$$N = 20382493$$

$$T = [Q(1), Q(2), Q(3), Q(4), Q(5), Q(6), Q(7), Q(8), Q(9), Q(10)]$$

$$p = 2$$

$$Q(1) \equiv 0 \pmod{2}$$

$$S = 10$$

$$N = 20382493$$

$$T = [Q(1), Q(2), Q(3), Q(4), Q(5), Q(6), Q(7), Q(8), Q(9), Q(10)]$$

$$p = 2$$

$$Q(1) \equiv 0 \pmod{2}$$

$$[2732, \quad 11736, \quad 20796, \quad 29831, \quad 38868, \quad \dots, \quad Q(10)]$$

$$S = 10$$

$$N = 20382493$$

$$T = [Q(1), Q(2), Q(3), Q(4), Q(5), Q(6), Q(7), Q(8), Q(9), Q(10)]$$

$$p = 2$$

$$Q(1) \equiv 0 \pmod{2}$$

A sequence of numbers is shown: $[2732, 11736, 20796, 29831, 38868, \dots, Q(10)]$. Two curved arrows indicate an increment of 2. The first arrow starts above 2732 and points to 20796, with a "+2" label above it. The second arrow starts above 20796 and points to 38868, also with a "+2" label above it.

$$[2732, 11736, 20796, 29831, 38868, \dots, Q(10)]$$

$$S = 10$$

$$N = 20382493$$

$$T = [Q(1), Q(2), Q(3), Q(4), Q(5), Q(6), Q(7), Q(8), Q(9), Q(10)]$$

$$p = 2$$

$$Q(1) \equiv 0 \pmod{2}$$

$$\begin{array}{c}
 \begin{array}{ccccccccc}
 & & +2 & & +2 & & & & \\
 & \swarrow & & \searrow & \swarrow & & \searrow & & \\
 [2732, & 11736, & 20796, & 29831, & 38868, & \dots, & Q(10)] \\
 \div 2^2 & \searrow & & & & & & & \\
 & [683, & 11736, & 5199, & 29831, & 9717, & \dots, & Q(10)]
 \end{array}
 \end{array}$$

$v_1^{(1)} = 2$

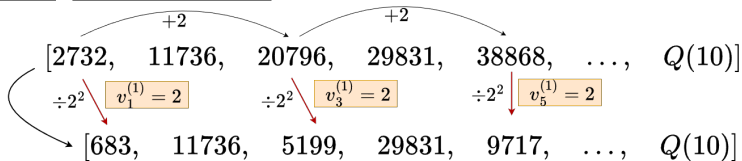
$$S = 10$$

$$N = 20382493$$

$$T = [Q(1), Q(2), Q(3), Q(4), Q(5), Q(6), Q(7), Q(8), Q(9), Q(10)]$$

$$p = 2$$

$$Q(1) \equiv 0 \pmod{2}$$



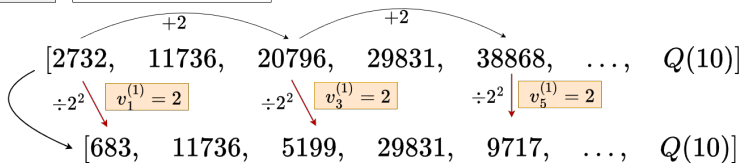
$$S = 10$$

$$N = 20382493$$

$$T = [Q(1), Q(2), Q(3), Q(4), Q(5), Q(6), Q(7), Q(8), Q(9), Q(10)]$$

$$p = 2$$

$$Q(1) \equiv 0 \pmod{2}$$



$$p = 3$$

$$Q(2) \equiv 0 \pmod{3}$$

$$Q(3) \equiv 0 \pmod{3}$$

$$S = 10$$

$$N = 20382493$$

$$T = [Q(1), Q(2), Q(3), Q(4), Q(5), Q(6), Q(7), Q(8), Q(9), Q(10)]$$

$$p = 2$$

$$Q(1) \equiv 0 \pmod{2}$$

$$\begin{array}{ccccccccc}
 & & +2 & & +2 & & & & \\
 & \swarrow & & \searrow & & \swarrow & & \searrow & \\
 [2732, & 11736, & 20796, & 29831, & 38868, & \dots, & Q(10)] \\
 \div 2^2 \swarrow & \boxed{v_1^{(1)} = 2} & \div 2^2 \swarrow & \boxed{v_3^{(1)} = 2} & \div 2^2 \swarrow & \boxed{v_5^{(1)} = 2} & & & \\
 [683, & 11736, & 5199, & 29831, & 9717, & \dots, & Q(10)]
 \end{array}$$

$$p = 3$$

$$Q(2) \equiv 0 \pmod{3}$$

$$Q(3) \equiv 0 \pmod{3}$$

$$\begin{array}{ccccccccc}
 [683, & 11736, & 5199, & 29831, & 9717, & \dots, & Q(10)] \\
 & & & & & & \vdots \\
 & & & & & & \vdots \\
 & & & & & & \vdots
 \end{array}$$

$$p = p_b$$

$$Q(x_{p_b}) \equiv 0 \pmod{p_b}$$

$$Q(\tilde{x}_{p_b}) \equiv 0 \pmod{p_b}$$

Algorithme 2 Algorithme du crible quadratique

Entrée: $N \in \mathbb{N}^*$, $b \in \mathbb{N}^*$, $S \geq 1$

Sortie: $(v_i)_{i \in \llbracket 1, k \rrbracket}$, $(x_i)_{i \in \llbracket 1, k \rrbracket}$, $k \in \llbracket 0, S \rrbracket$

```

1:  $T \leftarrow$  tableau tel que  $T[i] \leftarrow (i + \lfloor \sqrt{N} \rfloor)^2 - N$  pour  $i \in \llbracket 1, S \rrbracket$ 
2:  $V \leftarrow$  tableau tel que  $V[i] \leftarrow (0, \dots, 0) \in \mathbb{N}^b$  pour  $i \in \llbracket 1, S \rrbracket$ 
3: pour  $p \in \{p_1, \dots, p_b\}$  tel que  $N$  est un carré modulo  $p$  faire
4:    $x_1, x_2 \leftarrow$  les racines de  $(X + \lfloor \sqrt{N} \rfloor)^2 - N$  modulo  $p$ 
5:   pour  $i \in \{1, 2\}$  faire
6:      $q \leftarrow x_i$ 
7:     tant que  $q \leq S$  faire
8:       tant que  $T[q] \bmod p = 0$  faire
9:          $T[q] \leftarrow T[q]/p$ 
10:         $V[q] \leftarrow V[q] + (0, \dots, 1, \dots, 0)$  (en position  $p$ )
11:         $q \leftarrow q + p$ 
renvoyer L'ensemble des  $(i + \lfloor \sqrt{N} \rfloor, V[i])$  tels que  $T[i] = 1$  pour
 $i \in \llbracket 1, S \rrbracket$ 

```

Plan

Introduction et enjeux

La méthode de Dixon

Congruences de carrés

Etapes de la méthode

L'algorithme final

Optimisations

Crible quadratique

Approximation logarithmique

Résultats

Annexe

Justification

- ▶ $O(n)$ au lieu de $O(n^2)$, voire $O(n \log n)$
- ▶ $Q(x) = \prod_{i=1}^k p_i^{\alpha_i}$, soit $\ln(Q(x)) = \sum_{i=1}^k \alpha_i \ln(p_i)$.
Idée: soustraire par $\alpha_i \ln(p_i)$ au lieu de diviser par $p_i^{\alpha_i}$
- ▶ $\log_2(Q(x)) \approx \text{nb_bits}(Q(x))$
- ▶ Problème: on ne connaît pas α_i .
Solution: on soustrait par $\log_2(p_i)$ seulement. Des approximations nécessitent déjà un **seuil**

Justification

- ▶ $O(n)$ au lieu de $O(n^2)$, voire $O(n \log n)$
- ▶ $Q(x) = \prod_{i=1}^k p_i^{\alpha_i}$, soit $\ln(Q(x)) = \sum_{i=1}^k \alpha_i \ln(p_i)$.
Idée: soustraire par $\alpha_i \ln(p_i)$ au lieu de diviser par $p_i^{\alpha_i}$
- ▶ $\log_2(Q(x)) \approx \text{nb_bits}(Q(x))$
- ▶ Problème: on ne connaît pas α_i .
Solution: on soustrait par $\log_2(p_i)$ seulement. Des approximations nécessitent déjà un **seuil**

Justification

- ▶ $O(n)$ au lieu de $O(n^2)$, voire $O(n \log n)$
- ▶ $Q(x) = \prod_{i=1}^k p_i^{\alpha_i}$, soit $\ln(Q(x)) = \sum_{i=1}^k \alpha_i \ln(p_i)$.
Idée: soustraire par $\alpha_i \ln(p_i)$ au lieu de diviser par $p_i^{\alpha_i}$
- ▶ $\log_2(Q(x)) \approx \text{nb_bits}(Q(x))$
- ▶ Problème: on ne connaît pas α_i .
Solution: on soustrait par $\log_2(p_i)$ seulement. Des approximations nécessitent déjà un **seuil**

