

Crypto-proj1

Sepehr Azardar 810199357

سوال 1)

چه تفاوتی میان آدرس های شبکه ای اصلی و آزمایشی وجود دارد؟

آدرس های شبکه تست، همان طور که از اسمشان برمیآید، برای تست و آزمایش هستند و هیچ مقدار ارزشمندی را در خود نگه داری نمیکنند. اما در شبکه اصلی هر آدرس به بیت کوین های واقعی اشاره میکند و ارزش مالی دارند. تفاوت دیگر آنها در ظاهرشان هست. پیشوند های متفاوتی دارند.

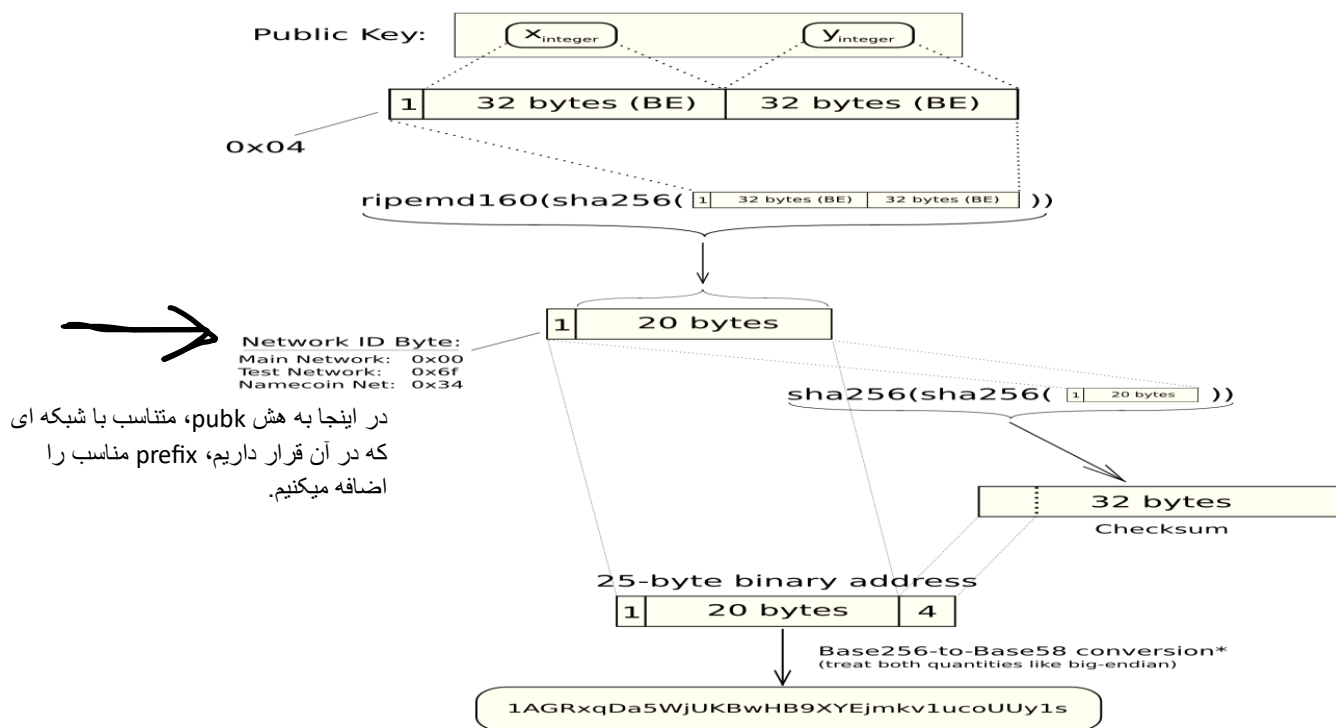
Version Byte prefix - Indicates which network the private key is to be used on.

