BLOCKCHAIN APPLICATION DEVELOPMENT

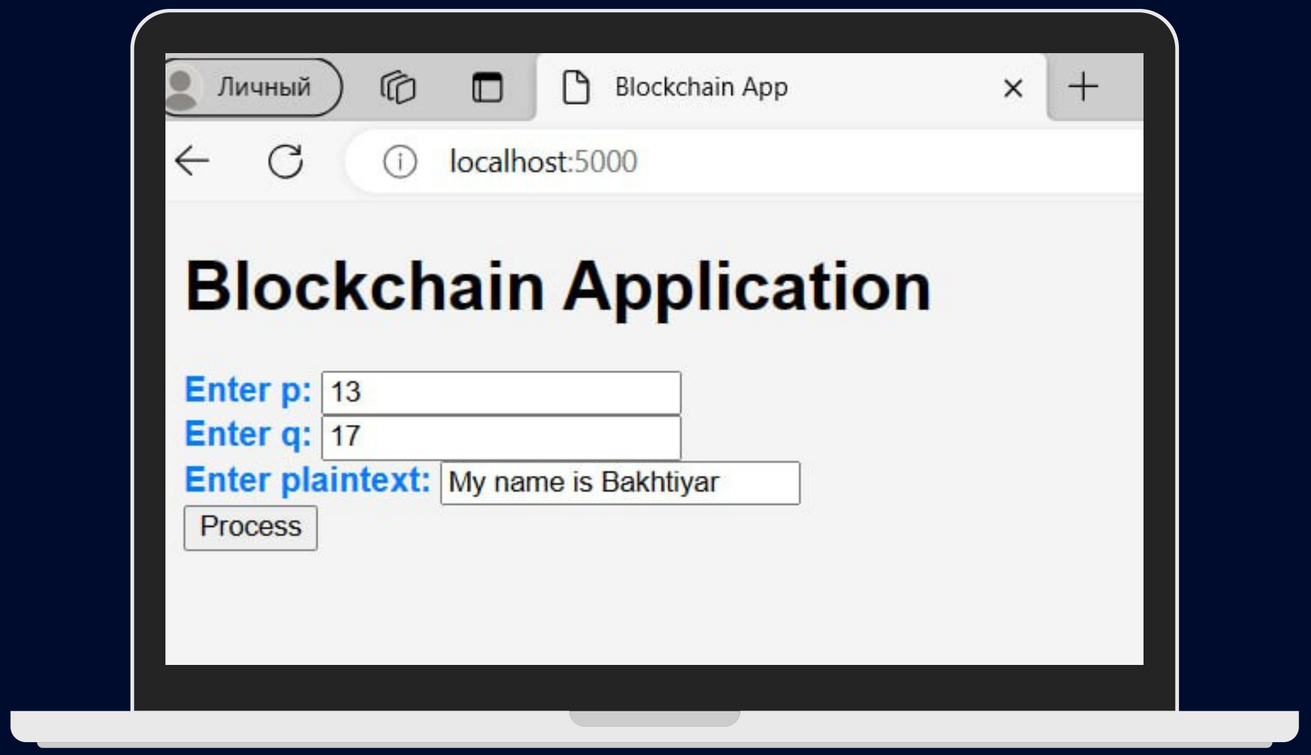
Les Baktyiar, Avgustkhan Sanzhar

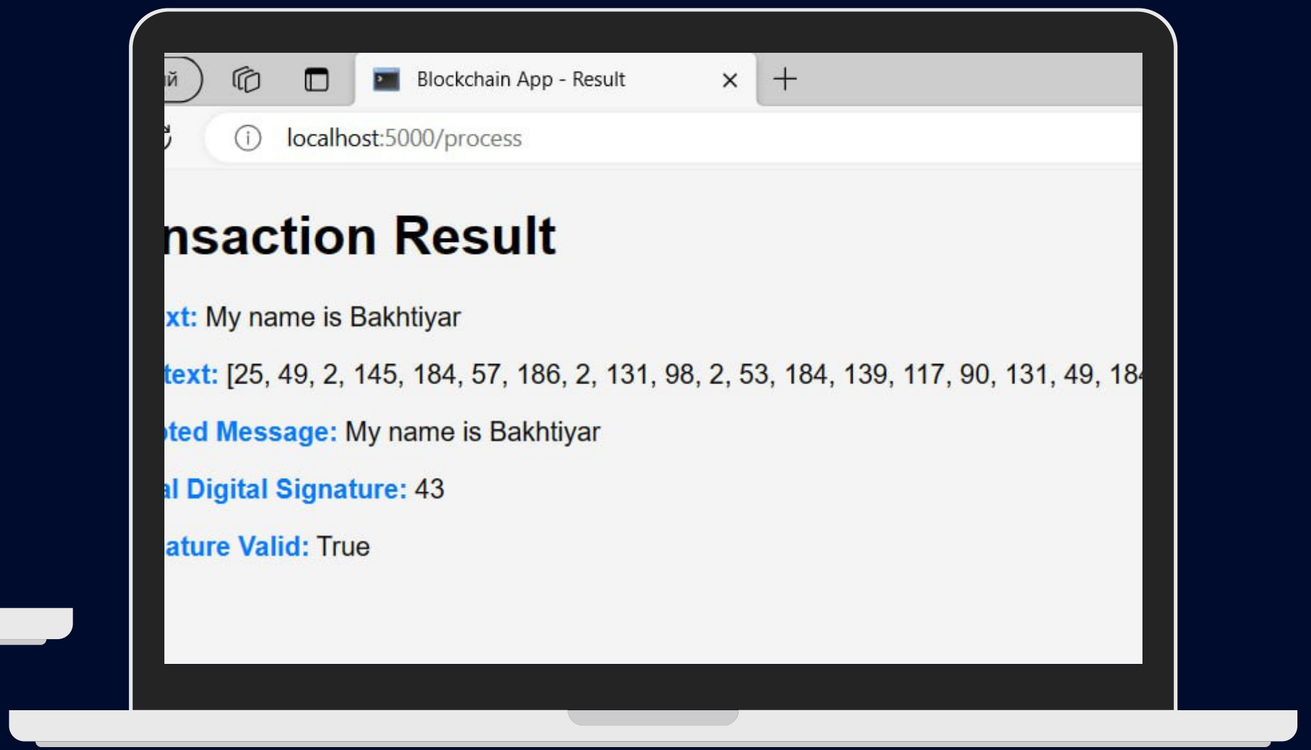
User Interface

```
 * Serving Flask app 'main'
 * Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
 * Running on http://127.0.0.1:5000
Press CTRL+C to quit
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 588-743-582
127.0.0.1 - - [24/Dec/2023 01:25:18] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [24/Dec/2023 01:25:19] "GET /favicon.ico HTTP/1.1" 404 -
127.0.0.1 - - [24/Dec/2023 01:25:38] "POST /process HTTP/1.1" 200 -
```