Justification

- ▶ $O(n)$ au lieu de $O(n^2)$, voire $O(n \log n)$
- ▶ $Q(x) = \prod_{i=1}^k p_i^{\alpha_i}$, soit $\ln(Q(x)) = \sum_{i=1}^k \alpha_i \ln(p_i)$.
Idée: soustraire par $\alpha_i \ln(p_i)$ au lieu de diviser par $p_i^{\alpha_i}$
- ▶ $\log_2(Q(x)) \approx \text{nb_bits}(Q(x))$
- ▶ Problème: on ne connaît pas α_i .
Solution: on soustrait par $\log_2(p_i)$ seulement. Des approximations nécessitent déjà un **seuil**

$$S = 10$$

$$N = 20382493$$

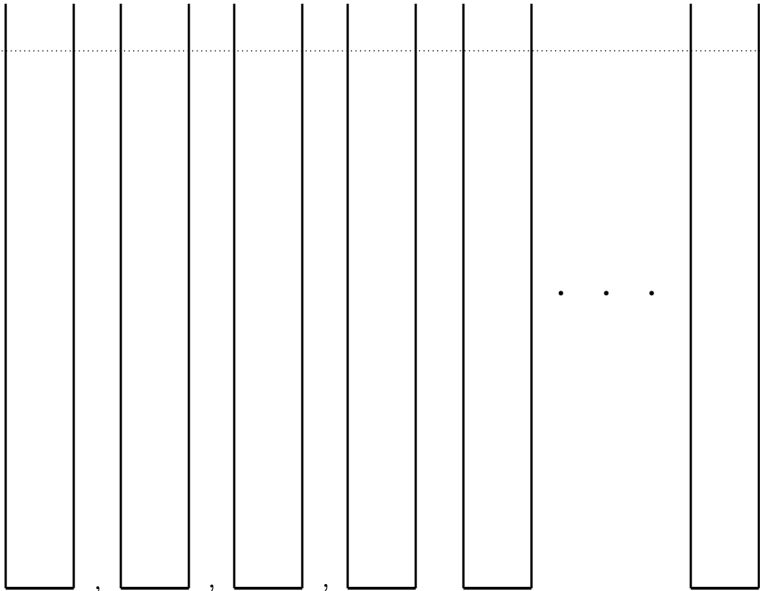
$$T = [\quad , \quad , \quad , \quad , \quad , \quad , \quad , \quad]$$

$$S = 10$$

$$N = 20382493$$

Seuil

$T = [$



$$S = 10$$

$$N = 20382493$$

$$Q(1) \equiv 0 \pmod{2}$$

Seuil

$T = [$

$\log_2 2$

,

+2

,

$\log_2 2$

,

+2

,

$\log_2 2$

,

...

,

$]$

$$S = 10$$

$$N = 20382493$$

$$Q(2) \equiv 0 \pmod{3}$$

$$Q(3) \equiv 0 \pmod{3}$$

Seuil

$T = [$

$\log_2 2$

,

$\log_2 3$

,

$\log_2 3$

$\log_2 2$

,

$\log_2 3$

$\log_2 2$

,

$]$

+3

+3

$$S = 10$$

$$N = 20382493$$

Seuil

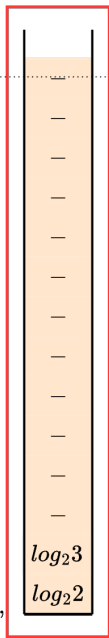
$$T = [\log_2 2, \log_2 3, \log_2 2, \log_2 3, \log_2 2]$$

$$S = 10$$

$$N = 20382493$$

Seuil

$$T = [\log_2 2, \log_2 3, \log_2 2, -, \log_2 3, \log_2 2]$$



Plan

Introduction et enjeux

La méthode de Dixon

 Congruences de carrés

 Etapas de la méthode

 L'algorithme final

Optimisations

 Crible quadratique

 Approximation logarithmique

Résultats

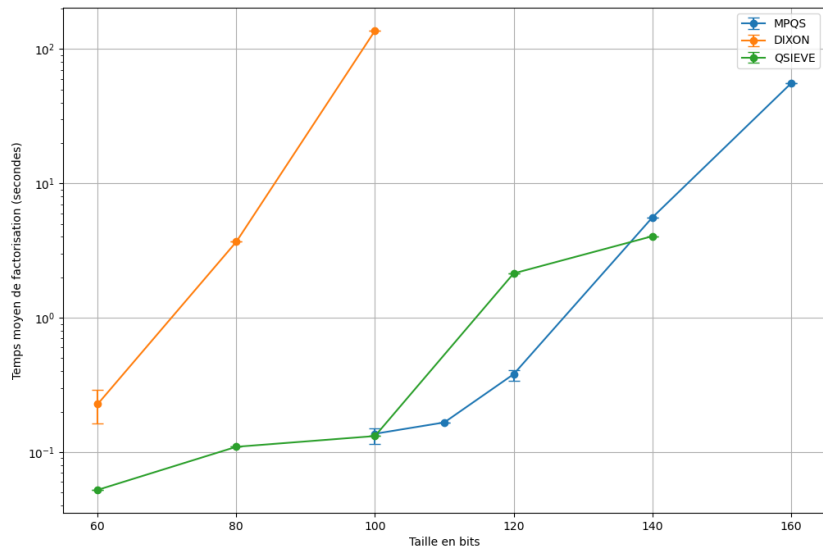
Annexe

Résultats

Après plusieurs centaines de tests, on a les résultats suivants:

Bits	Dixon	QSIEVE	MPQS
60	0.5s	0.05s	-
80	5s	0.1s	-
100	100s	0.1s	0.1s
120	-	2s	0.6s
140	-	5s	5s
160	-	-	80s

Graphique final



Plan

Introduction et enjeux

La méthode de Dixon

 Congruences de carrés

 Etapas de la méthode

 L'algorithme final

Optimisations

 Crible quadratique

 Approximation logarithmique

Résultats

Annexe

Proposition

Soient $b \in \mathbb{N}$, $(x_i)_{i \in \llbracket 1, b+1 \rrbracket} \in \mathbb{N}^{b+1}$ et $(v_i)_{i \in \llbracket 1, b+1 \rrbracket} \in \mathbb{F}_2^b$ les vecteurs valuations de $x_i^2 \pmod{N}$ pour $i \in \llbracket 1, b+1 \rrbracket$ et finalement $(\lambda_i)_{i \in \llbracket 1, b+1 \rrbracket} \in \{0, 1\}^{b+1}$ tels que,

$$\sum_{i=1}^{b+1} \lambda_i v_i = 0_{\mathbb{F}_2^b} = (2\alpha_1, \dots, 2\alpha_b)$$

On pose $y = \prod_{j=1}^b p_j^{\alpha_j}$ et $x = \prod_{j=1}^{b+1} x_j^{\lambda_j}$, alors $x^2 \equiv y^2 \pmod{N}$

Démonstration

$$\begin{aligned}
 x^2 &= \left(\prod_{i=1}^{b+1} x_i^2 \right)^{\lambda_i} \equiv \prod_{i=1}^{b+1} \prod_{j=1}^b p_j^{\lambda_i v_i^{(j)}} \pmod{N} \\
 &\equiv \prod_{j=1}^b \prod_{i=1}^{b+1} p_j^{\lambda_i v_i^{(j)}} \pmod{N} \\
 &\equiv \prod_{j=1}^b p_j^{\sum_{i=1}^{b+1} \lambda_i v_i^{(j)}} \pmod{N} \\
 &\equiv \left(\prod_{j=1}^b p_j^{\alpha_j} \right)^2 \pmod{N} \quad (\text{déf de } \alpha_j) \\
 &\equiv y^2 \pmod{N}
 \end{aligned}$$

Proposition

Si $Q = (\lfloor \sqrt{N} \rfloor + X)^2 - N$, alors
 $p \mid Q(x) \implies \forall k \in \mathbb{N}, p \mid Q(x + kp)$

Démonstration

En effet, supposons $p \mid Q(x)$, on a:

$$\begin{aligned} Q(x + kp) &= (\lfloor \sqrt{N} \rfloor + x + kp)^2 - N \\ &= Q(x) + 2kp(\lfloor \sqrt{N} \rfloor + x) + k^2 p^2 \\ &= Q(x) + p \times (2k(\lfloor \sqrt{N} \rfloor + x) + k^2 p) \end{aligned}$$

d'où $p \mid Q(x + kp)$

../c/vector.h

```
#pragma once  
#include <gmp.h>
```

```
void mod_vect(int* v, int mod, int n1);  
void add_vect(int* sum, int* op, int n1);
```

```
void div_vect(int* v, int d, int n1);  
void sub_vect(int** v, int i, int j, int n1);  
void prod_vect(mpz_t prod, mpz_t* z, int n1,  
               system_t s);
```

../c/vector.c

```
#include <gmp.h>
#include <assert.h>
#include <stdlib.h>
#include "system.h"

void mod_vect(int* v, int mod, int n1){
    for(int i = 0; i<n1; i++){
        v[i] = abs(v[i]) % mod;
    }
}

void add_vect(int* sum, int* op, int n1){
    for(int i = 0; i<n1; i++){
        sum[i] += op[i];
    }
}

void div_vect(int* v, int d, int n1){
    for(int i = 0; i<n1; i++){
        assert(v[i]%d == 0);
```

```
        v[i] /= d;
    }
}

void sub_vect(int** v, int i, int j, int n1){
    for(int k = 0; k<n1; k++){
        v[i][k] = v[i][k] - v[j][k];
    }
}

void prod_vect(mpz_t prod, mpz_t* z, int n1,
               system_t s){
    mpz_set_ui(prod, 1);
    for(int i = 0; i<n1; i++){
        if(s->sol[i]){
            mpz_mul(prod, prod, z[s->perm[i]]);
        }
    }
}
```

