Description: A tool that helps MAT302 students calculate and visualize cryptography concept.

Features:

1. Greatest common divisor using Euclid's algorithm
2. Bezout's identity
3. Multiplicative inverse with Euclid's algorithm
4. Fast powering
5. Carmichael function
6. Primary Test witness and non-witness
    a. Fermat
    b. Euler (Quadratic residue)
    c. Miller Rabin
7. Cryptography algorithm and how to crack
    a. DLP crack by Shank's baby giant step
    b. RSA crack by Pollard's Rho algorithm
8. Finite Field how to generate finite field of p^d elements
    a. Polynomial modulo
    b. Polynomial GCD
    c. Finite Field generation
9. Elliptic curve
    a. Determinant
    b. Finding points
    c. Point arithmetic
10. Digital Signature
    a. RSA
    b. Elgamal
    c. Elliptic Curve

Chose of Technology:

Backend: Python Flask for its lightweight and user friendly. First, we don't need to interact with the database or security. Second, we use the REST API which Flask supports.

Frontend: TypeScript

1. Greatest common divisor with Euclid for demonstration
   Sanitize input only for integers and convert to positive integer
   Request[GET]: URL: /gcd?a=2024&b=68
   Response: {gcd: 4, steps: [{dividend: 2024, divisor: 68, quotient: 29, remainder: 52}, {dividend: 68, divisor: 52, quotient: 1, remainder: 16},...]}
2. Bezout with steps
   Sanitize input only for integers and convert to positive integer return immediately if no integer solution according to Bezout
   Request[GET]: http://localhost:5555/bezout?a=1025&b=2046&c=88
   Response:

```
{
   "k": 88,
   "raw": {
       "0": {
           "1": -4,
           "4": 1
       },
       "1": {
           "4": -255,
           "1021": 1
       },
       "4": {
           "1021": -1,
           "1025": 1
       },
       "1021": {
           "1025": -1,
           "2046": 1
       }
   },
   "refined": {
       "0": {
           "1025": 2046,
           "2046": -1025
       },
       "1": {
           "1025": -511,
           "2046": 256
       },
       "4": {
           "1025": 2,
           "2046": -1
       },
       "1021": {
           "1025": -1,
           "2046": 1
       }
   }
}
```


3. Multiplicative inverse with steps

Sanitize input only for integers ‘a’ and ‘p’ that are coprime. Show step by step of the answer using Bezout.
Request[GET]: http://localhost:5555/mul_inverse?a=1025&p=2046
Response:


```
{
   "inverse": 1535,
   "raw": {
       "0": {
           "1": -4,
           "4": 1
       },
       "1": {
           "4": -255,
           "1021": 1
       },
       "4": {
           "1021": -1,
           "1025": 1
       },
       "1021": {
           "1025": -1,
           "2046": 1
       }
   },
   "refined": {
       "0": {
           "1025": 2046,
           "2046": -1025
       },
       "1": {
           "1025": -511,
           "2046": 256
       },
       "4": {
           "1025": 2,
           "2046": -1
       },
       "1021": {
           "1025": -1,
           "2046": 1
       }
   }
```

```
}
```

4. Fast Powering
   Sanitize input only for integers ‘a’ ‘n’ and ‘p’. Using pre-built fast powering in Python
   Request[GET]: http://localhost:5555/fast_power?a=1025&n=55555&p=2046
   Response:

```
{
   "result": 1055
}
```

5. Carmichael function
   Sanitize input only integer ‘n’ is accepted.
   Request[GET]: http://localhost:5555/carmichael_func?n=77
   Response:

```
{
   "result": 30
}
```

6. Primary Test
   Sanitize input for ‘a’ as potential witness, ‘n’ the number that will be test, ‘test’ the test that will be applied
   Return whether n is ‘test’ pseudoprime with respect to a
   Request[GET]: http://localhost:5555/primary?test=miller_rabin&a=3&n=252605
   Response:

```
{
   "result": false
}
```

