قسمت اول: تولید آدرس

1. تولید آدرس عادی

برای این کار ابتدا یک عدد 256 بیتی رندوم را به عنوان Private Key انتخاب می‌کنیم. پس از اعمال ECDSA بر روی آن، مقدار Public Key بدست می‌آید. سپس، پس از یک بار اعمال SHA256 و یک بار اعمال RIPEMD160 پس از آن، اگر نتیجه را به فرمت WIF تبدیل کنیم، آدرس کیف پول مورد نظر بدست می‌آید. در این بخش از کلاس زیر (فایل part1_q1.py) استفاده شده است:

```
class Wallet:
    class Network(enum.Enum):
        MAINNET = 0
        TESTNET = 1

    def __init__(self, network: Network = Network.TESTNET):
        self.network = network
        self._private_key = b""
        self._public_key = b""
        self._bitcoin_address = ""

    def generate(self) -> None:
        self._private_key = secrets.token_bytes(32)
        self._generate_public_key()
        self._generate_bitcoin_address()

    def generate_from_wif(self, private_key_wif: str) -> None:
        self._private_key = self._from_wif(private_key_wif)
        self._generate_public_key()
        self._generate_bitcoin_address()

    def _get_network_byte(self, is_private: bool = True) -> bytes:
        if is_private:
            if self.network == Wallet.Network.MAINNET:
                return b"\x80"
            if self.network == Wallet.Network.TESTNET:
                return b"\xef"
            raise ValueError("Invalid network")
        elif self.network == Wallet.Network.MAINNET:
            return b"\x00"
        elif self.network == Wallet.Network.TESTNET:
            return b"\x6f"
        else:
            raise ValueError("Invalid network")

    def _generate_public_key(self) -> None:
        public_key = ecdsa.SigningKey.from_string(
            self._private_key, curve=ecdsa.SECP256k1
        ).verifying_key

        if public_key is None:
            raise ValueError("Invalid public key")
        self._public_key = (b"\x04" + public_key.to_string())  # 0x04 is the prefix for uncompressed public keys

    def _generate_bitcoin_address(self) -> None:
        sha256 = hashlib.sha256(self._public_key).digest()
        ripemd160 = hashlib.new("ripemd160")
        ripemd160.update(sha256)

        self._bitcoin_address = self._to_wif(ripemd160.digest(), is_private=False)

    def _to_wif(self, key: bytes, is_private: bool = True) -> str:
        network_byte = self._get_network_byte(is_private)
        key_with_network_byte = network_byte + key
        sha256_1 = hashlib.sha256(key_with_network_byte).digest()
        sha256_2 = hashlib.sha256(sha256_1).digest()
        checksum = sha256_2[:4]
        binary_key = key_with_network_byte + checksum
        wif = base58.b58encode(binary_key).decode("utf-8")
        return wif

    def _from_wif(self, wif: str) -> bytes:
        binary_key = base58.b58decode(wif)
        key = binary_key[:-4]
        checksum = binary_key[-4:]
        sha256_1 = hashlib.sha256(key).digest()
        sha256_2 = hashlib.sha256(sha256_1).digest()
        if checksum != sha256_2[:4]:
            raise ValueError("Invalid WIF")
        network_byte = key[0:1]
        if network_byte != self._get_network_byte():
            raise ValueError("Invalid WIF")
        return key[1:]
```

تفاوت آدرس در شبکه اصلی و در شبکه تست، در Network Byte آن‌ها است. همانطور که در کد مشاهده می‌شود، برای شبکه اصلی مقدار 0x00 به ابتدای آدرس اضافه می‌شود و برای شبکه تست، مقدار 0x6F به آن اضافه می‌شود. در نتیجه زمانی که آدرس را به Base58 تبدیل می‌کنیم، در شبکه اصلی، اولین کاراکتر آدرس همواره 1 خواهد بود و در شبکه تست، این کاراکتر می‌تواند m و یا n باشد.

خروجی این کد به صورت زیر است:

```
Address:           mqvxfUnftcXwrX25mBC3hPpWQdFeZwccBC
Private key (WIF): 93QbFVvPXL4xWStG8vVZnw1qtLmZj1tV5jTMX3tJjBN5716ypkZ
Private key:       eff210fd94558066e82d4907042151385197219 69de28d7640d4d91b45ad2a00
Public key:        0469f52ddd3bc2cc0e29f15bafd853d03b5a18e7d0368ff6b18ab25e826030a8fe37ed546065e242d6b937b3d8c6
da5147c56716c69c19364cb2006e2b6f05002e
```

موارد خواسته شده به صورت زیر هستند:

Address: mqvxfUnftcXwrX25mBC3hPpWQdFeZwccBC

Private Key (WIF): 93QbFVvPXL4xWStG8vVZnw1qtLmZj1tV5jTMX3tJjBN5716ypkZ

2. تولید آدرس ویژه

در این حالت باید به تعداد زیادی Private Key مختلف را آزمایش کنیم تا در نهایت آدرس تولید شده، با پیشوند مورد نظر ما آغاز شود. در این بخش من پیشوند pas (3 حرف ابتدای اسمم) را انتخاب کردم. با توجه به اینکه می‌خواهیم 3 حرف ابتدایی آدرس ثابت باشد، به طور متوسط باید حدود $58^3 = 195{,}112$ کلید را آزمایش کنیم تا به نتیجه برسیم. برای این کار یک کلاس جدید ایجاد می‌کنیم (فایل part1_q2.py) که از کلاس قبلی ارث می‌برد:

```python
class VanityWallet(Wallet):
    def __init__(self, prefix: str, network: Wallet.Network = Wallet.Network.TESTNET):
        super().__init__(network)
        self._prefix = prefix
        self._number_of_tries = 0

    # some properties

    def generate(self) -> None:
        while True:
            super().generate()
            self._number_of_tries += 1
            if self.bitcoin_address[1:].startswith(self.prefix):
                break
```

خروجی این کد به صورت زیر است:

```
Address:           mpassrNwg2URHjHkpy2jNR46EfLEjsc2nV
Private key (WIF): 93Ru7CAHQKPcr3bXy837Uazm3fgPvm1tLeydtaAiwPSGMubMKdG
Private key:       f2ea4eb520d2364f9e3ca5c4e5c419c6954de78cd2eac1155045c49739da2848
Public key:        0455fc877f483e5488794e53f6decf320650c92933d9a1247bc6441019e1c15502476c02459d536769fb0fc8f853
e5a888629d56362536ac2a683fd9d6738b7a38
Prefix:            pas
Number of tries:   26502
```

موارد خواسته شده به صورت زیر هستند:

Address: mpassrNwg2URHjHkpy2jNR46EfLEjsc2nV

Private Key (WIF): 93Ru7CAHQKPcr3bXy837Uazm3fgPvm1tLeydtaAiwPSGMubMKdG

قسمت دوم: انجام تراکنش

ابتدا باید از طریق Faucetهای ارائه شده، مقداری پول به آدرس‌مان انتقال دهیم. در این بخش، برای ساخت آدرس از کد زیر استفاده شده و برای مقدار private_key_wif، یکی از کلیدهای ساخته شده به فرم WIF توسط سوال اول بخش اول استفاده شده است:

```
if __name__ == "__main__":
    private_key_wif = "92Zh9ENA7DeNBr3FXa1QMLi4igPAzUKy44TEPMW7rogBtGz4CaR"
    bitcoin.SelectParams("testnet")
    private_key = bitcoin.wallet.CBitcoinSecret(private_key_wif)
    public_key = private_key.pub
    address = bitcoin.wallet.P2PKHBitcoinAddress.from_pubkey(public_key)
    print(f"Private key: {private_key}")
    print(f"Public key: {public_key.hex()}")
    print(f"Address: {address}")
```

خروجی به صورت زیر است:

```
Private key: 92Zh9ENA7DeNBr3FXa1QMLi4igPAzUKy44TEPMW7rogBtGz4CaR
Public key: 049e9ec6c3ba76c4b4aa6f116d1e6d2f2f0f1601c9ef4fe3209ba2df51744d46cb5aca4409fd8c7bf62256be95d125a21eeed8a949bd4eda06d8e48f2ea4954bb1
Address: mu2XGdXpAM8fkFMWksFmMqCYP6oUXTVxRs
```

مقادیر خواسته شده به صورت زیر هستند:

Private Key (WIF): 92Zh9ENA7DeNBr3FXa1QMLi4igPAzUKy44TEPMW7rogBtGz4CaR

Address: mu2XGdXpAM8fkFMWksFmMqCYP6oUXTVxRs

از Faucet اول مقدار 0.0137566 بیت‌کوین به آدرس من انتقال داده شد. مشخصات این تراکنش در این لینک و این لینک قابل مشاهده است. هش تراکنش نیز به صورت زیر است:

12a4ccff3ed9bec0715fcf678b914397e1e7b3fcea239fe551f269c89810f909

تمامی تراکنش‌های این بخش به طور خلاصه از این لینک قابل مشاهده هستند.