../c/tonellishanks.h

#pragma once

#include <gmp.h>

void tonelli_shanks_ui(mpz_t n, **int** p, **int*** x1, **int*** x2

);
void tonelli_shanks_mpz(mpz_t a, mpz_t p, mpz_t
x1, mpz_t x2);

../c/tonellishanks.c

```
#include <stdint.h>
#include <gmp.h>
#include <stdio.h>
#include <assert.h>
#include <stdlib.h>

uint64_t modpow(uint64_t a, uint64_t b, uint64_t n)
{
    uint64_t x = 1, y = a;
    while (b > 0) {
        if (b % 2 == 1) {
            x = (x * y) % n; // multiplying with base
        }
        y = (y * y) % n; // squaring the base
        b /= 2;
    }
    return x % n;
}

void tonelli_shanks_ui(mpz_t n, unsigned long int p,
    int* x1, int* x2) {
    uint64_t q = p - 1;
    uint64_t ss = 0;
    uint64_t z = 2;
    uint64_t c, r, t, m;

    while ((q & 1) == 0) {
        ss += 1;
        q >>= 1;
    }
}
```

```
    }

    mpz_t temp, pj;
    mpz_init(temp);
    mpz_init_set_ui(pj, p);

    if (ss == 1) {
        //uint64_t r1 = modpow(n, (p + 1) / 4, p);
        mpz_powm_ui(temp, n, (p+1)/4, pj);
        uint64_t r1 = mpz_get_ui(temp);

        *x1 = r1;
        *x2 = p - r1;
        mpz_clears(temp, pj, NULL);
        return;
    }

    while (modpow(z, (p - 1) / 2, p) != (unsigned
        long int) p - 1) { // uint_64 only there
        // for the compiler to stop complaining
        z++;
    }

    c = modpow(z, q, p);

    //r = modpow(n, (q + 1) / 2, p);
    mpz_powm_ui(temp, n, (q+1)/2, pj);
    r = mpz_get_ui(temp);

    //t = modpow(n, q, p);
}
```

```

mpz_powm_ui(temp, n, q, pj);
t = mpz_get_ui(temp);

m = ss;

while(1){
    uint64_t i = 0, zz = t;
    uint64_t b = c, e;
    if (t == 1) {
        *x1 = r;
        *x2 = p - r;
        mpz_clears(temp, pj, NULL);
        return;
    }
    while (zz != 1 && i < (m - 1)) {
        zz = zz * zz % p;
        i++;
    }
    e = m - i - 1;
    while (e > 0) {
        b = b * b % p;
        e--;
    }
    r = r * b % p;
    c = b * b % p;
    t = t * c % p;
    m = i;
}

}

void tonelli_shanks_mpz(mpz_t n, mpz_t p, mpz_t
    x1, mpz_t x2){
    assert(mpz_legendre(n, p) == 1);

```

```

    mpz_t q, z;
    mpz_init_set(q, p);
    mpz_sub_ui(q, q, 1);
    int ss = 0;
    mpz_init_set_ui(z, 2);

    while(mpz_divisible_ui_p(q, 2) != 0){
        ss += 1;
        mpz_divexact_ui(q, q, 2);
    }

    mpz_t op1;
    mpz_init(op1);

    if (ss == 1) {
        //uint64_t r1 = modpow(n, (p + 1) / 4, p);
        mpz_add_ui(op1, p, 1);
        mpz_divexact_ui(op1, op1, 4);
        mpz_powm(op1, n, op1, p);

        mpz_set(x1, op1);
        mpz_sub(x2, p, x1);

        mpz_clears(q, z, op1, NULL);
        return;
    }

    mpz_t op2, op3;
    mpz_inits(op2, op3, NULL);

    mpz_sub_ui(op1, p, 1);
    mpz_divexact_ui(op1, op1, 2);

```



```

mpz_powm(op2, z, op1, p);

mpz_sub_ui(op3, p, 1);
while(mpz_cmp(op2, op3) != 0){
    mpz_add_ui(z, z, 1);
    mpz_powm(op2, z, op1, p);
}

mpz_t c, r, t, m, i, zz, b, e;
mpz_inits(c, r, t, m, i, zz, b, e, NULL);
mpz_powm(c, z, q, p);

mpz_add_ui(op1, q, 1);
mpz_divexact_ui(op1, op1, 2);
mpz_powm(r, n, op1, p);

mpz_powm(t, n, q, p);

mpz_set_ui(m, ss);
while(1){
    mpz_set_ui(i, 0);
    mpz_set(zz, t);
    mpz_set(b, c);

    if(mpz_cmp_ui(t, 1) == 0){
        mpz_set(x1, r);
        mpz_sub(x2, p, x1);

        mpz_clears(c, r, t, m, i, zz, b, e, op1, op2
            , op3, q, z, NULL);
        return;
    }

    mpz_sub_ui(op1, m, 1);
    while(mpz_cmp_ui(zz, 1) != 0 && mpz_cmp
        (i, op1) < 0){
        mpz_mul(zz, zz, zz);
        mpz_mod(zz, zz, p);
        mpz_add_ui(i, i, 1);
    }

    mpz_sub(e, m, i);
    mpz_sub_ui(e, e, 1);
    while(mpz_sgn(e) > 0){
        mpz_mul(b, b, b);
        mpz_mod(b, b, p);
        mpz_sub_ui(e, e, 1);
    }

    mpz_mul(r, r, b);
    mpz_mod(r, r, p);

    mpz_mul(c, b, b);
    mpz_mod(c, c, p);

    mpz_mul(t, t, c);
    mpz_mod(t, t, p);

    mpz_set(m, i);
}

```

../c/system.h

```
#pragma once
#include <stdbool.h>

typedef struct system {
    int** m;
    int* perm;
    int* sol;
    bool done;
    int n1, n2, arb;
```

```
} system_s;

typedef system_s* system_t;

system_t init_gauss(int** v, int n1, int n2);
void gaussian_step(system_t s);
void free_system(system_t s);
```

../c/system.c

```
#include "system.h"
#include "vector.h"
#include "list_matrix_utils.h"
#include <stdlib.h>
#include <stdio.h>
#include <stdbool.h>

void swap_lines_horz(system_t s, int i, int j){
    int* temp = s->m[i];
    s->m[i] = s->m[j];
    s->m[j] = temp;
}

void swap_lines_vert(system_t s, int i, int j){
    int temp = s->perm[i];
    s->perm[i] = s->perm[j];
    s->perm[j] = temp;

    for(int k = 0; k<s->n1; k++){
        int temp = s->m[k][i];
        s->m[k][i] = s->m[k][j];
        s->m[k][j] = temp;
    }
}

int find_index(system_t s, int from, int look){
    for(int i = from; i < s->n1; i++){
        if(s->m[i][look]){
            return i;
        }
    }
}
```

```
    }
    return -1;
}

system_t transpose(int** v, int n1, int n2){
    system_t s = malloc(sizeof(system_s));

    s->m = malloc(n2*sizeof(int*));
    for(int i = 0; i<n2; i++){
        s->m[i] = malloc(n1*sizeof(int));
        for(int j = 0; j<n1; j++){
            s->m[i][j] = v[j][i];
        }
    }

    s->n1 = n2;
    s->n2 = n1;
    return s;
}

void triangulate(system_t s){
    s->perm = malloc(s->n2*sizeof(int));
    for(int i = 0; i<s->n2; i++){
        s->perm[i] = i;
    }

    int i = 0;
    int j = 0;
    while(i<s->n1 && j<s->n2){
        int k = find_index(s, i, j);
```

```

    if(k != -1){
        if(i != j){
            swap_lines_vert(s, i, j);
        }

        swap_lines_horz(s, i, k);

        for(int l = i + 1; l < s->n1; l++){
            if(s->m[l][i] == 1){
                sub_vect(s->m, l, i, s->n2);
                mod_vect(s->m[l], 2, s->n2);
            }
        }
        i++;
        j = i;
    }
    else{
        j++;
    }
}

}

void get_arbitrary(system_t triangulated){
    for(int i = triangulated->n1-1; i>=0; i--){
        int j = 0;
        while(j < triangulated->n2 && !triangulated
            ->m[i][j]){
            j++;
        }
        if(j<triangulated->n2){
            triangulated->arb = j+1;
            return;
        }
    }
}

```

```

    }

    fprintf(stderr, "ERROR: All vectors are zero in
        system\n");
    exit(1);
}

void init_sol(system_t s){
    s->sol = malloc(s->n2*sizeof(int));
    for(int i = s->arb; i<s->n2; i++){
        s->sol[i] = 0;
    }
}

void iter_sol(system_t s){
    int i = s->arb;
    while(i<s->n2 && (s->sol[i] == 1)){
        s->sol[i] = 0;
        i++;
    }
    if(i >= s->n2){
        s->done = true;
        return;
    }
    s->sol[i] = 1;
}

system_t init_gauss(int** v, int n1, int n2){
    //printf("Initial vectors\n");
    //print_ll(v, n1, n2);

    system_t s = transpose(v, n1, n2);
    s->done = false;
}

```

```

//printf("Transposed\n");
//print_ll(s->m, s->n1, s->n2);

for(int i = 0; i<s->n1; i++){
    mod_vect(s->m[i], 2, s->n2);
}

//printf("Modded\n");
//print_ll(s->m, s->n1, s->n2);

triangulate(s);

//printf("Triangulated\n");
//print_ll(s->m, s->n1, s->n2);

get_arbitrary(s);
init_sol(s);

return s;
}

void gaussian_step(system_t s){
    iter_sol(s);

    for(int i = s->n1-1; i>=0; i--){
        int j = 0;

