WriteUp Lattices

Hashimoto

1 Vectors

O texto explica aritmética básica de Vetores e pede para calcular:

$$3 * (2 * v - w) \cdot 2 * u$$

dado que $v = (2, 6, 3) \quad w = (1, 0, 0) \quad u = (7, 7, 2)$.

Esse problema é introdutório, dá para fazer "na mão":

$$\begin{gathered}
3 * (2 * v - w) \cdot 2 * u \\
3 * (2 * (2, 6, 3) - (1, 0, 0)) \cdot 2 * (7, 7, 2) \\
3 * ((4, 12, 6) - (1, 0, 0)) \cdot (14, 14, 4) \\
3 * (3, 12, 6) \cdot (14, 14, 4) \\
(9, 36, 18) \cdot (14, 14, 4) \\
9 \times 14 + 36 \times 14 + 18 \times 4 \\
126 + 504 + 72 \\
702
\end{gathered}$$

GG!

2 Size and Basis

O texto explica coisas sobre bases e tamanho/magnitude do vetor e pede para calcular o tamanho de $(4, 6, 2, 5)$

Novamente, dá para fazer na mão:

$$\begin{gathered}
\|(4, 6, 2, 5)\| \\
\sqrt{4 \times 4 + 6 \times 6 + 2 \times 2 + 5 \times 5} \\
\sqrt{16 + 36 + 4 + 25} \\
\sqrt{81}9
\end{gathered}$$

GG!

3 Gram Schmidt

O texto dá o algoritmo de *Gram-Schmidt*, mas não normaliza, só torna os vetores ortogonais entre si (mudei um pouco o formato):

`For` $(i := 0;\ i < n;\ i\ += 1)$ `do`
. $u_i := v_i$
. `For` $(j := 0;\ j < i;\ j\ += 1)$ `do`
. . $proj_{v_i \to u_j} := \frac{v_i \cdot u_j}{\|u_j\|} * u_j$
. . $u_i = u_i - proj_{v_i \to u_j}$
. `End For`
`End For`

O objetivo é achar o $u_4[1]$ (arredondado para 5 casa decimais) com essa entrada:

$$v_1 = (4, 1, 3, -1), \quad v_2 = (2, 1, -3, 4), \quad v_3 = (1, 0, -2, 7), \quad v_4 = (6, 2, 9, -5)$$

Dessa vez fazer na mão vai ser meio chato, então fiz uma pequena biblioteca em `zig` (está em ``src/vector.zig'').

Fiz um programa com a entrada hard-coded (`gran-schmidt.zig`), e essa é a saída:

```
[
    { 4.0e+00, 1.0e+00, 3.0e+00, -1.0e+00 },
    { 2.5925925925925926e+00, 1.1481481481481481e+00, -2.5555555555555554e+00,
3.851851851851852e+00 },
    { -7.229219143576826e-01, -1.0201511335012596e+00, 2.012594458438287e+00,
2.125944584382871e+00 },
    { -3.619189659458133e-01, 9.161073825503352e-01, 2.1488938603032537e-01,
1.1309967685806699e-01 }
]
```

Nosso u_4 está na última linha, $u_4[1]$ é

$$9.161073825503352e - 01$$
$$0.9161073825503352$$
$$0.91611$$

GG!

4 What's a Lattice

O texto explica o que é uma *Lattice*, um *Dominio Fundamental* de uma base e como se calcula o volume de um Domínio Fundamental dessa base. Então pede para calcular o volume do Domínio Fundamental dessa base:

$$v_1 = (6, 2, -3), \quad v_2 = (5, 1, 4), \quad v_3 = (2, 7, 1)$$

Esse dá para fazer na mão, mas é mais fácil escrever um programa que faz as contas para mim. Está aqui (em ``src/lattice.zig'') a saída:

```
-2.55e+02
```

Ou seja, -255.
NotGG : (
Ah, o "volume não pode ser negativo". Então... 255.
GG!

5 Gaussian Reduction

O texto cita os Problemas de procura do *Menor Vetor* e do *Vetor Mais Próximo* em uma Lattice.

A *Redução de Gauss em uma Lattice* é um algorítmo usado para resolver o Problema do Menor Vetor.

v_1, v_2
While ($true$) do
. If ($\|v_2\| < \|v_1\|$) do
. . $tmp := v_1$
. . $v_1 := v_2$
. . $v_2 := tmp$
. End If
. # Isso é uma divisão inteira
. $m := floor\left(\frac{v_1 \cdot v_2}{v_1 \cdot v_2}\right)$
. If ($m = 0$) do
. . return v_1, v_2
. End If
. $v_2 := v_2 - m * v_1$
End While

Temos que aplicar o algoritmo para

$$v_1 = (846835985, 9834798552), \quad v_2 = (87502093, 123094980)$$

A saída `gaussian-reduction.zig` é:

```
7410790865146821
```

GG!