0x80 = Mainnet

0xEF = Testnet

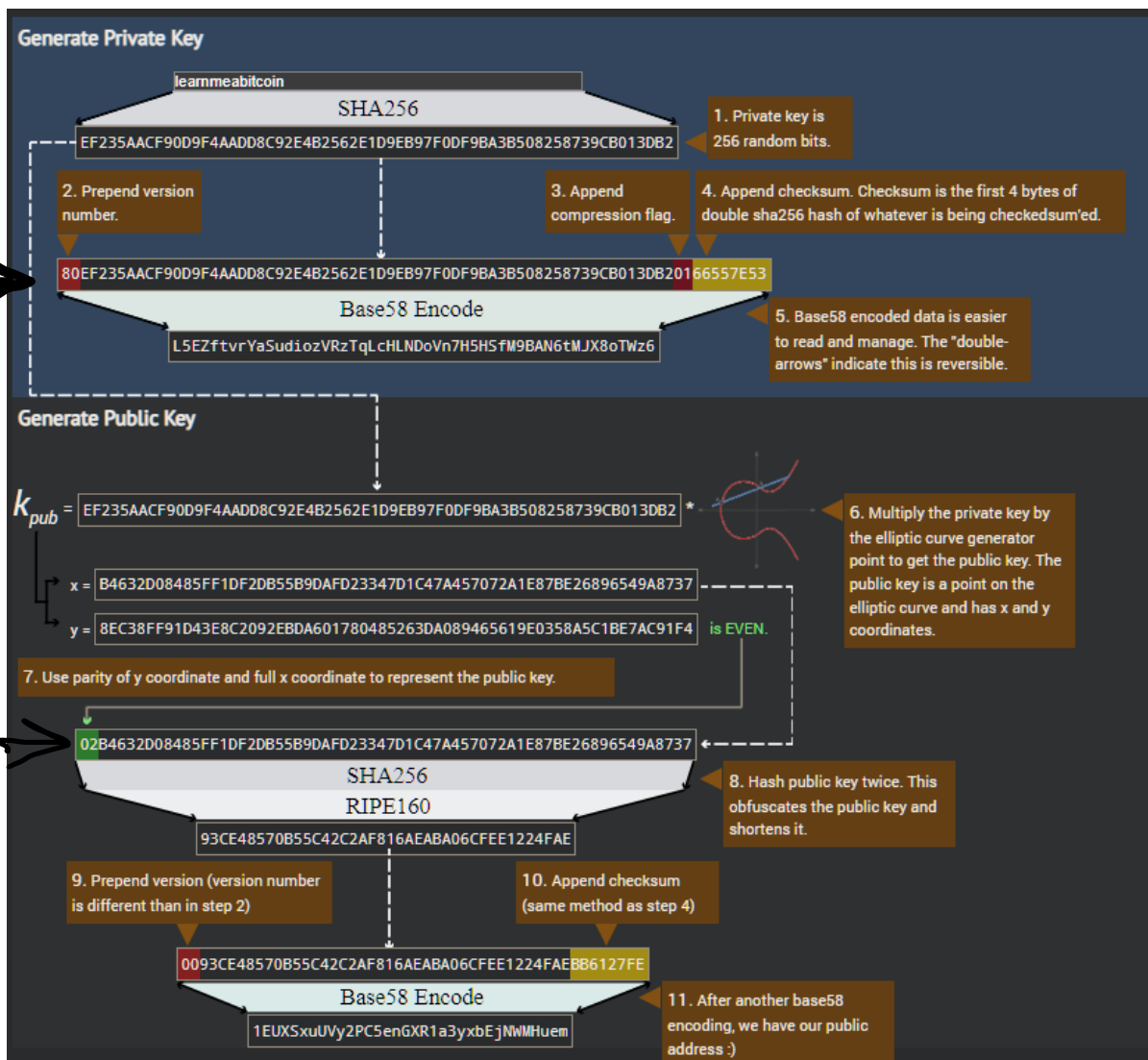
دو نوع public key داریم، uncompressed، و ورژن جدید تر که compressed هست. و prefix ای که برای mainnet و testnet داریم، با همدیگر فرق دارند ولی فرایند ساخت شان و ... شبیه هم هست. در واقع در دوجا ما prefix اضافه میکنیم و باتوجه به اینکه در mainnet هستیم یا testnet ... prefix مناسب را اضافه میکنیم. یکی در فرمت wif برای کلید خصوصی و دیگری در wallet-addr قبل از base58 و checksum:

Elliptic-Curve Public Key to BTC Address conversion



*In a standard base conversion, the 0x00 byte on the left would be irrelevant (like writing '052' instead of just '52'), but in the BTC network the left-most zero chars are carried through the conversion. So for every 0x00 byte on the left end of the binary address, we will attach one '1' character to the Base58 address. This is why main-network addresses all start with '1'.