```

```

        while(j < s->n2 && !s->m[i][j]){
            j++;
        }

        if(j<s->n2){
            s->sol[j] = 0;

            for(int k = s->n2-1; k>j; k--){
                s->sol[j] -= s->m[i][k] * s->sol[k];
            }

            s->sol[j] = abs(s->sol[j]) % 2;
        }
    }
}

void free_system(system_t s){
    for(int i = 0; i<s->n1; i++){
        free(s->m[i]);
    }
    free(s->m);
    free(s->sol);
    free(s->perm);
    free(s);
}

```

../c/parse_input.h

```
#pragma once
#include <gmp.h>
#include <stdbool.h>

typedef enum {DIXON, QSIEVE, MPQS, PMPQS}
    TYPE;

typedef struct input_s {
    char* output_file;
    int bound, sieving_interval;
```

```
    mpz_t N;
    bool quiet;
    TYPE algorithm;
    int extra;
    int delta;
} input_t;

input_t* parse_input(int argc, char** argv);
void free_input(input_t* input);
```

../c/parse_input.c

```
#include "parse_input.h"
#include <stdlib.h>
#include <string.h>
#include <gmp.h>
#include <stdbool.h>
```

```
input_t* init_input(void){
    input_t* input = malloc(sizeof(input_t));
    input->bound = -1;
    input->output_file = NULL;
    input->sieving_interval = -1;
    input->extra = -1;
    input->quiet = false;
    input->algorithm = QSIEVE;
    input->delta = 0;
    mpz_init_set_ui(input->N, 0);
    return input;
}
```

```
bool valid_int(char* str){
    int i = 0;
    char c = str[i];
    while(c != '\0'){
        if(c < 48 || c > 57) return false;
        c = str[++i];
    }
    return true;
}
```

```
void free_input(input_t* input){
    if(input->output_file) free(input->output_file);
    mpz_clear(input->N);
    free(input);
}
```

```
input_t* parse_input(int argc, char** argv){
    input_t* input = init_input();

    int i = 1;
    while(i < argc){
        if(strcmp(argv[i], "-b") == 0 || strcmp(argv[i],
            "--bound") == 0){
            i++;
            if(i < argc){
                if(valid_int(argv[i])) input->bound =
                    atoi(argv[i]);
                else return NULL;
            }
            else return NULL;
        }

        else if(strcmp(argv[i], "-s") == 0 || strcmp(
            argv[i], "--sieving_interval") == 0){
            i++;
            if(i < argc){
                if(valid_int(argv[i])) input->
                    sieving_interval = atoi(argv[i]);
                ;
                else return NULL;
            }
            else return NULL;
        }
    }
}
```

```

}

else if(strcmp(argv[i], "-e") == 0 || strcmp(
    argv[i], "--extra") == 0){
    i++;
    if(i < argc){
        if(valid_int(argv[i])) input->extra =
            atoi(argv[i]);
        else return NULL;
    }
    else return NULL;

}

else if(strcmp(argv[i], "-n") == 0 || strcmp(
    argv[i], "--number") == 0){
    i++;
    if(i < argc){
        if(valid_int(argv[i])) mpz_set_str(
            input->N, argv[i], 10);
        else return NULL;
    }
    else return NULL;

}

else if(strcmp(argv[i], "-d") == 0 || strcmp(
    argv[i], "--delta") == 0){
    i++;
    if(i < argc){
        if(valid_int(argv[i])) input->delta =
            atoi(argv[i]);
        else return NULL;
    }
    else return NULL;

}

else if(strcmp(argv[i], "-o") == 0){

```

```

    i++;
    if(i < argc) input->output_file = argv[i];
    else return NULL;

}

else if(strcmp(argv[i], "-t") == 0 || strcmp(
    argv[i], "--type") == 0){
    i++;
    if(i < argc) {
        if(strcmp(argv[i], "dixon") == 0)
            input->algorithm = DIXON;
        else if(strcmp(argv[i], "qsieve") ==
            0) input->algorithm =
            QSIEVE;
        else if(strcmp(argv[i], "mpqs") == 0)
            input->algorithm = MPQS;
        else if(strcmp(argv[i], "pmpqs") ==
            0) input->algorithm =
            PMPQS;
        else return NULL;
    }
    else return NULL;

}

else if(strcmp(argv[i], "-q") == 0 ||
    strcmp(argv[i], "-stfu") == 0 /*
        easter egg*/ ||
    strcmp(argv[i], "--quiet") == 0){
    input->quiet = true;
}

else return NULL;

i++;

```



```
}
```

```
}
```

```
return input;
```

../c/list_matrix_utils.h

```
#pragma once
```

```
void print_list(int* l, int n);
```

```
void print_ll(int** ll, int n1, int n2);
```

```
void free_ll(int** m, int n1);
```

../c/list_matrix_utils.c

```
#include <stdio.h>
#include <stdlib.h>
```

```
void print_list(int* l, int n){
    for(int i = 0; i<n; i++){
        printf("%d_", l[i]);
    }
    printf("\n");
}

void print_ll(int** ll, int n1, int n2){
    for(int i = 0; i<n1; i++){
```

```
        print_list(ll[i], n2);
    }
    printf("\n");
}

void free_ll(int** m, int n1){
    for(int i = 0; i<n1; i++){
        free(m[i]);
    }
    free(m);
}
```

../c/factorbase.h

```
#pragma once  
#include <gmp.h>
```

```
// bruh  
bool is_prime(int n);
```

```
// calculates  $\pi(n)$ , the number of prime numbers  $\leq n$   
int pi(int n);
```

```
// returns a list of piB first primes
```

```
int* primes(int piB, int B);
```

```
/** Reduces the factor base of the algorithm, refer to:  
 * Quadratic sieve factorisation algorithm  
 * Bc. Ondřej Vladyka  
 * Section 2.3.1 (p.16)  
 */  
int* prime_base(mpz_t n, int* pb_len, int* primes,  
                int piB);
```

../c/factorbase.c

```
#include <stdbool.h>
#include <gmp.h>
#include <stdlib.h>

bool is_prime(int n) {
    // Corner cases
    if (n <= 1)
        return false;
    if (n <= 3)
        return true;

    // This is checked so that we can skip
    // middle five numbers in below loop
    if (n % 2 == 0 || n % 3 == 0)
        return false;

    for (int i = 5; i * i <= n; i = i + 6)
        if (n % i == 0 || n % (i + 2) == 0)
            return false;

    return true;
}

int pi(int n) {
    int k = 0;
    for (int i = 2; i <= n; i++) {
        if (is_prime(i)) k++;
    }
    return k;
}
```

```
int* primes(int piB, int B){
    int* p = malloc(piB*sizeof(int));
    int k = 0;
    for (int i = 2; i <= B; i++) {
        if (is_prime(i)){
            p[k] = i;
            k++;
        }
    }
    return p;
}

/* Used for legendre symbol, exists in gmp already
bool euler_criterion(mpz_t n, int p){
    int e = (p-1)/2;
    mpz_t r, p1;
    mpz_init(r);
    mpz_init_set_ui(p1, p);
    mpz_powm_ui(r, n, e, p1);
    return(mpz_cmp_ui(r, 1) == 0);
}
*/

int* prime_base(mpz_t n, int* pb_len, int* primes,
               int piB){
    int* pb = malloc(piB*sizeof(int));
    pb[0] = 2;
```

```
int j = 1;
mpz_t p1;
mpz_init(p1);
for(int i = 1; i < piB; i++){
    mpz_set_ui(p1, primes[i]);
    if(mpz_legendre(n, p1) == 1){
        //printf("%d\n", primes[i]);
        pb[j] = primes[i];
        j++;
    }
}
```

```
    }
}
*pb_len = j;
pb = realloc(pb, (j+1)*sizeof(int)); // +1 used
    for mpqs

mpz_clear(p1);
return pb;
}
```

../c/main.c

```
#include <stdbool.h>
#include <gmp.h>
#include <sys/time.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include "system.h"
#include "vector.h"
#include "parse_input.h"
#include "factorbase.h"
#include "list_matrix_utils.h"

// Include algorithms
// Dixon's method
#include "../dixon/dixon.h"

// The Quadratic Sieve
#include "../qsieve/qsieve.h"

// Multipolynomial Quadratic Sieve
#include "../mpqs/polynomial.h"
#include "../mpqs/mpqs.h"
#include "../mpqs/parallel_mpqs.h"

/**
 *
 *
 * START OF ALGORITHM
 *
```

```
*/

void rebuild_mpqs(mpz_t prod, mpz_t* d, int* v, int*
    primes, int n1, system_t s){
    mpz_set_ui(prod, 1);
    mpz_t temp;
    mpz_init(temp);
    for(int i = 0; i < n1; i++){
        if(s->sol[i]){
            mpz_mul(prod, prod, d[s->perm[i]]);
        }
        mpz_ui_pow_ui(temp, primes[i], v[i]);
        mpz_mul(prod, prod, temp);
    }
    mpz_clear(temp);
}

void rebuild(mpz_t prod, int* v, int* primes, int n1){
    /** Rebuilds the product of primes to the power of
        half
        * the solution found by the gaussian solve

        * EX:
        * v = (1, 2, 3, 1)
        * primes = [2, 3, 5, 7]
        * prod = 2**1 * 3**2 * 5**3 * 7**1
        * returns prod
        *
        */
}
```

```

    mpz_set_ui(prod, 1);
    mpz_t temp;
    mpz_init(temp);
    for(int i = 0; i < n1; i++){
        mpz_ui_pow_ui(temp, primes[i], v[i]);
        mpz_mul(prod, prod, temp);
    }
    mpz_clear(temp);
}

void sum_lignes(int* sum, int** v, system_t s){
    /** Sums the lines of vectors into 'sum' according
        the solution of the
        * output of the system 's', such that each power
        is even
    */
    for(int i = 0; i < s->n1; i++){
        sum[i] = 0;
    }

    for(int i = 0; i < s->n2; i++){
        if(s->sol[i]){
            add_vect(sum, v[s->perm[i]], s->n1);
        }
    }
}

void factor(input_t* input){
    int piB = pi(input->bound);
    if(!input->quiet) printf("pi(B) = %d\n", piB);
    int* p = primes(piB, input->bound);