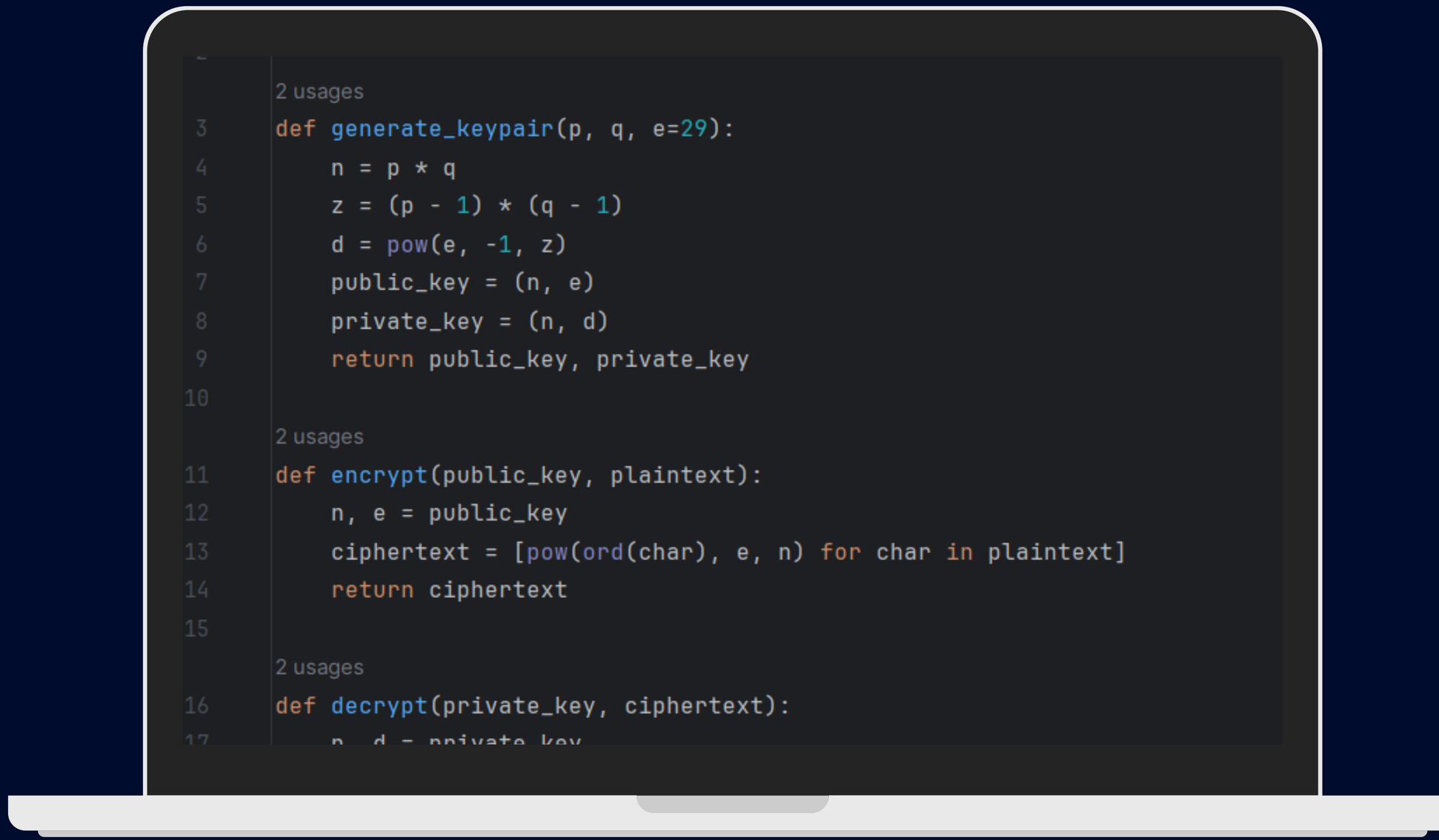
Created a web-based user interface using Flask for user interaction in python. in short, the user is asked for input data, two prime numbers (according to the rules of the RSA algorithm) and the text that we encrypt. Then, the text is encrypted according to the algorithm, to its ASCII value.

Interface


Личный
Blockchain App
localhost:5000
Blockchain Application
Enter p: 13
Enter q: 17
Enter plaintext: My name is Bakhtiyar
Process


Blockchain App - Result
localhost:5000/process
nsaction Result
xt: My name is Bakhtiyar
text: [25, 49, 2, 145, 184, 57, 186, 2, 131, 98, 2, 53, 184, 139, 117, 90, 131, 49, 184
ted Message: My name is Bakhtiyar
l Digital Signature: 43
ature Valid: True

Encryption Integration



```
2 usages
def generate_keypair(p, q, e=29):
    n = p * q
    z = (p - 1) * (q - 1)
    d = pow(e, -1, z)
    public_key = (n, e)
    private_key = (n, d)
    return public_key, private_key

2 usages
def encrypt(public_key, plaintext):
    n, e = public_key
    ciphertext = [pow(ord(char), e, n) for char in plaintext]
    return ciphertext

2 usages
def decrypt(private_key, ciphertext):
    n, d = private_key
```

Key Pair Generation:

The generate_keypair function generates a public-private key pair using the provided prime numbers p and q.
The public key (public_key) consists of n (product of p and q) and e (a fixed value, 29 in your case). The private key (private_key) consists of n and d (calculated using the modular inverse of e).