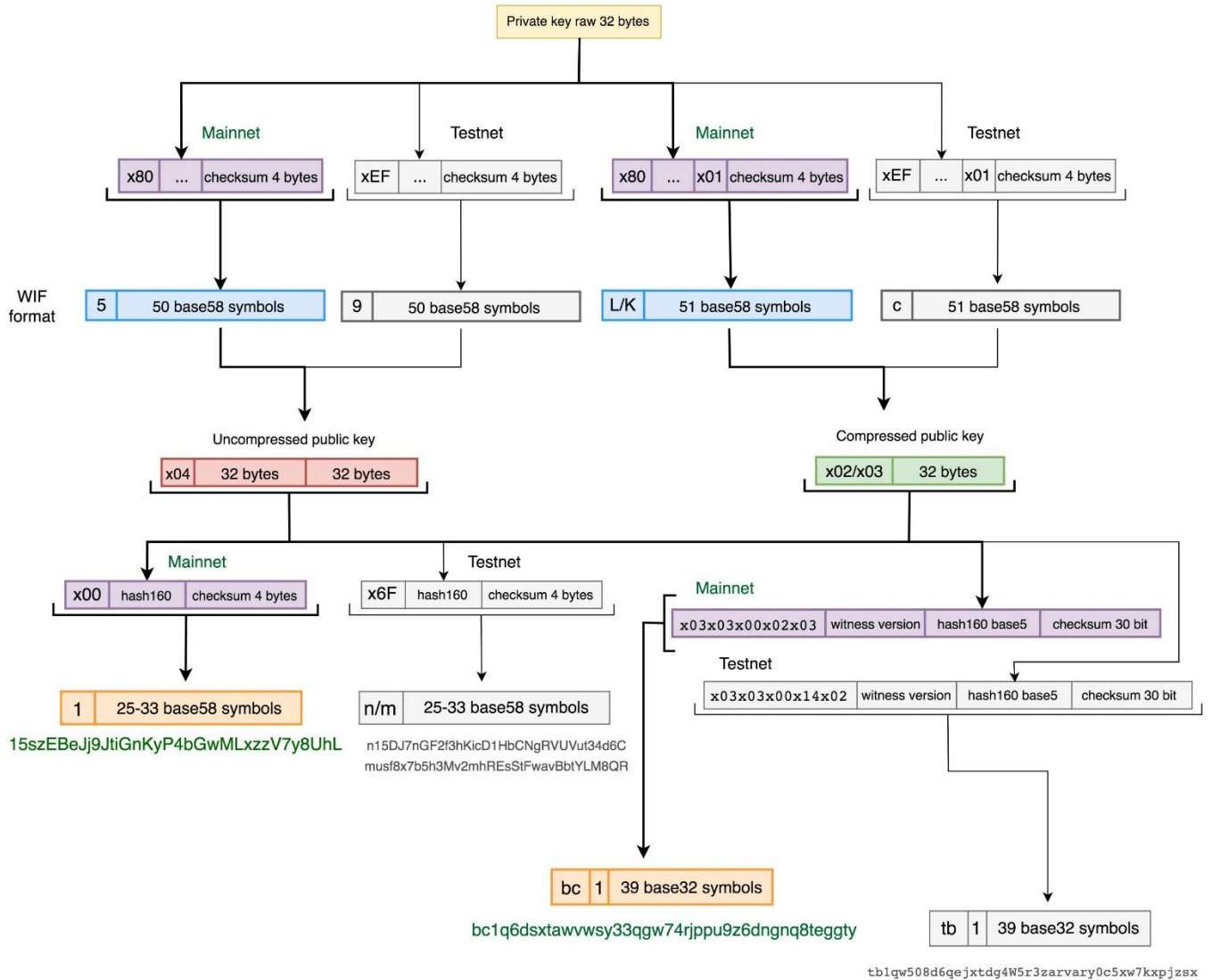
compressed key format: برای حالت



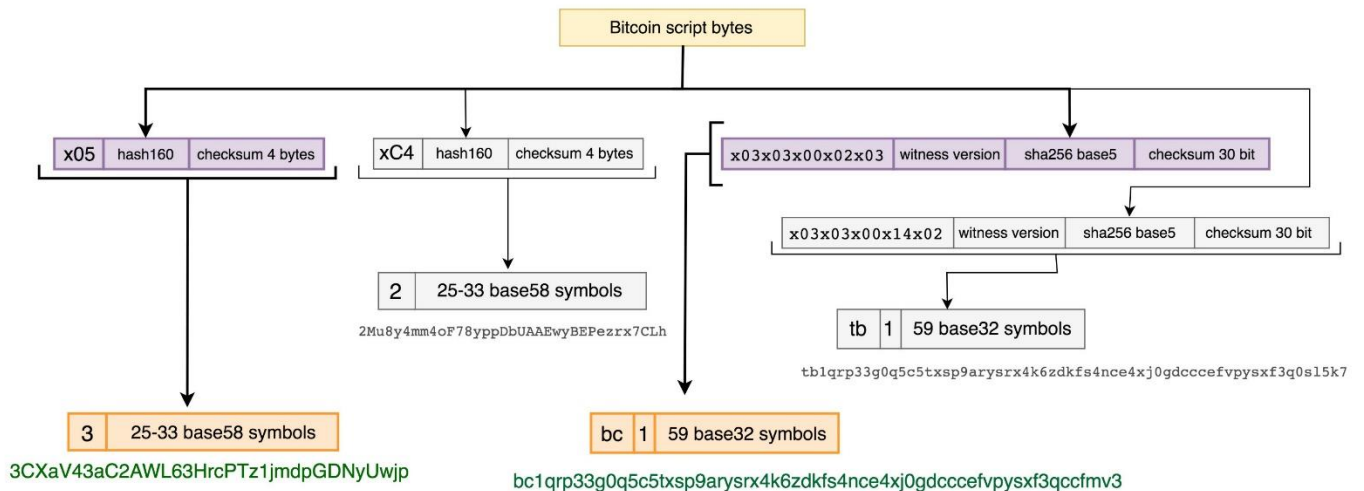
انواع prefix ها هستند، که باید توجه به اینکه در چه شبکه ای هستیم و از چه پروتکل ای استفاده میکنیم، پیشنهاد مناسب را به prk, pbk از تبدیل آنها (به ترتیب) به BTC addr و wif-format استفاده کنیم. نکته دیگری که وجود دارد این است که به ازای اضافه کردن پیشنهاد های با توجه به اینکه pbk مون در حالت compressed هست یا un compressed، میتونه پیشنهاد های متفاوتی را در base58 ببینیم.



Random 256 bit integer



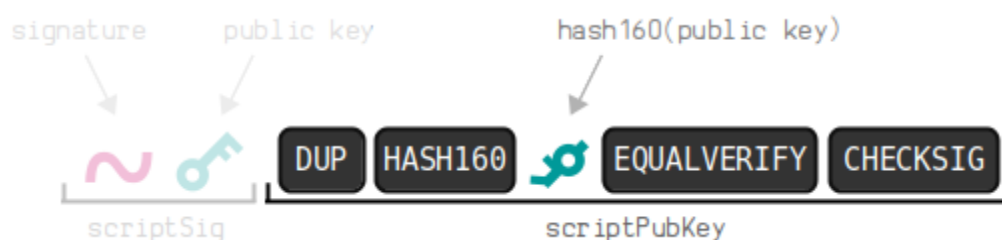
Bitcoin script



<https://learnmeabitcoin.com/technical/txid-footnote-unique-txids>

```
def P2PKH_scriptPubKey():  
    return [ OP_DUP, OP_HASH160, Hash160(my_public_key), OP_EQUALVERIFY, OP_CHECKSIG ]
```