```

```

    int pb_len;
    int* pb;
    switch(input->algorithm){
        case DIXON:
            pb = p;
            pb_len = piB;
            break;
        case QSIEVE:
            pb = prime_base(input->N, &pb_len, p,
                             piB);
            if(!input->quiet) printf("base_reduction_
                                   %f%%\n", (float)pb_len/piB
                                   *100);
            free(p);
            break;
        case MPQS:
            pb = prime_base(input->N, &pb_len, p,
                             piB);
            pb[pb_len] = -1;
            if(!input->quiet) printf("base_reduction_
                                   %f%%\n", (float)pb_len/piB
                                   *100);
            free(p);
            break;
        case PMPQS:
            pb = prime_base(input->N, &pb_len, p,
                             piB);
            pb[pb_len] = -1;
            if(!input->quiet) printf("base_reduction_
                                   %f%%\n", (float)pb_len/piB
                                   *100);
            free(p);
            break;
    }
}

```



```

}
int target_nb = pb_len + input->extra;

mpz_t* z = malloc((target_nb)*sizeof(mpz_t));
for(int i = 0; i < target_nb; i++){
    mpz_init(z[i]);
}

//Getting zis
int** v;
mpz_t* d;
struct timeval t1, t2;
gettimeofday(&t1, 0);
switch(input->algorithm){
    case DIXON:
        v = dixon(z, input->N, pb_len, pb, input
            ->extra, input->quiet);
        break;
    case QSIEVE:
        v = qsieve(z, input->N, pb_len, pb,
            input->extra, input->
            sieving_interval, input->quiet);
        break;
    case MPQS:
        d = malloc(target_nb*sizeof(mpz_t));
        for(int i = 0; i < target_nb; i++){
            mpz_init(d[i]);
        }
        v = mpqs(z, d, input->N, pb_len, pb,
            input->extra, input->
            sieving_interval, input->delta,
            input->quiet);
        break;

```

```

    case PMPQS:
        d = malloc(target_nb*sizeof(mpz_t));
        for(int i = 0; i < target_nb; i++){
            mpz_init(d[i]);
        }
        v = parallel_mpqs(z, d, input->N,
            pb_len, pb, input->extra, input
            ->sieving_interval, input->delta,
            input->quiet);
        break;
}

gettimeofday(&t2, 0);
long seconds = t2.tv_sec - t1.tv_sec;
long microseconds = t2.tv_usec - t1.tv_usec;
double time_spent = seconds + microseconds*1e
    -6;
if(!input->quiet) printf("Time to get zis: %fs\n",
    time_spent);

mpz_t f, Z1, Z2, test1, test2;
mpz_inits(f, Z1, Z2, test1, test2, NULL);

//gaussian init
system_t s;
int* sum;
switch(input->algorithm){
    case DIXON:
        s = init_gauss(v, target_nb, pb_len);
        sum = malloc(pb_len*sizeof(int));
        break;
    case QSIEVE:
        s = init_gauss(v, target_nb, pb_len);

```

```

        sum = malloc(pb_len*sizeof(int));
        break;
    case MPQS:
        // for -1
        s = init_gauss(v, target_nb, pb_len+1);
        sum = malloc((pb_len+1)*sizeof(int));
        break;
    case PMPQS:
        // for -1
        s = init_gauss(v, target_nb, pb_len+1);
        sum = malloc((pb_len+1)*sizeof(int));
        break;
}
if(!input->quiet) printf("2^%d solutions to L
iterate\n", s->n2 - s->arb);

bool done = false;
while(!done){
    gaussian_step(s);

    prod_vect(Z1, z, target_nb, s);
    sum_lignes(sum, v, s);
    div_vect(sum, 2, pb_len);

    switch(input->algorithm){
        case DIXON:
            rebuild(Z2, sum, pb, pb_len);
            break;
        case QSIEVE:
            rebuild(Z2, sum, pb, pb_len);
            break;
        case MPQS:
            rebuild_mpqqs(Z2, d, sum, pb, pb_len,

```

```

            s);
            break;
        case PMPQS:
            rebuild_mpqqs(Z2, d, sum, pb, pb_len,
            s);
            break;
    }

    // TEST
    mpz_set(test1, Z1);
    mpz_mul(test1, test1, test1);
    mpz_set(test2, Z2);
    mpz_mul(test2, test2, test2);
    assert(mpz_congruent_p(test1, test2, input
        ->N) != 0);
    // END TEST

    mpz_sub(f, Z1, Z2);
    mpz_gcd(f, f, input->N);

    if(mpz_cmp_ui(f, 1) != 0 && mpz_cmp(f,
        input->N) != 0){
        assert(mpz_divisible_p(input->N, f));
        if(!input->quiet) gmp_printf("%Zd=0
            %Zd\n", input->N, f);
        done = true;
    }

    mpz_add(f, Z1, Z2);
    mpz_gcd(f, f, input->N);

    if(mpz_cmp_ui(f, 1) != 0 && mpz_cmp(f,
        input->N) != 0){

```

```

    assert(mpz_divisible_p(input->N, f));
    if(!input->quiet) gmp_printf("%Zd\n", input->N, f);
    done = true;
}

if(s->done){
    if(!input->quiet) fprintf(stderr, "ERROR:
no solution for this set of zi\n"
    );
    exit(1);
}

}

free(sum);
free(pb);
free_system(s);
free_ll(v, target_nb);
for(int i = 0; i < target_nb; i++){
    mpz_clear(z[i]);
}
free(z);
switch(input->algorithm){
    case DIXON:
        break;
    case QSIEVE:
        break;
    case MPQS:
        for(int i = 0; i < target_nb; i++){
            mpz_clear(d[i]);
            free(d);
            break;
        }
    case PMPQS:

```

```

        for(int i = 0; i < target_nb; i++){
            mpz_clear(d[i]);
            free(d);
            break;
        }

    mpz_clears(f, Z1, Z2, test1, test2, NULL);
}

int main(int argc, char** argv){
    input_t* input = parse_input(argc, argv);
    if(input==NULL){
        fprintf(stderr, "ERROR: Invalid input\n");
        return 1;
    }

    if(mpz_cmp_ui(input->N, 0) == 0){
        fprintf(stderr, "ERROR: No input number,
use -n %%number%%\n");
        return 1;
    }

    if(input->bound == -1) input->bound =
        10000;
    if(input->sieving_interval == -1) input->
        sieving_interval = 100000;
    if(input->extra == -1) input->extra = 1;

    struct timeval t1, t2;
    gettimeofday(&t1, 0);
    factor(input);
    gettimeofday(&t2, 0);
}

```

```
long seconds = t2.tv_sec - t1.tv_sec;
long microseconds = t2.tv_usec - t1.tv_usec;
double time_spent = seconds + microseconds*1e-6;
if(!input->quiet) printf("Total time: %fs\n", time_spent);
}

free_input(input);
return 0;
```

../c/dixon/dixon.h

`#pragma once`

`int extra, bool tests);`

`int** dixon(mpz_t* z, mpz_t N, int pb_len, int* pb,`

../c/dixon/dixon.c

```
#include <gmp.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>

bool vectorize_dixon(mpz_t n, int* v, int pb_len, int*
    pb){
    /** Attempts naive factorisation to 'n' with the
        primes in
        * the prime base 'pb' and putting the result into '
        v', vector of powers of
        * the primes in the prime base
        * If it succeeds, returns true, otherwise, returns
        false
    */
    for(int i = 0; i < pb_len; i++){
        v[i] = 0;
    }

    for(int i = 0; i < pb_len && (mpz_cmp_ui(n, 1)
        != 0); i++){
        while (mpz_divisible_ui_p(n, pb[i])){
            v[i]++;
            mpz_divexact_ui(n, n, pb[i]);
        }
    }

    if(mpz_cmp_ui(n, 1) == 0)
        return true;
    return false;
}
```

```
}

int** dixon(mpz_t* z, mpz_t N, int pb_len, int* pb,
    int extra, bool tests){
    /** Gets pb_len+extra b-smooth realtions
        defined at:
        * Quadratic sieve factorisation algorithm
        * Bc. Ondřej Grej Vladyka
        * Definition 1.11 (p.5)
    */

    //ceil(sqrt(n))
    mpz_t sqrt_N;
    mpz_init(sqrt_N);
    mpz_sqrt(sqrt_N, N);
    mpz_add_ui(sqrt_N, sqrt_N, 1);

    mpz_t zi;
    mpz_t zi_cpy;
    mpz_init_set(zi, sqrt_N);
    mpz_init(zi_cpy);

    int** v = malloc((pb_len+extra)*sizeof(int*));

    for(int i = 0; i < pb_len+extra; i++){
        bool found = false;
        int* vi = malloc(pb_len*sizeof(int));

        while(!found){
```

```

    mpz_add_ui(zi, zi, 1);
    mpz_mul(zi_cpy, zi, zi);
    mpz_mod(zi_cpy, zi_cpy, N);

    found = vectorize_dixon(zi_cpy, vi,
        pb_len, pb);
}
if(!tests){
    printf("\r");
    printf("%.1f%%%", (float)i/(pb_len+extra
        -1)*100);
    fflush(stdout);
}

    v[i] = vi;
    mpz_set(z[i], zi);
}
if(!tests) printf("\n");

    mpz_clears(sqrt_N, zi, zi_cpy, NULL);

    return v;
}