برای ایجاد تراکنش‌های این بخش، ابتدا تعدادی کلاس نوشته شد که در فایل transaction.py قابل دسترس هستند:

```
def address_to_pub_key_hash160(address: str) -> bytes:
    pub_key_hash = base58.b58decode_check(address)[1:]
    return pub_key_hash


def P2PKH_script_pub_key(pub_key_hash: bytes) -> CScript:
    return CScript([OP_DUP, OP_HASH160, pub_key_hash, OP_EQUALVERIFY, OP_CHECKSIG])  # type: ignore


class Destination:
    def __init__(
        self, address: str, amount: float, script_pub_key: CScript | None = None
    ):
        self._address = address
        self._script_pub_key = (
            P2PKH_script_pub_key(address_to_pub_key_hash160(address))
            if script_pub_key is None
            else script_pub_key
        )
        self._amount = amount

    # some properties

    @property
    def TxOut(self) -> CMutableTxOut:
        return CMutableTxOut(int(self.amount * COIN), self.script_pub_key)


class UnspentTransactionOutput:
    def __init__(
        self,
        tx_id: str,
        index: int,
        script_pub_key: CScript,
        custom_sig: CScript | None = None,
    ):
        self._tx_id = tx_id
        self._index = index
        self._script_pub_key = script_pub_key
        self._custom_sig = custom_sig

    # some properties

    @property
    def TxIn(self) -> CMutableTxIn:
        return CMutableTxIn(COutPoint(lx(self.tx_id), self.index))
```

```
class Transaction:
    class Network(enum.Enum):
        MAINNET = "mainnet"
        TESTNET = "testnet"

    def __init__(self, private_key: str, network: Network = Network.TESTNET):
        self._network = network
        bitcoin.SelectParams(self._network.value)

        self._private_key = bitcoin.wallet.CBitcoinSecret(private_key)
        self._public_key = self._private_key.pub
        self._address = bitcoin.wallet.P2PKHBitcoinAddress.from_pubkey(self._public_key)
        self._destinations = []
        self._utxos = []
        self._tx = CMutableTransaction()

    @property
    def address(self) -> str:
        return str(self._address)

    def add_destination(self, destination: Destination) -> None:
        self._destinations.append(destination)

    def add_utxo(self, utxo: UnspentTransactionOutput) -> None:
        self._utxos.append(utxo)

    def create(self) -> requests.Response:
        if not self._destinations:
            raise ValueError("No destinations were added to the transaction")
        if not self._utxos:
            raise ValueError(
                "No unspent transaction outputs were added to the transaction"
            )
        self._create_transaction()
        self._verify()
        return self._broadcast_transaction()

    def my_P2PKH_script_pub_key(self) -> CScript:
        return P2PKH_script_pub_key(Hash160(self._public_key))

    def _my_P2PKH_script_sig(self, txin_script_pub_key: CScript) -> CScript:
        signature = self._create_OP_CHECKSIG_signature(txin_script_pub_key)
        return CScript([signature, self._public_key])  # type: ignore

    def _create_transaction(self) -> None:
        txins = [utxo.TxIn for utxo in self._utxos]
        txouts = [destination.TxOut for destination in self._destinations]
        self._tx = CMutableTransaction(txins, txouts)

    def _create_OP_CHECKSIG_signature(self, txin_script_pub_key: CScript) -> bytes:
        sighash = SignatureHash(txin_script_pub_key, self._tx, 0, SIGHASH_ALL)
        sig = self._private_key.sign(sighash) + bytes([SIGHASH_ALL])  # type: ignore
        return sig

    def _verify(self):
        for i, _ in enumerate(self._utxos):
            txin_script_pub_key = self._utxos[i].script_pub_key
            txin_script_sig = self._utxos[i].custom_sig
            if txin_script_sig is None:
                txin_script_sig = self._my_P2PKH_script_sig(txin_script_pub_key)
            self._tx.vin[i].scriptSig = txin_script_sig
            VerifyScript(
                self._tx.vin[i].scriptSig,
                txin_script_pub_key,
                self._tx,
                i,
                (SCRIPT_VERIFY_P2SH,),
            )

    def _broadcast_transaction(self) -> requests.Response:
        raw_transaction = b2x(self._tx.serialize())
        headers = {"content-type": "application/x-www-form-urlencoded"}
        return requests.post(
            TRANSACTION_BROADCAST_URL,
            headers=headers,
            data='{"tx": "%s"}' % raw_transaction,
            timeout=60,
        )
```

1. خروجی اول غیر قابل خرج و خروجی دوم قابل خرج توسط هرکس

برای خروجی‌ای که توسط هیچ‌کس قابل خرج نیست، می‌توانیم از اسکریپت زیر استفاده کنیم:

Script = OP_RETURN

همچنین برای خروجی‌ای که توسط هرکس قابل خرج شدن است، می‌توان از اسکریپت زیر استفاده کرد:

Script = OP_CHECKSIG

در این بخش از مقدار 0.0137566 بیت‌کوین موجود، 0.002 آن را به خروجی غیر قابل خرج، 0.008 آن را به خروجی قابل خرج توسط هرکس و باقی آن را به Transaction Fee اختصاص دادم. این کار توسط کد زیر که در فایل part2_q1_1.py قابل دسترس است، انجام شده است:

```
UNSPENDABLE_SCRIPT_PUB_KEY = CScript([OP_RETURN])  # type: ignore
SPENDABLE_BY_ANYONE_SCRIPT_PUB_KEY = CScript([OP_CHECKSIG])  # type: ignore


def main():
    private_key = "92Zh9ENA7DeNBr3FXa1QMLi4igPAzUKy44TEPMW7rogBtGz4CaR"

    tx = Transaction(private_key)
    tx.add_destination(Destination(tx.address, 0.002, UNSPENDABLE_SCRIPT_PUB_KEY))
    tx.add_destination(
        Destination(tx.address, 0.008, SPENDABLE_BY_ANYONE_SCRIPT_PUB_KEY)
    )
    tx.add_utxo(
        UnspentTransactionOutput(
            "12a4ccff3ed9bec0715fcf678b914397e1e7b3fcea239fe551f269c89810f909",
            0,
            tx.my_P2PKH_script_pub_key(),
        )
    )

    resp = tx.create()
    print(f"[{resp.status_code}] {resp.reason}")
    print(resp.text)
```

اطلاعات این تراکنش در این لینک و این لینک قابل دسترس است. هش تراکنش نیز به صورت زیر است:

76d3ef0f1c733e5b6a15da0233ceca7a5694674cf3f511c6015fdc3d7f52b00a

حال باید مقدار قابل خرج را به حساب خود برگردانیم. از مقدار 0.008 بیت‌کوین این خروجی، 0.0078 را به حساب خودم برگرداندم و باقی آن را به Transaction Fee اختصاص دادم. این کار را با روش P2PKH و با کد زیر که در فایل part2_q1_2.py قرار دارد، انجام دادم:

```
SPENDABLE_BY_ANYONE_SCRIPT_PUB_KEY = CScript([OP_CHECKSIG])  # type: ignore


def main():
    private_key = "92Zh9ENA7DeNBr3FXa1QMLi4igPAzUKy44TEPMW7rogBtGz4CaR"

    tx = Transaction(private_key)
    tx.add_destination(Destination("mu2XGdXpAM8fkFMWksFmMqCYP6oUXTVxRs", 0.0078))
    tx.add_utxo(
        UnspentTransactionOutput(
            "76d3ef0f1c733e5b6a15da0233ceca7a5694674cf3f511c6015fdc3d7f52b00a",
            1,
            SPENDABLE_BY_ANYONE_SCRIPT_PUB_KEY,
        )
    )

    resp = tx.create()
    print(f"[{resp.status_code}] {resp.reason}")
    print(resp.text)
```

اطلاعات این تراکنش در این لینک و این لینک قابل دسترس است. همچنین، هش تراکنش به صورت زیر است:

15c794bbb169f272fb1ed45526eef5613a851716a1df574480c12f23126772be

2. خروجی از نوع MultiSig

برای این بخش ابتدا 3 آدرس جدید ایجاد می‌کنیم که به صورت زیر هستند:

Private Key 1: 93RdsGKJn6ExkLEWpVogpC5kRibgoJDWZWUzjonohYhynkScHLX

Public Key 1: 04c9a3dc14a523ad8b6b36f445ce55475e0f716ff0a8e7f92103e36cb344a64a2160c48c2e97a8471537da6154acf6d612d2eae18af07884e25eec1b4dd07a59e9

Address 1: n3FpTJHHU17VrYxgAbUmuhYUvRoCTzhEzR

Private Key 2: 938zdHuD6PVuUb26s31xevjTBPY6fggvrxv9fhJz5UGeAfqF61j

Public Key 2: `04844325ab760b86d29b90a59d88d57ed3ef8b9124b804257ec15d7b4e426400cc74e028f73c949e36b1919aaf823dbc35440538e49151e28ce8c80528f0e0dd21`
Address 2: `mgPaYYnNtAC6V3NNrHjfU43HkszV9vszFu`

Private Key 3: `91cxgQtoYrMLUcDi3PsRvQnPQa9hgx9jc3keaq3vNXzmiaBFBrW`
Public Key 3: `048f6a3736d960a861bce581abd4e24f87977dc2804cd3eddec08dbe03804aea562ba30396818f3cf3cfdf0eb0ed8841abb2b3b28510ce89922d275f1f58e707b8`
Address 3: `mmux7bRFj5SVKKQyqGvrJiysSNJw2D9Axw`