هر tx پولی مشخصی را به هر خروجی خود، اختصاص میدهد و شرطی بر روی آن (برای خرج کردن) میگذارد و هرکس آن شرط را بتواند برقرار کند میتواند از خروجی آن تراکنس استفاده کند و آن را خرج کند. یکی از این script ها معروف، p2pkh (Pay-to-Public-Key-Hash) آدرس هایی که با 1 شروع میشوند از این روش استفاده میکنند.



به شرطی که توسط sender بر روی خروجی tx گذاشته میشود، scriptPubKey میگوییم. کسی که میخواهد این خروجی را خرج کند (UTXO) (دوتایی (TXID, VOUT)) باید ScriptSig را در کنار این utxo که به عنوان ورودی در تراکنش جدید (که شخص میخواهد آن را از این طریق خرج کند) قرار دارد، ارائه دهد. در صورتی که verify بشود شخص اجازه خرج کردن آن را خواهد داشت.

```
def P2PKH_scriptSig(txin, txout, txin_scriptPubKey):  
    signature = create_OP_CHECKSIG_signature(txin, txout, txin_scriptPubKey, my_private_key)  
    return [signature, my_public_key]
```

ScriptSig شامل pbk, sig هست که با قرار گرفتن در کنار scriptPubKey، شرط را pass میکند و فرد میتواند این خروجی تراکنش را خرج کند. Sig در واقع، امضا کردن هش چند مقدار با کلید خصوصی مان هست. این مقادیر:

Txin. s, txout. s, txin_scriptPubKey

```
def send_from_P2PKH_transaction(amount_to_send, txid_to_spend, utxo_index,  
                                txout_scriptPubKey):  
    txout = create_txout(amount_to_send, txout_scriptPubKey)  
    txin_scriptPubKey = P2PKH_scriptPubKey(my_address)  
    txin = create_txin(txid_to_spend, utxo_index)  
    txin_scriptSig = P2PKH_scriptSig(txin, txout, txin_scriptPubKey)  
    new_tx = create_signed_transaction(txin, txout, txin_scriptPubKey, txin_scriptSig)  
    return broadcast_transaction(new_tx)
```

همان طور که گفته شد، شرطی که برای خروجی میگذاریم، (scriptPubKey) به همراه مقدار، txout را برای میسازد، هر تراکنش از تعدادی، txout و txin و sig و ... تشکیل شده است.

```
def unspendable_script_PubKey():  
    return [OP_RETURN]
```

```
def public_spenttable_script_PubKey():  
    return [OP_TRUE]
```

چون که میخواستیم، unspendable tx داشته باشیم. پس OP_RETURN را اضافه میکنیم. تا هرکس دیگری بخواهد شرط این خروجی را برطرف کند در سر استک با return مواجه بشود و در نتیجه هیچ وقت شرط را نمیتوانیم pass کنیم و بنابراین کسی نمی تواند آن را خرج کند. از طرفی برای tx دومی، true بر میگردانیم. یعنی در واقع شرطی وجود ندارد. پس هرکسی میتواند آن را خرج کند بدون ارائه sig خاصی.

```
def create_two_outputs_signed_transaction(txin, first_txout, second_txout,  
                                          txin_scriptPubKey, txin_scriptSig):
```

این تابع تراکنشی را بر میگرداند که یه ورودی دارد و دو خروجی در حالت دیفالت برای یه ورودی و یه خروجی تابع داریم و برای حالت های دیگه باید extend کنیم.

```
def create_two_outputs_OP_CHECKSIG_signature(txin, first_txout,  
                                             second_txout, txin_scriptPubKey, seckey):
```