```

../c/qsieve/qsieve.h

```
#pragma once  
#include <gmp.h>  
#include <stdbool.h>
```

```
bool vectorize_qsieve(mpz_t n, int* v, int pb_len, int*
```

```
    pb);  
int** qsieve(mpz_t* z, mpz_t N, int pb_len, int* pb,  
    int extra, int s, bool tests);
```


../c/qsieve/qsieve.c

```
#include <gmp.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <math.h>
```

```
#include "../system.h"
#include "../tonellishanks.h"
```

```
bool vectorize_qsieve(mpz_t n, int* v, int pb_len, int*
pb){
    /** Attempts naive factorisation to 'n' with the
        primes in
        * the prime base 'pb' and putting the result into '
        v', vector of powers of
        * the primes in the prime base
        * If it succeeds, returns true, otherwise, returns
        false
    */
    for(int i = 0; i < pb_len; i++){
        v[i] = 0;
    }

    for(int i = 0; i < pb_len && (mpz_cmp_ui(n, 1)
!= 0); i++){
        while (mpz_divisible_ui_p(n, pb[i])){
            v[i]++;
            mpz_divexact_ui(n, n, pb[i]);
        }
    }
}
```

```
    }

    if(mpz_cmp_ui(n, 1) == 0)
        return true;
    return false;
}

float* prime_logs(int* pb, int pb_len){
    float* plogs = malloc(pb_len*sizeof(float));

    for(int i = 0; i < pb_len; i++){
        plogs[i] = log2(pb[i]);
    }

    return plogs;
}

int calculate_threshold(mpz_t N, mpz_t sqrt_N, int
s, int loop_number, int* pb, int pb_len){

    mpz_t qstart;
    mpz_init_set_ui(qstart, s);
    mpz_mul_ui(qstart, qstart, loop_number);
    mpz_add(qstart, qstart, sqrt_N);
    mpz_mul(qstart, qstart, qstart);
    mpz_sub(qstart, qstart, N);

    int t = mpz_sizeinbase(qstart, 2) - (int) log2(pb[
pb_len-1]);
    mpz_clear(qstart);
}
```

```

    return t;
}

int** qsieve(mpz_t* z, mpz_t N, int pb_len, int* pb,
             int extra, int s, bool quiet){
    /** Gets pb_len+extra zis that are b-smooth,
        defined at:
        * Quadratic sieve factorisation algorithm
        * Bc. Ondřej Grej Vladyka
        * Definition 1.11 (p.5)
        */

    //ceil(sqrt(n))
    mpz_t sqrt_N;
    mpz_init(sqrt_N);
    mpz_sqrt(sqrt_N, N);
    mpz_add_ui(sqrt_N, sqrt_N, 1);

    mpz_t zi;
    mpz_init_set(zi, sqrt_N);
    mpz_t qx;
    mpz_init(qx);

    int** v = malloc((pb_len+extra)*sizeof(int*));
    for(int i = 0; i < pb_len+extra; i++){
        v[i] = malloc(pb_len*sizeof(int*));
    }
    float* sinterval = malloc(s*sizeof(float));
    float* plogs = prime_logs(pb, pb_len);

    // TESTS
    mpz_t temp;

```

```

    mpz_init(temp);
    // END TESTS

    int* x1 = malloc(pb_len*sizeof(int));
    int* x2 = malloc(pb_len*sizeof(int));

    // find solution for 2
    mpz_set(temp, sqrt_N);
    mpz_mul(temp, temp, temp);
    mpz_sub(temp, temp, N);
    x1[0] = 0;
    if(mpz_divisible_ui_p(temp, 2) == 0) x1[0] = 1;

    int sol1, sol2;
    for(int i = 1; i < pb_len; i++){
        tonelli_shanks_ui(N, pb[i], &sol1, &sol2);
        x1[i] = sol1;
        x2[i] = sol2;

        // change solution from  $x\hat{A}^2 = n \pmod{p}$  to (
             $\sqrt{N} + x\hat{A}^2 = n \pmod{p}$ 
        mpz_set_ui(temp, x1[i]);
        mpz_sub(temp, temp, sqrt_N);
        mpz_mod_ui(temp, temp, pb[i]);

        x1[i] = mpz_get_ui(temp);

        mpz_set_ui(temp, x2[i]);
        mpz_sub(temp, temp, sqrt_N);
        mpz_mod_ui(temp, temp, pb[i]);
    }

```

```

        x2[i] = mpz_get_ui(temp);
    }
    mpz_clear(temp);

    int loop_number = 0;
    int relations_found = 0;
    int tries = 0;
    while(relations_found < pb_len + extra){

        for(int i = 0; i<s; i++){
            sinterval[i] = 0;
        }

        // sieve for 2
        while(x1[0]<s){
            sinterval[x1[0]] += plogs[0];
            x1[0] += pb[0];
        }
        x1[0] = x1[0] - s;

        // sieve other primes
        for(int i = 1; i < pb_len; i++){

            while(x1[i]<s){
                sinterval[x1[i]] += plogs[i];
                x1[i] += pb[i];
            }

            while(x2[i]<s){

                sinterval[x2[i]] += plogs[i];
                x2[i] += pb[i];
            }
        }
    }

```

```

        //next interval
        x1[i] = x1[i] - s;
        x2[i] = x2[i] - s;
    }

    int t = calculate_threshold(N, sqrt_N, s,
                                loop_number, pb, pb_len);
    //printf("t = %d\n", t);

    bool found;
    for(int i = 0; i<s && relations_found <
        pb_len + extra; i++){
        if(sinterval[i] > t){
            tries++;

            // zi = sqrt(n) + x where x = s*
            // loopnumber + i
            mpz_set_ui(zi, s);
            mpz_mul_ui(zi, zi, loop_number);
            mpz_add_ui(zi, zi, i);
            mpz_add(zi, zi, sqrt_N);

            // qx = zi**2 - N
            mpz_mul(qx, zi, zi);
            mpz_sub(qx, qx, N);

            found = vectorize_qsieve(qx, v[
                relations_found], pb_len, pb);

            if(found){
                mpz_set(z[relations_found], zi);
                relations_found++;
            }
        }
    }

```

```

        found = false;
        if(!quiet){
            printf("\r");
            printf("0.1f%%%u|u%.1f%%",
                (float)
                relations_found/(
                pb_len+extra)*100, (
                float)relations_found/
                tries*100);
            fflush(stdout);
        }
    }
}

        loop_number++;
    }

    if(!quiet) printf("\n");

    mpz_clears(sqrt_N, zi, qx, NULL);
    free(x1);
    free(x2);
    free(sinterval);
    free(plogs);

    return v;
}

```

../c/mpqs/common_mpqs.h

```
#pragma once  
#include <gmp.h>  
#include <stdbool.h>
```

```
int calculate_threshold_mpqs(mpz_t sqrt_N, int s,  
    int* pb, int pb_len, int delta);
```

```
float* prime_logs_mpqs(int* pb, int pb_len);  
bool vectorize_mpqs(mpz_t n, int* v, int pb_len, int*  
    pb);  
bool already_added(mpz_t zi, mpz_t* z, int  
    relations_found);
```

../c/mpqs/common_mpqs.c

```
#include <gmp.h>
#include <stdbool.h>
#include <math.h>
#include <stdlib.h>
#include <stdio.h>

int calculate_threshold_mpqs(mpz_t sqrt_N, int s,
    int* pb, int pb_len, int delta){

    mpz_t qstart;
    mpz_init_set_ui(qstart, s);
    mpz_mul(qstart, qstart, sqrt_N);

    int t = mpz_sizeinbase(qstart, 2) - (int) log2(pb[
        pb_len-1]) - delta;
    mpz_clear(qstart);
    return t;
}

float* prime_logs_mpqs(int* pb, int pb_len){
    float* plogs = malloc(pb_len*sizeof(float));

    for(int i = 0; i<pb_len; i++){
        plogs[i] = log2(pb[i]);
    }

    return plogs;
}

bool vectorize_mpqs(mpz_t n, int* v, int pb_len, int*
```

```
pb){
    /** Attempts naive factorisation to 'n' with the
        primes in
        * the prime base 'pb' and putting the result into '
          v', vector of powers of
        * the primes in the prime base
        * If it succeeds, returns true, otherwise, returns
          false
    */
    for(int i = 0; i<pb_len; i++){
        v[i] = 0;
    }
    if(mpz_sgn(n)<0){
        v[pb_len] = 1;
        mpz_neg(n, n);
    }
    else{
        v[pb_len] = 0;
    }

    for(int i = 0; i<pb_len && (mpz_cmp_ui(n, 1)
        != 0); i++){
        while (mpz_divisible_ui_p(n, pb[i])){
            v[i]++;
            mpz_divexact_ui(n, n, pb[i]);
        }
    }

    if(mpz_cmp_ui(n, 1) == 0)
        return true;
}
```

```
    return false;
}

bool already_added(mpz_t zi, mpz_t* z, int
relations_found){
    for(int i = 0; i<relations_found; i++){
        if(mpz_cmp(zi, z[i]) == 0){
```

```
            return true;
        }
    }
    return false;
}
```