حال از حالت 2-of-3 در روش MultiSig استفاده می‌کنیم. اسکریپت زیر می‌تواند در این بخش مورد استفاده قرار بگیرد:

Script: `2 <public_key1> <public_key2> <public_key3> 3 OP_CHECKMULTISIG`

در این بخش از کد زیر استفاده کردم که در فایل part2_q2_1.py قابل دسترس است:

```
def multi_sig_2_of_3(pub1: bytes, pub2: bytes, pub3: bytes) -> CScript:
    return CScript([2, pub1, pub2, pub3, 3, OP_CHECKMULTISIG])  # type: ignore


def main():
    private_key = "92Zh9ENA7DeNBr3FXa1QMLi4igPAzUKy44TEPMW7rogBtGz4CaR"

    pub1 = bytes.fromhex(
"04c9a3dc14a523ad8b6b36f445ce55475e0f716ff0a8e7f92103e36cb344a64a2160c48c2e97a8471537da6154acf6d612d2eae18af07884e25eec1b4dd07a59e9"
    )
    pub2 = bytes.fromhex(
"04844325ab760b86d29b90a59d88d57ed3ef8b9124b804257ec15d7b4e426400cc74e028f73c949e36b1919aaf823dbc35440538e49151e28ce8c80528f0e0dd21"
    )
    pub3 = bytes.fromhex(
"048f6a3736d960a861bce581abd4e24f87977dc2804cd3eddec08dbe03804aea562ba30396818f3cf3cfdf0eb0ed8841abb2b3b28510ce89922d275f1f58e707b8"
    )

    tx = Transaction(private_key)
    tx.add_destination(
        Destination(tx.address, 0.0076, multi_sig_2_of_3(pub1, pub2, pub3))
    )
    tx.add_utxo(
        UnspentTransactionOutput(
            "15c794bbb169f272fb1ed45526eef5613a851716a1df574480c12f23126772be",
            0,
            tx.my_P2PKH_script_pub_key(),
        )
    )

    resp = tx.create()
    print(f"[{resp.status_code}] {resp.reason}")
    print(resp.text)
```

اطلاعات این تراکنش (انتقال مقدار 0.0076 بیت‌کوین) در این لینک و این لینک قابل دسترس است. همچنین هش تراکنش به صورت زیر است:

`6c9cea1530ef89838e551bf08481ba4193be9f0ed9d8dcdcceff20a645f9e357`

حال باید با استفاده از signature دو تا از سه آدرس ایجاد شده، این مقدار را به آدرس اصلی برگردانیم. در این بخش 0.0074 بیت‌کوین را به آدرس اصلی بازمی‌گردانم و باقی آن را به Transaction Fee اختصاص می‌دهم. این کار توسط کد زیر که در فایل part2_q2_2.py قرار دارد، انجام شده است:

```
def multi_sig_2_of_3(pub1: bytes, pub2: bytes, pub3: bytes) -> CScript:
    return CScript([2, pub1, pub2, pub3, 3, OP_CHECKMULTISIG])  # type: ignore


def sig_script_2_of_3(sig1: bytes, sig2: bytes) -> CScript:
    return CScript([OP_0, sig1, sig2])  # type: ignore


def sign(
    tx: Transaction, utxo_index: int, private_key: bitcoin.wallet.CBitcoinSecret
) -> bytes:
    txin_script_pub_key = tx._utxos[utxo_index].script_pub_key
    sighash = SignatureHash(txin_script_pub_key, tx._tx, 0, SIGHASH_ALL)
    return private_key.sign(sighash) + bytes([SIGHASH_ALL])  # type: ignore


def main():
    private_key = "92Zh9ENA7DeNBr3FXa1QMLi4igPAzUKy44TEPMW7rogBtGz4CaR"

    bitcoin.SelectParams("testnet")
    private_key1 = bitcoin.wallet.CBitcoinSecret(
        "93RdsGKJn6ExkLEWpVogpC5kRibgoJDWZWUzjonohYhynkScHLX"
    )
    public_key1 = private_key1.pub
    private_key2 = bitcoin.wallet.CBitcoinSecret(
        "938zdHuD6PVuUb26s31xevjTBPY6fggvrxv9fhJz5UGeAfqF61j"
    )
    public_key2 = private_key2.pub
    private_key3 = bitcoin.wallet.CBitcoinSecret(
        "91cxgQtoYrMLUcDi3PsRvQnPQa9hgx9jc3keaq3vNXzmiaBFBrW"
    )
    public_key3 = private_key3.pub

    tx = Transaction(private_key)

    tx.add_destination(Destination(tx.address, 0.0074))
    tx.add_utxo(
        UnspentTransactionOutput(
            "6c9cea1530ef89838e551bf08481ba4193be9f0ed9d8dcdceff20a645f9e357",
            0,
            multi_sig_2_of_3(public_key1, public_key2, public_key3),
        )
    )

    tx._create_transaction()

    sig1 = sign(tx, 0, private_key1)
    sig2 = sign(tx, 0, private_key2)
    tx._utxos[0]._custom_sig = sig_script_2_of_3(sig1, sig2)

    resp = tx.create()
    print(f"[{resp.status_code}] {resp.reason}")
    print(resp.text)
```