این تابع، sig برای تراکنشی که دوتا txout دارد را میسازد. به صورت default برای یه ورودی و یه خروجی تابع sig کار میکنه و برای حالت های دیگه باید تابع متناظرش رو بنویسیم.

```
def scriptSig(txin, first_txout, second_txout, txin_scriptPubKey):  
    signature = create_two_outputs_OP_CHECKSIG_signature(txin, first_txout,\  
                                                         second_txout, txin_scriptPubKey, my_private_key)
```

برای حالت دو خروجی و یه ورودی، sig, pubk رو برمیگردونه.

```
def gen_tx(first_amount_to_spend, second_amount_to_spend, txid_to_spend,  
           utxo_index, first_txout_scriptPubKey, second_txout_scriptPubKey):
```

تراکنشی با یه ورودی و دو خروجی با مقادیر داده شده و با txout_scriptPubKey ها و txin داده شده میسازد و در شبکه broadcast میکند.

```
def send_from_P2PKH_transaction(amount_to_send, txid_to_spend,  
                                utxo_index, txout_scriptPubKey):  
    txout = create_txout(amount_to_send, txout_scriptPubKey)  
    txin_scriptPubKey = get_txin_scriptPubKey()  
    txin = create_txin(txid_to_spend, utxo_index)  
    new_tx = create_signed_transaction(txin, txout, txin_scriptPubKey, [])  
    return broadcast_transaction(new_tx)
```

در اینجا، چون تراکنشی که می‌خواهیم به عنوان ورودی خرج کنیم از نوع public spendable هست (یعنی هر کسی میتواند از آن استفاده کند.) پس نیازی به txin_scriptSig. مانند قبل txin_scriptPubKey، txin، txout را می‌سازیم و در کنار هم تراکنش جدید را می‌سازیم و در شبکه broadcast می‌کنیم.

Tx.Q2: multi-sig

```
def multiSig_output_script():  
    return [OP_2, pbk1, pbk2, pbk3, OP_3, OP_CHECKMULTISIG]
```

برای multi-sig tx، قفل یا شرط (scriptPubKey) (که وظیفه ی verify کردن را به عهده دارد.) باید تمام pbk را در اسکرپت مون بیاوریم. و چون باید 2 تا از 3 نفر sig ارائه بدهند تا شرط pass شود برای همین از op_2, op_3 استفاده کرده ایم.

```
def create_multisig_transaction(amount_to_send, txid_to_spend, utxo_index,  
txout_scriptPubKey):  
    txout = create_txout(amount_to_send, txout_scriptPubKey)  
    txin_scriptPubKey = P2PKH_scriptPubKey()  
    txin = create_txin(txid_to_spend, utxo_index)  
    txin_scriptSig = P2PKH_scriptSig(txin, txout, txin_scriptPubKey)  
    new_tx = create_signed_transaction(txin, txout, txin_scriptPubKey,  
txin_scriptSig)  
    return broadcast_transaction(new_tx)
```

در اینجا مقدار btc، utxo و txout_scriptPubKey را به عنوان ورودی می‌دهیم و تراکنش مورد نظر ساخته می‌شود و در شبکه broadcast می‌شود. در اینجا تنها تفاوت با سوال قبل در txout_scriptPubKey میباشد که فرمت آن به شکل قطعه کد بالا میباشد.

```
def multi_scriptSig(txin, txout, txin_scriptPubKey):  
    sig1 = create_OP_CHECKSIG_signature(txin, txout, txin_scriptPubKey, prk1)  
    sig2 = create_OP_CHECKSIG_signature(txin, txout, txin_scriptPubKey, prk2)  
    return [OP_0, sig1, sig2]
```

در اینجا scriptSig برای multi، مینویسیم. در واقع در scriptSig ما مقادیر لازم برای pass کردن شرط گذاشته شده برای خرج کردن را provide می‌کنیم (و scriptPubKey وریفای میکند.) و بدین شکل میتوانیم به (خروجی از) تراکنش را خرج کنیم. برای اینکار باید sig ها دوفر مورد نظر را فراهم کنیم. تا وقتی scriptPubKey، scriptSig در کنار هم قرار میگیرند تا به فرمت زیر برسیم.