../c/mpqs/polynomial.h

```
#pragma once
#include <gmp.h>
#include <stdbool.h>
```

```
struct poly_s {
    mpz_t d;
    mpz_t N;

    mpz_t a;
    mpz_t b;
    mpz_t c;

    mpz_t zi;
    mpz_t qx;
```

```
// used to make operations without declaring and
// freeing everytime
mpz_t op1, op2, op3;
};
```

```
typedef struct poly_s* poly_t;
```

```
void get_next_poly(poly_t p);
poly_t init_poly(mpz_t N, int M);
void calc_poly(poly_t p, mpz_t x);
poly_t copy_poly(poly_t p);
void free_poly(poly_t p);
```


../c/mpqs/polynomial.c

```
#include "polynomial.h"
```

```
#include <gmp.h>
```

```
#include <stdlib.h>
```

```
#include <assert.h>
```

```
#include <stdio.h>
```

```
#include "../tonellishanks.h"
```

```
void calc_coefficients(poly_t p){  
    mpz_mul(p->a, p->d, p->d);
```

```
    mpz_t x1, x2;
```

```
    mpz_inits(x1, x2, NULL);
```

```
    tonelli_shanks_mmpz(p->N, p->d, x1, x2);
```

```
    // getting ready for congruence solve for raising  
    solution
```

```
    mpz_mul_ui(p->op1, x1, 2);
```

```
    mpz_mul(p->op2, x1, x1);
```

```
    mpz_sub(p->op2, p->op2, p->N);
```

```
    mpz_divexact(p->op2, p->op2, p->d);
```

```
    mpz_neg(p->op2, p->op2);
```

```
    mpz_mod(p->op2, p->op2, p->d);
```

```
    mpz_t g, n, m;
```

```
    mpz_inits(g, n, m, NULL);
```

```
    mpz_gcdext(g, n, m, p->d, p->op1);
```

```
    assert(mpz_cmp_ui(g, 1) == 0);
```

```
    mpz_mul(p->op1, p->op2, m); // t
```

```
    mpz_clears(g, n, m, NULL);
```

```
    mpz_set(p->b, p->d);
```

```
    mpz_mul(p->b, p->b, p->op1);
```

```
    mpz_add(p->b, p->b, x1);
```

```
    mpz_mul(p->op1, p->b, p->b);
```

```
    assert(mpz_congruent_p(p->op1, p->N, p->a)  
           != 0);
```

```
    mpz_sub(p->c, p->op1, p->N);
```

```
    mpz_divexact(p->c, p->c, p->a);
```

```
    mpz_clears(x1, x2, NULL);
```

```
}
```

```
void get_next_poly(poly_t p){
```

```
    mpz_nextprime(p->d, p->d);
```

```
    while(mpz_legendre(p->N, p->d) != 1){  
        mpz_nextprime(p->d, p->d);
```

```
    }  
    calc_coefficients(p);
```

```
}
```

```
poly_t init_poly(mpz_t N, int M){  
    poly_t p = malloc(sizeof(struct poly_s));
```

```
    mpz_inits(p->d, p->N, p->a, p->b, p->c, p->  
              ->op1, p->op2, p->op3, p->zi, p->  
              qx, NULL);
```

```

mpz_set(p->N, N);

// choose value of d according to 2.4.2
// sqrt( (sqrt(2N))/M )
mpz_mul_ui(p->op1, N, 2);
mpz_sqrt(p->op1, p->op1);
mpz_div_ui(p->op1, p->op1, M);
mpz_sqrt(p->op1, p->op1);
mpz_prevprime(p->d, p->op1);

// get next prime such that (n/p) = 1
while(mpz_legendre(N, p->d) != 1){
    mpz_nextprime(p->d, p->d);
}

calc_coefficients(p);
return p;
}

void calc_poly(poly_t p, mpz_t x){
    mpz_mul(p->zi, p->a, x);
    mpz_add(p->zi, p->zi, p->b);

    mpz_mul(p->qx, x, x);
    mpz_mul(p->qx, p->qx, p->a);

    mpz_mul(p->op1, p->b, x);
    mpz_mul_ui(p->op1, p->op1, 2);
    mpz_add(p->qx, p->qx, p->op1);

```

```

    mpz_add(p->qx, p->qx, p->c);
}

void free_poly(poly_t p){
    mpz_clears(p->d, p->N, p->a, p->b, p->c,
               p->op1, p->op2, p->op3, p->zi, p->
               qx, NULL);
    free(p);
}

poly_t copy_poly(poly_t p){
    poly_t cpy = malloc(sizeof(struct poly_s));

    mpz_inits(cpy->d, cpy->N, cpy->a, cpy->b,
              cpy->c, cpy->op1, cpy->op2, cpy->
              op3, cpy->zi, cpy->qx, NULL);

    mpz_set(cpy->d, p->d);
    mpz_set(cpy->N, p->N);

    mpz_set(cpy->a, p->a);
    mpz_set(cpy->b, p->b);
    mpz_set(cpy->c, p->c);

    return cpy;
}

```

../c/mpqs/mpqs.h

```
#pragma once
```

```
#include <gmp.h>
```

```
#include <stdbool.h>
```

```
int** mpqs(mpz_t* z, mpz_t* d, mpz_t N, int pb_len  
    , int* pb, int extra, int s, int delta, bool quiet);
```

../c/mpqs/mpqs.c

```
#include <gmp.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <math.h>
#include <time.h>

#include "polynomial.h"
#include "common_mpqs.h"
#include "../system.h"
#include "../tonellishanks.h"

int** mpqs(mpz_t* z, mpz_t* d, mpz_t N, int pb_len
, int* pb, int extra, int s, int delta, bool quiet){
    /** Gets pb_len+extra zis that are b-smooth,
        defined at:
        * Quadratic sieve factorisation algorithm
        * Bc. Ondřej Vladyka
        * Definition 1.11 (p.5)
        */

    //ceil(sqrt(n))
    mpz_t sqrt_N;
    mpz_init(sqrt_N);
    mpz_sqrt(sqrt_N, N);
    mpz_add_ui(sqrt_N, sqrt_N, 1);

    mpz_t x;
```

```
    mpz_init(x);
    poly_t Q = init_poly(N, s);

    int** v = malloc((pb_len+extra)*sizeof(int*));
    for(int i = 0; i < pb_len+extra; i++){
        v[i] = malloc((pb_len+1)*sizeof(int*)); //
            +1 for -1
    }
    float* sinterval = malloc(2*s*sizeof(float));
    float* plogs = prime_logs_mpqs(pb, pb_len);
    int t = calculate_threshold_mpqs(sqrt_N, s, pb,
        pb_len, delta);

    // TESTS
    mpz_t temp;
    mpz_init(temp);
    // END TESTS

    int* r = malloc(pb_len*sizeof(int));
    int* x1 = malloc(pb_len*sizeof(int));
    int* x2 = malloc(pb_len*sizeof(int));

    int sol1, sol2;
    for(int i = 1; i < pb_len; i++){
        tonelli_shanks_ui(N, pb[i], &sol1, &sol2);
        r[i] = sol1;
    }
}
```

```

mpz_t g, m, n, pi;
mpz_inits(g, m, n, pi, NULL);

int relations_found = 0;
clock_t start;
start = clock();
int tries = 0;
while(relations_found < pb_len + extra){

    // for 2
    mpz_set_ui(temp, 0);
    calc_poly(Q, temp);
    x1[0] = 0;
    if(mpz_divisible_ui_p(Q->qx, 2) == 0) x1
        [0] = 1;

    //others
    for(int i = 1; i<pb_len; i++){
        mpz_set_ui(pi, pb[i]);
        mpz_gcdext(g, m, n, Q->a, pi);
        if(mpz_cmp_ui(g, 1) != 0){
            fprintf(stderr, "ERROR: Number is
                too small for the current
                implementation of MPQS\n");
        };
        exit(1);
    }

    mpz_set_ui(temp, r[i]);
    mpz_sub(temp, temp, Q->b);
    mpz_mul(temp, temp, m);
    mpz_mod(temp, temp, pi);

```

```

x1[i] = mpz_get_ui(temp);

//calc_poly(Q, temp);
//assert(mpz_divisible_ui_p(Q->qx, pb[i]
    ]) != 0);

mpz_set_ui(temp, pb[i]);
mpz_sub_ui(temp, temp, r[i]);
mpz_sub(temp, temp, Q->b);
mpz_mul(temp, temp, m);
mpz_mod(temp, temp, pi);

x2[i] = mpz_get_ui(temp);

//calc_poly(Q, temp);
//assert(mpz_divisible_ui_p(Q->qx, pb[i]
    ]) != 0);

//realign sieving interval to [-s, s]
int k = (x1[i] + s)/pb[i];
x1[i] -= k * pb[i];
x1[i] += s;

k = (x2[i] + s)/pb[i];
x2[i] -= k * pb[i];
x2[i] += s;

//mpz_set_si(temp, -s);
//mpz_add_ui(temp, temp, x1[i]);
//calc_poly(Q, temp);
//assert(mpz_divisible_ui_p(Q->qx, pb[i]
    ]) != 0);