6 Find the Lattice

O texto me dá um algoritmo de geração de chaves e encriptação/decriptação, a chave pública e a cifra, temos que achar a chave privada e decriptar a cifra para achar a mensagem original.

Algoritmo de geração de chaves é:

```
pub.x := primo(512)
limiteMax := pub.x / 2
limiteMin := pub.x / 4
```

```
priv.x := random(2, limiteMax)
While (true) do
.  priv.y := randim(limiteMin, limiteMax)
.  if (mdc(priv.x, priv.y)= 1) do
.  .  break While
.  end If
end While
temp := inverseMod(priv.x, pub.x)
pub.y = (temp * priv.y)%pub.x
return pub priv
```

Algoritmo de encriptação é:

```
m := mensagem
r := random(2, pub.x / 2)
cifra := (r * pub.y + m)%pub.x
return cifra
```

Algoritmo de decriptação é:

```
c := cifra
temp := (priv.x * c)%pub.x
mensagem := (temp*inverseMod(priv.x, priv.y))%priv.y
return mensagem
```

As informações dadas são:

$$pub.x =$$

$$\begin{gathered}7638232120454925879231554234011842347641017888219021175304217 3\\58715878636183252433454896490677496516149889316745664606749499\\24142016089801920392511529225 7\end{gathered}$$

$$pub.y =$$

$$\begin{gathered}21632689021945600938436935721701997075017877974979984634621295\\92239973581462651622978282637513865274199374452805292639586264\\7913174390295359264011090748 00\end{gathered}$$

$$cifra =$$

$$\begin{gathered}5605696495253720664142881956908624307570671858477482119657 43\\6163663663844731169035682344974286379049123733356009125671 92\\428031253275524116226726912348652 3\end{gathered}$$

Por ler o algoritmo de geração de chaves sabemos que:

- `pub.x` é um primo de 512 bits.

- $2 \leq \texttt{priv.x} < \frac{\texttt{pub.x}}{2}$

- $\frac{\texttt{pub.x}}{4} \leq \texttt{priv.y} < \frac{\texttt{pub.x}}{2}$

- `priv.x`, `priv.y` são primos entre si

- $\texttt{priv.x} * temp = 1 \ mod \texttt{pub.x}$

- $\texttt{pub.y} = \texttt{priv.y} * temp) \ mod \texttt{pub.x}$

+ $\texttt{priv.x} * \texttt{pub.y} = \texttt{priv.y} \ mod \texttt{pub.x}$

Por ler o algoritmo de encriptação sabemos que:

- $2 \leq r < \frac{\texttt{pub.x}}{2}$

- $\texttt{cifra} = r * \texttt{pub.y} + \texttt{m} \ mod \texttt{pub.x}$

Por ler o algoritmo de decriptação sabemos que:

- $temp = \texttt{priv.x} * \texttt{cifra} \ mod \texttt{pub.x}$

- $\texttt{priv.x} * inv = 1 \ mod \texttt{priv.y}$

- $\texttt{m} = temp * inv \ mod \texttt{priv.y}$

+ $\texttt{priv.x} * \texttt{m} = temp \ mod \texttt{priv.y}$

No fim, *graças a ajudas univesitárias*, só precisamos disso

- $\texttt{priv.x} * \texttt{pub.y} = \texttt{priv.y} \ mod \texttt{pub.x}$

$$\begin{gathered}
\texttt{priv.x} * \texttt{pub.y} = \texttt{priv.y} \ mod \texttt{pub.x} \\
\texttt{priv.x} * \texttt{pub.y} - k * \texttt{pub.x} = \texttt{priv.y} \\
\texttt{priv.x}(\texttt{pub.y}, 1) - k(\texttt{pub.x}, 1) = (\texttt{priv.y}, \texttt{priv.x} - k)
\end{gathered}$$

Se usarmos $\{(\texttt{pub.y}, 1), (\texttt{pub.x}, 1)\}$ como entrada para Redução de Gauss, vamos saber *magicamente* que um dos retornos $(x_1, y_1), (x_2, y_2)$ é o `priv.y`.

Com o nosso `priv.y` (tentativa e erro com 4 possibilidades) podemos calcular o `priv.x`:

$$\texttt{priv.x} = \texttt{priv.y} * \texttt{inverseMod(pub.y, pub.x)} \ mod \texttt{pub.x}$$

Isso foi implementado em `where.py` *(infelizmente zig não suporta números tão grande.*

```
0 b'crypto{Gauss_lattice_attack!}'
1 b"u\xc9F\xfa\x91'\xb6[Y\xf6\xbb>\x1e\xb1\r\x8eN\xba\xe1\xf7\x90}mu\\u\xb3
\xb6\xa9\x00\x15t"
2 b'\x13C#d\x1cd@\xe60\xfdP\x19\xe8\x1d\xe8\x8f\xf0\x0br\xefV\xb3b\x11\xe6\x02
\xe4\xf3*3\xb2\xc9'
3 b'\x00'
```

Está lá na primeira tentativa `crypto{Gauss_lattice_attack!}`

GG!