OP_0, sig1, sig2 OP_2, pbk1, pbk2, pbk3, OP_3, OP_CHECKMULTISIG

و در صورت درست بودن مقادیر شرط پاس میشود و اجازه خرج کردن داده میشود، در غیر این صورت رد میشه و در شبکه propagate نمیشه

TX Q3

دو عدد اول که در نظر گرفته شده: 9851 و 7789

```
def my_locking_script(sum_byte, sub_byte):  
    return [OP_2DUP, OP_ADD, OP_HASH160, Hash160(sum_byte),  
            OP_EQUALVERIFY, OP_SUB, OP_HASH160, Hash160(sub_byte), OP_EQUAL]
```

در اینجا برای VERIFY کردن جمع و تفریقاز مکانیزم هش استفاده شده است. ما اینجا نیمه LOCK/VERIFY SCRIPT اش رو مینویسیم و VALE/SIG SCRIPT که همان P1,P2 هستند،(که ابتدا باید عدد بزرگ تر توسط SPENDER، فراهم شود.) نیمه دیگر SCRIPT هست تا شرط PASS شود.

در واقع فرم کامل SCRIPT:

P1, P2, OP_2DUP, OP_ADD, OP_HASH160, Hash160(sum_byte),

OP_EQUALVERIFY, OP_SUB, OP_HASH160, Hash160(sub_byte), OP_EQUAL

ابتدا P1,P2 را با OP_2DUP، دو برابر شده . سپس با همدیگر جمع میشوند و سپس هش گرفته میشود و مقدار آن با مقدار هشی که از قبل بدست آمده است مقایسه میشود و در صورت برابری، تفریق میکنیم، هش میگیریم و برابری را چک میکنیم. در صورت درست بودن، این بدین معناست که spender همان p1,p2 ای که ما در نظر گرفته ایم را دارد.

```
def run(amount_to_send, txid_to_spend, utxo_index):  
    prime_num1 = 9851  
    prime_num2 = 7789  
    sum_prm = prime_num1 + prime_num2  
    sub_prm = prime_num1 - prime_num2  
    sum_bytes = sum_prm.to_bytes(2, 'little')  
    sub_bytes = sub_prm.to_bytes(2, 'little')  
    txout_scriptPubKey = my_locking_script(sum_bytes, sub_bytes)  
    txout = create_txout(amount_to_send, txout_scriptPubKey)  
    txin_scriptPubKey = get_txin_scriptPubKey()  
    txin = create_txin(txid_to_spend, utxo_index)  
    txin_scriptSig = P2PKH_scriptSig(txin, txout, txin_scriptPubKey)  
    new_tx = create_signed_transaction(txin, txout, txin_scriptPubKey,  
txin_scriptSig)  
    response = broadcast_transaction(new_tx)
```

در اینجا حاصل جمع و تفریق را ابتدا به بایت تبدیل کرده و سپس هش میگیریم و txout_scriptPubKey را میسازیم.

و بقیه مراحل مانند قبل هست و در آخر تراکنش را در شبکه منتشر میکنیم.

Q3.2

```
def my_unlocking_script(p1,p2):  
    return [p1,p2]
```

همانطور که گفته شد، scriptSig در اینجا باید p1,p2 را فراهم کند.

```
def run(amount_to_send, txid_to_spend, utxo_index):  
    prime_num1 = 9851  
    prime_num2 = 7789  
    sum_prm = prime_num1 + prime_num2  
    sub_prm = prime_num1 - prime_num2  
    sum_bytes = sum_prm.to_bytes(2,'little')  
    sub_bytes = sub_prm.to_bytes(2,'little')  
  
    txout_scriptPubKey = get_txin_scriptPubKey()  
    txout = create_txout(amount_to_send, txout_scriptPubKey)  
    txin_scriptPubKey = my_locking_script(sum_bytes, sub_bytes)  
  
    txin = create_txin(txid_to_spend, utxo_index)  
  
    txin_scriptSig = my_unlocking_script(prime_num1.to_bytes(2,'little'),  
    prime_num2.to_bytes(2,'little'))  
    new_tx = create_signed_transaction(txin, txout, txin_scriptPubKey,  
    txin_scriptSig)  
    response = broadcast_transaction(new_tx)  
  
    print(response.status_code, response.reason)  
    print(response.text)
```

در اینجا هم برای فرستادن و تولید کردن تراکنش جدید باید، txin_scriptSig که در قطعه کد قبلی توضیح داده شد را بسازیم و بقیه موارد مشابه دفعه های قبلی میباشد. . برای اینکار، دو عدد اول انتخاب شده را به بایت تبدیل میکنیم و به عنوان ورودی pass میکنیم.