```

```

}

for(int i = 0; i < 2*s; i++){
    sinterval[i] = 0;
}

/*
// sieve for 2
while(x1[0] < 2*s){
    sinterval[x1[0]] += plogs[0];
    x1[0] += pb[0];
}
*/

// sieve other primes
for(int i = 30; i < pb_len; i++){

    while(x1[i] < 2*s){
        sinterval[x1[i]] += plogs[i];
        x1[i] += pb[i];
    }

    while(x2[i] < 2*s){
        sinterval[x2[i]] += plogs[i];
        x2[i] += pb[i];
    }
}

bool found;
bool update_time = false;
for(int i = 0; i < 2*s && relations_found <
    pb_len + extra; i++){

```

```

if(sinterval[i] > t){
    tries++;
    mpz_set_si(x, -s);
    mpz_add_ui(x, x, i);
    calc_poly(Q, x);

    if(!already_added(Q->zi, z,
        relations_found)){
        found = vectorize_mpps(Q->qx,
            v[relations_found],
            pb_len, pb);
        if(found){
            mpz_set(z[relations_found],
                Q->zi);
            mpz_set(d[relations_found],
                Q->d);
            relations_found++;
            update_time = true;
            found = false;
            if(!quiet){
                printf("\r");
                printf("%0.1f%%%u|u%.1f
                    %%%", (float)
                        relations_found/(
                            pb_len+extra)
                            *100, (float)
                                relations_found/
                                    tries*100);
                fflush(stdout);
            }
        }
    }
}
}
}

```

```

    }

    if(update_time && !quiet) printf("␣(~%.0fs␣
        left)␣␣␣␣␣␣␣␣␣␣", (double)(clock() -
        start)/CLOCKS_PER_SEC/
        relations_found*((pb_len+extra -
        relations_found)));
    get_next_poly(Q);
}

if(!quiet) printf("\n");

```

```

    mpz_clears(sqrt_N, temp, g, m, n, pi, x, NULL);
    free(x1);
    free(x2);
    free(r);
    free(sinterval);
    free(plogs);
    free_poly(Q);

    return v;
}

```

../c/mpqs/parallel_mpqs.h

```
#pragma once
#include <gmp.h>
#include "polynomial.h"
#include <sys/time.h>
#include <stdint.h>
```

```
struct sieve_arg_s {
    // used for sieving
    int* pb;
    int pb_len;
    int extra;
    int* r;
    float* plogs;
    int s;
    int t;
    int* relations_found;
    int** v;
    bool quiet;
    mpz_t* z;
    mpz_t* d;
};
```

```
poly_t Qinit;
```

```
// used to print progress and predicted time left
struct timeval begin;
uint_fast64_t* tries;
```

```
// used to constantly have a certain number of
    threads running
int thread_id;
bool* threads_running;
```

```
};
typedef struct sieve_arg_s sieve_arg_t;
```

```
bool already_added(mpz_t zi, mpz_t* z, int
    relations_found);
```

```
void* sieve_100_polys (void* args);
```

```
int** parallel_mpqs(mpz_t* z, mpz_t* d, mpz_t N,
    int pb_len, int* pb, int extra, int s, int delta,
    bool quiet);
```


../c/mpqs/parallel_mpqs.c

```
#include <gmp.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <math.h>
#include <time.h>
#include <pthread.h>
#include <sys/time.h>

#include "polynomial.h"
#include "common_mpqs.h"
#include "parallel_mpqs.h"
#include "../system.h"
#include "../tonellishanks.h"

pthread_mutex_t mutex;

void* sieve_100_polys (void* args){
    sieve_arg_t* arg = (sieve_arg_t*) args;

    poly_t Q = copy_poly(arg->Qinit);

    mpz_t temp, g, m, n, pi, x;
    mpz_inits(temp, g, m, n, pi, x, NULL);
    float* sinterval = malloc(2*arg->s*sizeof(float));
    int* x1 = malloc(arg->pb_len*sizeof(int));
    int* x2 = malloc(arg->pb_len*sizeof(int));
```

```
for(int i = 0; i<100 && *(arg->relations_found)
    < arg->pb_len + arg->extra; i++){
    get_next_poly(Q);

    //get sol for 2
    mpz_set_ui(temp, 0);
    calc_poly(Q, temp);
    x1[0] = 0;
    if(mpz_divisible_ui_p(Q->qx, 2) == 0) x1
        [0] = 1;

    //get sol for others
    for(int i = 1; i<arg->pb_len; i++){
        mpz_set_ui(pi, arg->pb[i]);
        mpz_gcdext(g, m, n, Q->a, pi);
        if(mpz_cmp_ui(g, 1) != 0){
            fprintf(stderr, "ERROR: Number is
                too small for the current
                implementation of MPQS\n");
            ;
            exit(1);
        }

        mpz_set_ui(temp, arg->r[i]);
        mpz_sub(temp, temp, Q->b);
        mpz_mul(temp, temp, m);
        mpz_mod(temp, temp, pi);

        x1[i] = mpz_get_ui(temp);
```

```

//calc_poly(Q, temp);
//assert(mpz_divisible_ui_p(Q->qx, arg
    ->pb[i]) != 0);

mpz_set_ui(temp, arg->pb[i]);
mpz_sub_ui(temp, temp, arg->r[i]);
mpz_sub(temp, temp, Q->b);
mpz_mul(temp, temp, m);
mpz_mod(temp, temp, pi);

x2[i] = mpz_get_ui(temp);

//calc_poly(Q, temp);
//assert(mpz_divisible_ui_p(Q->qx, arg
    ->pb[i]) != 0);

//realign sieving interval to [-s, s]
int k = (x1[i] + arg->s)/arg->pb[i];
x1[i] -= k * arg->pb[i];
x1[i] += arg->s;

k = (x2[i] + arg->s)/arg->pb[i];
x2[i] -= k * arg->pb[i];
x2[i] += arg->s;

//mpz_set_si(temp, -arg->s);
//mpz_add_ui(temp, temp, x1[i]);
//calc_poly(Q, temp);
//assert(mpz_divisible_ui_p(Q->qx, arg
    ->pb[i]) != 0);
}

//reset sieving_interval

```

```

for(int i = 0; i < 2*arg->s; i++){
    sinterval[i] = 0;
}

/*
// sieve for 2
while(x1[0] < 2*arg->s){
    sinterval[x1[0]] += arg->plogs[0];
    x1[0] += arg->pb[0];
}
*/

// sieve other primes
for(int i = 30; i < arg->pb_len; i++){
    while(x1[i] < 2*arg->s){
        sinterval[x1[i]] += arg->plogs[i];
        x1[i] += arg->pb[i];
    }
    while(x2[i] < 2*arg->s){
        sinterval[x2[i]] += arg->plogs[i];
        x2[i] += arg->pb[i];
    }
}

bool found;
bool update_time = false;
pthread_mutex_lock(&mutex);
for(int i = 0; i < 2*arg->s && *(arg->
    relations_found) < arg->pb_len +
    arg->extra; i++){
    if(sinterval[i] > arg->t){
        *(arg->tries) += 1;
        mpz_set_si(x, -arg->s);
    }
}

```



```

int** parallel_mpqs(mpz_t* z, mpz_t* d, mpz_t N,
    int pb_len, int* pb, int extra, int s, int delta,
    bool quiet){
    /** Gets pb_len+extra zis that are b-smooth,
        defined at:
        * Quadratic sieve factorisation algorithm
        * Bc. Ondřej Vladyka
        * Definition 1.11 (p.5)
        */

    //ceil(sqrt(n))
    mpz_t sqrt_N;
    mpz_init(sqrt_N);
    mpz_sqrt(sqrt_N, N);
    mpz_add_ui(sqrt_N, sqrt_N, 1);

    poly_t Q = init_poly(N, s);

    int** v = malloc((pb_len+extra)*sizeof(int*));
    for(int i = 0; i<pb_len+extra; i++){
        v[i] = malloc((pb_len+1)*sizeof(int*)); //
            +1 for -1
    }
    float* plogs = prime_logs_mpqs(pb, pb_len);

    int* r = malloc(pb_len*sizeof(int));
    int sol1, sol2;
    for(int i = 1; i < pb_len; i++){
        tonelli_shanks_ui(N, pb[i], &sol1, &sol2);
        r[i] = sol1;
    }
    int t = calculate_threshold_mpqs(sqrt_N, s, pb,

```

```

    pb_len, delta);

    sieve_arg_t* args = malloc(8*sizeof(sieve_arg_t)
    );
    pthread_t* threads = malloc(8*sizeof(pthread_t))
    ;
    bool* threads_running = malloc(8*sizeof(bool));
    for(int i = 0; i<8; i++){
        threads_running[i] = false;
    }

    int relations_found = 0;
    uint_fast64_t tries = 0;
    struct timeval begin;
    gettimeofday(&begin, 0);
    while(relations_found < pb_len + extra){
        for(int i = 0; i<8; i++){
            if(!threads_running[i]){
                args[i] = (sieve_arg_t) {
                    pb,
                    pb_len,
                    extra,
                    r,
                    plogs,
                    s,
                    t,
                    &relations_found,
                    v,
                    quiet,
                    z,
                    d,
                    Q,
                    begin,

```

```

        &tries,
        i,
        threads_running
    };
    threads_running[i] = true;
    pthread_create(threads+i, NULL,
        sieve_100_polys, args+i);
}
for(int i = 0; i<100; i++){
    get_next_poly(Q);
}
}
}
if(!quiet) printf("\n");

```

```

for(int i = 0; i<8; i++){
    pthread_join(threads[i], NULL);
}

free(threads);
free(args);
free(r);
free(plogs);
free(threads_running);
free_poly(Q);
mpz_clear(sqrt_N);

return v;
}

```