Encrypting a Message:

The encrypt function takes the public key (public_key) and a plaintext message (plaintext) as input.
It converts each character in the plaintext message to its corresponding ASCII value and raises it to the power of e modulo n.
The resulting ciphertext is a list of integers.

Decrypting a Message:

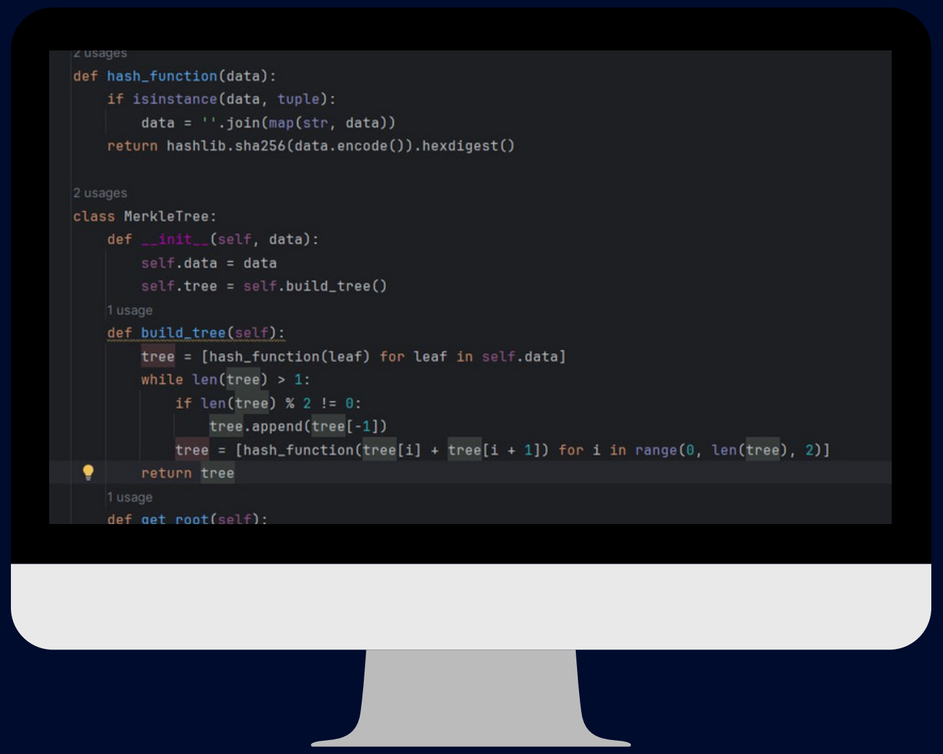
The decrypt function takes the private key (private_key) and the ciphertext as input.
It raises each integer in the ciphertext to the power of d modulo n.
The decrypted values are converted back to characters, and the original plaintext message is reconstructed.

```python
def simple_hash(message):
    sha256 = hashlib.sha256()
    sha256.update(message.encode('utf-8'))
    return int(sha256.hexdigest(), 16)

2 usages
def sign_message(private_key, message):
    n, d = private_key
    hash_value = simple_hash(message)
    signature = pow(hash_value, d, n)
    return signature

2 usages
def verify_signature(public_key, message, received_signature):
    n, e = public_key
    computed_hash = pow(received_signature, e, n)
    return computed_hash == simple_hash(message) % n
```

The sign_message function takes the private key (private_key) and a message (message) as input. It calculates a hash value (hash_value) of the message using the simple_hash function. The hash value is then signed using the private key, resulting in a digital signature (signature).