اطلاعات این تراکنش در این لینک و این لینک و همچنین هش زیر قابل دسترس است:

`804b02bd9c3db8c2c480cca63342a6758d7c91c221028403df82ae3be483dba1`

3. اطلاع از 2 عدد اول برای خرج تراکنش

در این بخش با توجه به اینکه scriptPubKey برای همه افراد نمایان می‌شود، اگر حاصل جمع و تفریق اعداد اول را به صورت مستقیم در این اسکریپت بگذاریم، هر فردی می‌تواند دو عدد اول را بدست آورده و تراکنش را خرج کند. به همین دلیل، از خاصیت Preimage Resistance توابع هش استفاده می‌کنیم و هش جمع و تفریق این دو عدد را در اسکریپت قرار می‌دهیم. در واقع اسکریپت مد نظر می‌تواند به صورت زیر باشد:

Script: `OP_2DUP, OP_ADD, OP_HASH160, Hash160(SUM), OP_EQUALVERIFY, OP_SUB, OP_HASH160, Hash160(DIFF), OP_EQUAL`

در این حالت کافیست عدد دوم را در top استک و عدد اول را زیر آن قرار دهیم و نتیجه را به عنوان scriptSig استفاده کنیم. در این بخش اعداد اول به صورت زیر انتخاب شده‌اند:

Num1 = 977

Num2 = 881

Sum = 1858

Diff = 96

نکته: به دلیل اشتباهی که در scriptPubKey این بخش داشتم، مجددا از Faucet ذکر شده مقداری بیت‌کوین دریافت کردم که اطلاعات این تراکنش در این لینک و این لینک و هش زیر قابل دسترسی است:

6c7b5a4c4551bafd06b8279f4a64c445d4b46245a65a38c31c058468909c36ff

این کار توسط کد زیر که در فایل part2_q3_1.py قابل دسترس است، انجام شده است:

```
PRIME_NUM1 = 977
PRIME_NUM2 = 881

SUM = PRIME_NUM1 + PRIME_NUM2
SUM_IN_BYTES = SUM.to_bytes(2, byteorder="little")
DIFF = PRIME_NUM1 - PRIME_NUM2
DIFF_IN_BYTES = DIFF.to_bytes(1, byteorder="little")

SCRIPT_SUM_DIFF_PUB_KEY = CScript([OP_2DUP, OP_ADD, OP_HASH160, Hash160(SUM_IN_BYTES), OP_EQUALVERIFY, OP_SUB,
OP_HASH160, Hash160(DIFF_IN_BYTES), OP_EQUAL])  # type: ignore


def main():
    private_key = "92Zh9ENA7DeNBr3FXa1QMLi4igPAzUKy44TEPMW7rogBtGz4CaR"

    tx = Transaction(private_key)
    tx.add_destination(Destination(tx.address, 0.015, SCRIPT_SUM_DIFF_PUB_KEY))
    tx.add_utxo(
        UnspentTransactionOutput(
            "6c7b5a4c4551bafd06b8279f4a64c445d4b46245a65a38c31c058468909c36ff",
            1,
            tx.my_P2PKH_script_pub_key(),
        )
    )

    resp = tx.create()
    print(f"[{resp.status_code}] {resp.reason}")
    print(resp.text)
```

لازم به ذکر است که حاصل جمع یک عدد 2 بایتی و حاصل تفریق آن‌ها یک عدد 1 بایتی است.

اطلاعات این تراکنش در این لینک و این لینک و همچنین هش زیر قابل دسترس است:

e2b73c7d033a2f672b65f64c68322aced2bccc4db44e7cd7a355dcf7ba0b2955

حال باید این تراکنش را به حساب خودمان بازگردانیم. در این حالت اسکریپت Sig به صورت زیر است:

Script: 977, 881

این کار توسط کد زیر که در فایل part2_q3_2.py قرار دارد انجام شده است:

```
PRIME_NUM1 = 977
PRIME_NUM1_IN_BYTES = PRIME_NUM1.to_bytes(2, byteorder="little")
PRIME_NUM2 = 881
PRIME_NUM2_IN_BYTES = PRIME_NUM2.to_bytes(2, byteorder="little")

SUM = PRIME_NUM1 + PRIME_NUM2
SUM_IN_BYTES = SUM.to_bytes(2, byteorder="little")
DIFF = PRIME_NUM1 - PRIME_NUM2
DIFF_IN_BYTES = DIFF.to_bytes(1, byteorder="little")

SCRIPT_SUM_DIFF_PUB_KEY = CScript([OP_2DUP, OP_ADD, OP_HASH160, Hash160(SUM_IN_BYTES), OP_EQUALVERIFY, OP_SUB,
OP_HASH160, Hash160(DIFF_IN_BYTES), OP_EQUAL])  # type: ignore
SCRIPT_PRIME_NUMS_SIG = CScript([PRIME_NUM1_IN_BYTES, PRIME_NUM2_IN_BYTES]) # type: ignore


def main():
    private_key = "92Zh9ENA7DeNBr3FXa1QMLi4igPAzUKy44TEPMW7rogBtGz4CaR"

    tx = Transaction(private_key)
    tx.add_destination(Destination(tx.address, 0.014))
    tx.add_utxo(
        UnspentTransactionOutput(
            "e2b73c7d033a2f672b65f64c68322aced2bccc4db44e7cd7a355dcf7ba0b2955",
            0,
            SCRIPT_SUM_DIFF_PUB_KEY,
            SCRIPT_PRIME_NUMS_SIG,
        )
    )

    resp = tx.create()
    print(f"[{resp.status_code}] {resp.reason}")
    print(resp.text)
```

اطلاعات این تراکنش در این لینک و این لینک و همچنین هش زیر قابل دسترس است:

65f47ef01f3145ed2993f570e9c573e3a7299cd7a27a2bff295d1d7346638965

قسمت سوم: استخراج بلوک

ابتدا باید یک آدرس مخصوص Mainnet داشته باشیم که به صورت زیر است:

Private Key (WIF): 5JWoEUpPb1BCRMTYUqNNq4L7eEAptfiz9FKsBAj7niAJWaQ6uZJ

Public Key: 04bd113d6628b7a3054293dcbfa4d0e98af6cdf1d0f1d519ce7e1033b43acd8e8f609a0191c30b2c614408f59f00ac3f5f028974216b0710d611073dd141909bc8

Address: 1Ny5UQ4B6XRuyuB8BPgUPvCXDdR3WV9xd5

با توجه به شماره دانشجویی‌ام، باید از بلاک 9385 استفاده کنم که هش آن به صورت زیر است:

00000000673405ffe87f801032e901c7f423adddc7b51773e6b108e617e75516

حال برای تولید تراکنش coinbase، از یک کلاس که از کلاس Transaction ارث می‌برد استفاده می‌کنم:

```
class BaseCoinTransaction(Transaction):
    def __init__(
        self,
        private_key: str,
        data: str,
        network: Transaction.Network = Transaction.Network.MAINNET,
    ):
        super().__init__(private_key, network)
        self._data = data

        self._destinations.append(Destination(self.address, BITCOIN_MINE_AWARD))
        self._utxos.append(UnspentTransactionOutput("0" * 64, 0xFFFFFFFF, CScript([]), self._get_coinbase_sig()))  # type: ignore

    def _get_coinbase_sig(self) -> CScript:
        hex_data = self._data.encode("utf-8").hex()
        return CScript([bytes.fromhex(hex_data)])  # type: ignore

    def create(self) -> CMutableTransaction:
        self._create_transaction()
        self._tx.vin[0].scriptSig = self._utxos[0].custom_sig
        return self._tx

    def _broadcast_transaction(self) -> Response:
        raise NotImplementedError("BaseCoinTransaction cannot be broadcasted")

    def _verify(self) -> None:
        raise NotImplementedError("BaseCoinTransaction cannot be verified")
```

در اینجا مقدار BITCOIN_MINE_AWARD برابر با 6.25 بیت‌کوین در نظر گرفته شده است.

داده مدنظر (810199385PashaBarahimi) در scriptSig قرار می‌گیرد.