   There are 3 tests in total which are fermat, euler, miller_rabin
7. Cryptography crack
   a. DLP crack by Shank
      Sanitize input for ‘g’, ‘y’ and ‘p’. ‘p’ must be prime. Return ‘n’ such that g^n ≡ y mod p
      Request: http://localhost:5555/dlp_crack?g=8&y=6&p=11
      Response:

```
{
   "i": 3,
   "j": 0,
   "x": 3
}
```

   b. RSA crack by Pollard’s Rho algorithm with steps
      Sanitize input for ‘n’, ‘poly’, ‘x0’. ‘poly’ is a map that maps degree to coefficients to represent polynomial function. ‘n’ is the number to factorize. ‘x0’ is the initial number to start with. ‘limit’ is optional and default to 100

Request[POST]: http://localhost:5555/rsa_crack_rho
Body:

```
{
   "n": 4087,
   "poly": {
       "0": 1,
       "1": 1,
       "2": 1
   },
   "x0": 2
}
```

Response:


```
{
   "result": 61,
   "steps": [
       {
           "gcd": null,
           "k": null,
           "x_k": null,
           "x_m": 2
       },
       {
           "gcd": 1,
           "k": 0,
           "x_k": 2,
           "x_m": 7
       },
       {
           "gcd": 1,
           "k": 1,
           "x_k": 7,
           "x_m": 57
       },
       {
           "gcd": 1,
           "k": 1,
           "x_k": 7,
           "x_m": 3307
       },
       {
           "gcd": 1,
           "k": 2,
```

```
            "x_k": 3307,
            "x_m": 2745
        },
        {
            "gcd": 1,
            "k": 2,
            "x_k": 3307,
            "x_m": 1343
        },
        {
            "gcd": 1,
            "k": 2,
            "x_k": 3307,
            "x_m": 2626
        },
        {
            "gcd": 61,
            "k": 2,
            "x_k": 3307,
            "x_m": 3734
        }
    ]
}
```


8. Finite field
    a. Polynomial modulo
    Sanitize input for ‘dividend’ and ‘divisor’ must be a map that maps degree to the coefficient.
    Request[POST]: http://localhost:5555/poly_modulo
    Body:


```
{
    "dividend":
        {"0": 1, "1": 0, "2": 0, "3": 0, "4": 0, "5": 0, "6": 0, "7": 1},
    "divisor":
        {"0": 0, "1": 1, "2": 1, "3": 0, "4": 1},
    "p": 2
}
```


    Response:


```
{
    "0": 1,
    "1": 1,
    "2": 0,
    "3": 1,
```

```
    "4": 0,
    "5": 0,
    "6": 0,
    "7": 0
}
```


b. Polynomial gcd
Sanitize input for ‘poly1’ and ‘poly2’ must be a map that maps degree to the coefficient
Request[POST]: http://localhost:5555/poly_gcd
Body:


```
{
    "poly1":
        {"0": 0, "1": 3, "2": 0, "3": 0, "4": 1},
    "poly2":
        {"0": 3, "1": 0, "2": 1},
    "p": 5
}
```