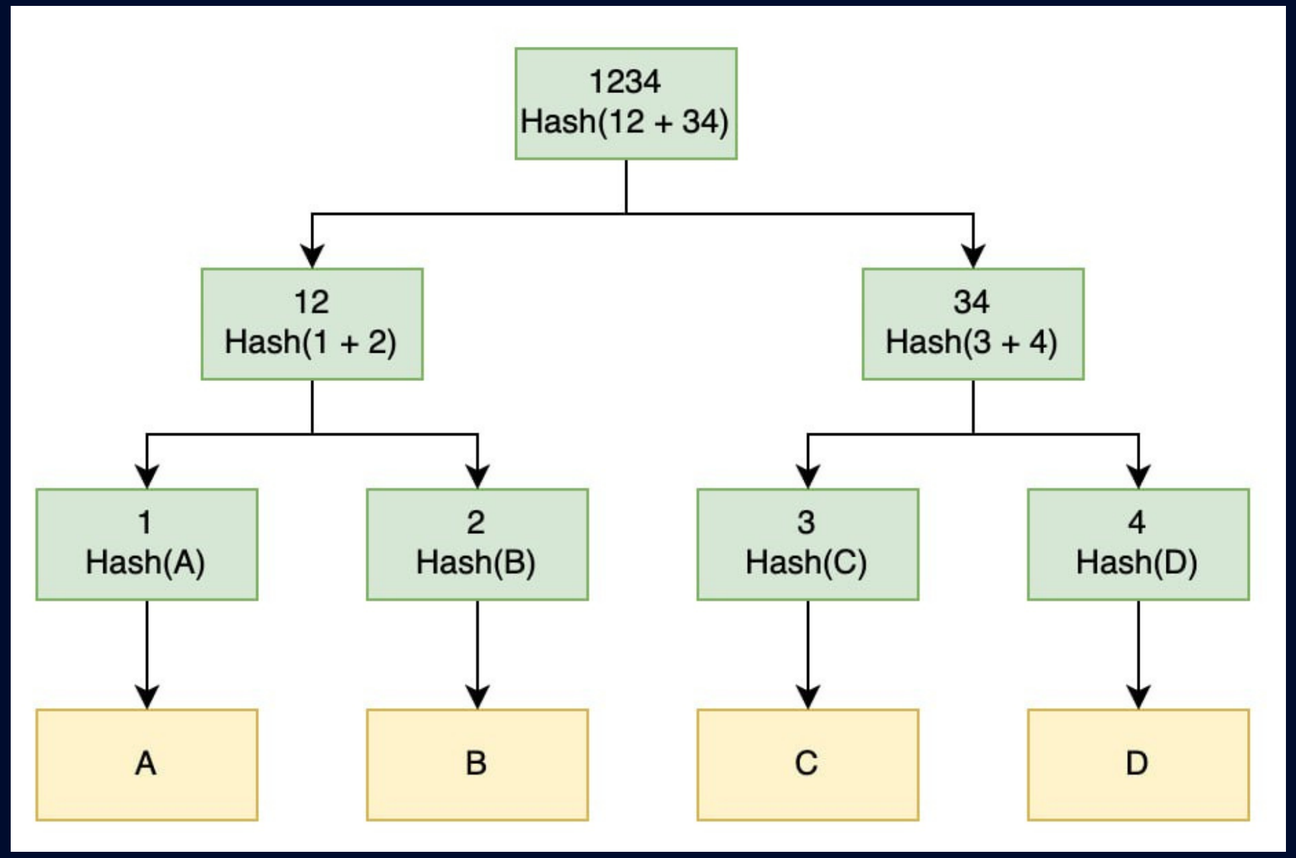
The verify_signature function takes the public key (public_key), a message (message), and a received signature (received_signature) as input. It calculates a hash value of the message using the simple_hash function. The received signature is decrypted using the public key, resulting in a computed hash value (computed_hash).

MERKLE TREE



```
2 usages
def hash_function(data):
    if isinstance(data, tuple):
        data = ''.join(map(str, data))
    return hashlib.sha256(data.encode()).hexdigest()

2 usages
class MerkleTree:
    def __init__(self, data):
        self.data = data
        self.tree = self.build_tree()
    1 usage
    def build_tree(self):
        tree = [hash_function(leaf) for leaf in self.data]
        while len(tree) > 1:
            if len(tree) % 2 != 0:
                tree.append(tree[-1])
            tree = [hash_function(tree[i] + tree[i + 1]) for i in range(0, len(tree), 2)]
        return tree
    1 usage
    def get_root(self):
```

A Merkle tree, also known as a binary hash tree, is a data structure used in cryptography and computer science to efficiently verify the integrity and consistency of a large set of data. It is particularly prevalent in blockchain technology.



Pairs of leaf nodes are then hashed together to create intermediate nodes. If the number of nodes is odd, the last node is duplicated to make it even.