همانطور که در منابع اشاره شده، داده‌ها در بلاک بیت‌کوین به صورت little endian ذخیره می‌شوند و به همین دلیل این مورد در کد نیز رعایت شده است. کد زیر برای mine کردن بلاک نوشته شده است (این کد در فایل part3.py در دسترس است):

```
def main():
    data = "810199385PashaBarahimi"
    private_key = "5JWoEUpPb1BCRMTYUqNNq4L7eEAptfiz9FKsBAj7niAJWaQ6uZJ"
    bits = "0x1f010000"  # 16 bits leading 0s
    timestamp = int(time.time())
    _ = int(input("Enter the previous block number: "))  # unused
    prev_hash = input("Enter the previous block hash: ")

    basecoin = BaseCoinTransaction(private_key, data, Transaction.Network.MAINNET)
    tx = basecoin.create()

    block = BitcoinBlock([tx], prev_hash, bits, timestamp)
    print("Mining...")
    hash_value = block.mine()

    print(f"Block hash:   {b2lx(hash_value)}")
    print(f"Block header: {b2x(block.header)}")
    print(f"Block body:   {b2x(block.body)}")
    print(f"Block hex:    {b2x(block.block)}")

    print(f"Merkle root:  {block.merkle_root}")
    print(f"Nonce:        {block.nonce}")
    print(f"Version:      {block.version}")
    print(f"Timestamp:    {timestamp}")
    print(f"Bits:         {bits}")
    print(f"Target:       {block.target}")
```

پیاده‌سازی کلاس BitcoinBlock نیز به صورت زیر است:

```
class BitcoinBlock:
    def __init__(
        self,
        transactions: list[CMutableTransaction],
        prev_block_hash: str,
        bits: str = "0x1f010000",
        timespamp: int = int(time.time()),
    ):
        self._transactions = transactions
        self._prev_block_hash = prev_block_hash
        self._merkle_root = self._calculate_merkle_root()
        self._timestamp = timespamp
        self._bits = int(bits, 16)
        self._target = self._get_target(bits)
        self._version = 2
        self._nonce = 0
        self._partial_header = self._get_partial_header()
        self._header = self._partial_header
        self._body = self._get_body()

    # some properties

    def _calculate_merkle_root(self) -> str:
        hashes = [Hash(tx.serialize()) for tx in self._transactions]
        while len(hashes) > 1:
            if len(hashes) % 2 != 0:
                hashes.append(hashes[-1])
            hashes = [
                Hash(hash1 + hash2) for hash1, hash2 in zip(hashes[::2], hashes[1::2])
            ]
        return b2lx(hashes[0])

    @staticmethod
    def _get_target(bits: str) -> bytes:
        exponent = bits[2:4]
        coefficient = bits[4:]
        target = int(coefficient, 16) * 2 ** (8 * (int(exponent, 16) - 3))
        target_hex = hex(target)[2:]
        return bytes.fromhex(target_hex.zfill(64))

    def _get_hash_value(self) -> bytes:
        nonce = struct.pack("<L", self._nonce)
        self._header = self._partial_header + nonce
        return Hash(self._header)

    def _get_partial_header(self) -> bytes:
        return (
            struct.pack("<L", self._version)
            + bytes.fromhex(self._prev_block_hash)[::-1]
            + bytes.fromhex(self._merkle_root)[::-1]
            + struct.pack("<LL", self._timestamp, self._bits)
        )

    def _get_body(self) -> bytes:
        return b"".join(tx.serialize() for tx in self._transactions)

    def _print_hash_rate(self, start: float, end: bool = False) -> None:
        elapsed_time = time.time() - start
        rate = self._nonce / elapsed_time
        if rate < 1e3:
            unit = "H/s"
        elif rate < 1e6:
            rate /= 1e3
            unit = "KH/s"
        elif rate < 1e9:
            rate /= 1e6
            unit = "MH/s"
        else:
            rate /= 1e9
            unit = "GH/s"
        print(f"\r{' ' * 20}\rHash rate: {rate:.2f} {unit}", end="")
        if end:
            print()

    def mine(self) -> bytes:
        start = time.time()
        while self._nonce < 2**32:
            hash_value = self._get_hash_value()
            if hash_value[::-1] < self._target:
                self._print_hash_rate(start, end=True)
                return hash_value
            self._nonce += 1
            if self._nonce % 1000 == 0:
                self._print_hash_rate(start)
        raise ValueError("Nonce overflow")
```

نتیجه اجرای کد نیز به صورت زیر است:

Block Header: 020000001655e717e608b1e67317b5c7ddad23f4c701e93210807fe8ff0
53467000000000ddd704686669647e4709fa55767f5e7c7b898ba321e7419c41ecb7e15
6858759a6a070640000011fe73e0100

Block Body (Transactions): 0100000001000000000000000000000000000000000000000000
0000000000000000000000000ffffffff1716383130313939333835506173686142617
26168696d69ffffffff0140be4025000000001976a914f0f5ab2dba58627e6c6999eb0
e45fb878efe111e88ac00000000

Hash: 000043d18dcb1b05f6cc9c637152f8a162822759c8c188ccc791bcebda327a74

داده ذخیره شده (810199385PashaBarahimi) به صورت hex در بخش زیر نمایش داده شده است:

810199385PashaBarahimi: 0x3831303139393338355061736861426172616869 6D69