Response:


```
{
    "0": 2,
    "1": 0,
    "2": 0
}
```


c. Finite Field generation
Sanitize input for ‘p’ and ‘d’ as ‘p’ is prime and generate a field with p^d elements
Request[GET]: http://localhost:5555/generate_field?p=2&d=2
Response:


```
{
    "add": [
        [
            {
                "0": 0,
                "1": 0
            },
            {
                "0": 1,
                "1": 0
            },
            {
                "0": 0,
                "1": 1
            },
```

```
        {
            "0": 1,
            "1": 1
        }
    ],
    [
        {
            "0": 1,
            "1": 0
        },
        {
            "0": 0,
            "1": 0
        },
        {
            "0": 1,
            "1": 1
        },
        {
            "0": 0,
            "1": 1
        }
    ],
    [
        {
            "0": 0,
            "1": 1
        },
        {
            "0": 1,
            "1": 1
        },
        {
            "0": 0,
            "1": 0
        },
        {
            "0": 1,
            "1": 0
        }
    ],
    [
```

```
            {
                "0": 1,
                "1": 1
            },
            {
                "0": 0,
                "1": 1
            },
            {
                "0": 1,
                "1": 0
            },
            {
                "0": 0,
                "1": 0
            }
        ]
    ],
    "elements": {
        "0": {
            "0": 0,
            "1": 0
        },
        "1": {
            "0": 1,
            "1": 0
        },
        "2": {
            "0": 0,
            "1": 1
        },
        "3": {
            "0": 1,
            "1": 1
        }
    },
    "irreducible_poly": {
        "0": 1,
        "1": 1,
        "2": 1
    },
    "prod": [
```

```
    [
        {
            "0": 0,
            "1": 0,
            "2": 0
        },
        {
            "0": 0,
            "1": 0,
            "2": 0
        },
        {
            "0": 0,
            "1": 0,
            "2": 0
        },
        {
            "0": 0,
            "1": 0,
            "2": 0
        }
    ],
    [
        {
            "0": 0,
            "1": 0,
            "2": 0
        },
        {
            "0": 1,
            "1": 0,
            "2": 0
        },
        {
            "0": 0,
            "1": 1,
            "2": 0
        },
        {
            "0": 1,
            "1": 1,
            "2": 0
```

```
        }
    ],
    [
        {
            "0": 0,
            "1": 0,
            "2": 0
        },
        {
            "0": 0,
            "1": 1,
            "2": 0
        },
        {
            "0": 1,
            "1": 1,
            "2": 0
        },
        {
            "0": 1,
            "1": 0,
            "2": 0
        }
    ],
    [
        {
            "0": 0,
            "1": 0,
            "2": 0
        },
        {
            "0": 1,
            "1": 1,
            "2": 0
        },
        {
            "0": 1,
            "1": 0,
            "2": 0
        },
        {
            "0": 0,
```

```
            "1": 1,
            "2": 0
        }
    ]
]}
```


9. Elliptic Curve
    a. Determinant
    Sanitize input for ‘a’, ‘b’, ‘p’ must be integer. ‘p’ must be prime. Return the determinant of the elliptic curve
    Request: http://localhost:5555/elliptic_curve/determinant?a=21&b=1&p=31
    Response:


```
{
  "determinant": 26,
  "message": "The elliptic curve is non-singular."
}
```


    b. Points
    Sanitize input for ‘a’, ‘b’, ‘p’ must be integer. ‘p’ must be prime. Return all the points on the elliptic curve
    Request: http://localhost:5555/elliptic_curve/points?a=2&b=4&p=5
    Response:


```
    {
      "points": [
        [
          Infinity,
          Infinity
        ],
        [
          0,
          2
        ],
        [
          0,
          3
        ],
        [
          2,
          1
        ],
        [
          2,
          4
        ],
```

```
            [
                4,
                1
            ],
            [
                4,
                4
            ]
        ]
    }
```


c. Point arithmetic

I. Addition

Sanitize ‘a’, ‘b’, ‘p’, ‘x1’, ‘y1’, ‘x2’, ‘y2’. ‘p’ must be prime. ‘x1’, ‘y1’ and ‘x2’, ‘y2’ must both be on the curve. Return the result after adding ‘x1’, ‘y1’ to ‘x2’, ‘y2’

Request:
http://localhost:5555/elliptic_curve/points_add?a=3&b=8&p=13&x1=2&y1=3&x2=2&y2=3

Response:


```
{
  "result": [
      12,
      11
  ]
}
```


Ii. Multiplication

Sanitize ‘a’, ‘b’, ‘p’, ‘x’, ‘y’, ‘k’. ‘p’ must be prime. ‘x’, ‘y’ must be on the curve. Return the result after multiply ‘x’, ‘y’ to ‘k’

Request:http://localhost:5555/elliptic_curve/points_mul?a=3&b=8&p=13&x=2&y=3&k=8

Response:


```
{
  "result": [
      2,
      10
  ]
}
```


Iii. Order

Sanitize ‘a’, ‘b’, ‘p’, ‘x’, ‘y’. ‘p’ must be prime. ‘x’, ‘y’ must be on the curve. Return the order of ‘x’, ‘y’

Request: http://localhost:5555/elliptic_curve/point_order?a=3&b=8&p=13&x=2&y=3

Response:


```
{
  "order": 9
```

```
}
```

10. Digital Signature