Q4

Block structure

Field	Description	Size
Magic no	value always 0xD9B4BEF9	4 bytes
Blocksize	number of bytes following up to end of block	4 bytes
Blockheader	consists of 6 items	80 bytes
Transaction counter	positive integer $VI = VarInt$	1 - 9 bytes
transactions	the (non empty) list of transactions	<Transaction counter>-many transactions

Field	Purpose	Updated when...	Size (Bytes)
Version	Block version number	You upgrade the software and it specifies a new version	4
hashPrevBlock	256-bit hash of the previous block header	A new block comes in	32
hashMerkleRoot	256-bit hash based on all of the transactions in the block	A transaction is accepted	32
Time	Current block timestamp as seconds since 1970-01-01T00:00 UTC	Every few seconds	4
Bits	Current target in compact format	The difficulty is adjusted	4

Fields















Field	Description
Version	The version of the block.
Previous Block Hash	The Block Hash of the block that this block is being built on top of. This is what "chains" the blocks together.
Merkle Root	All of the transactions in this block, hashed together. Basically provides a single-line summary of all the transactions in this block.
Time	When a miner is trying to mine this block, the <i>Unix</i> time at which this block header is being hashed is noted within the block header itself.
Bits	A shortened version of the Target.
Nonce	The field that miners change in order to try and get a hash of the block header (a Block Hash) that is below the Target.


Data Structure

Field	Size	Data
Version	4 bytes	Little-endian
Previous Block Hash	32 bytes	Little-endian
Merkle Root	32 bytes	Little-endian
Time	4 bytes	Little-endian
Bits	4 bytes	Little-endian
Nonce	4 bytes	Little-endian

(CoinBase)Tx structure:

Fields

Field	Data	Size	Description																								
Version	01000000 	4 bytes	Which version of transaction data structure we're using.																								
Input Count	01	Variable	Indicates the upcoming number of inputs.																								
Input(s)	<table><tr><th>Field</th><th>Data</th><th>Size</th><th>Description</th></tr><tr><td>TXID</td><td>796...efc </td><td>32 bytes</td><td>Refer to an existing transaction.</td></tr><tr><td>VOUT</td><td>01000000 </td><td>4 bytes</td><td>Select one of its outputs.</td></tr><tr><td>ScriptSig Size</td><td>6a</td><td>Variable</td><td>Indicates the upcoming size of the unlocking code.</td></tr><tr><td>ScriptSig</td><td>473...825</td><td></td><td>A script that unlocks the input.</td></tr><tr><td>Sequence</td><td>ffffffff </td><td>4 bytes</td><td></td></tr></table>			Field	Data	Size	Description	TXID	796...efc 	32 bytes	Refer to an existing transaction.	VOUT	01000000 	4 bytes	Select one of its outputs.	ScriptSig Size	6a	Variable	Indicates the upcoming size of the unlocking code.	ScriptSig	473...825		A script that unlocks the input.	Sequence	ffffffff 	4 bytes	
	Field	Data	Size	Description																							
	TXID	796...efc 	32 bytes	Refer to an existing transaction.																							
	VOUT	01000000 	4 bytes	Select one of its outputs.																							
	ScriptSig Size	6a	Variable	Indicates the upcoming size of the unlocking code.																							
	ScriptSig	473...825		A script that unlocks the input.																							
	Sequence	ffffffff 	4 bytes																								
Output Count	01	Variable	Indicates the upcoming number of outputs.																								
Output(s)	<table><tr><th>Field</th><th>Data</th><th>Size</th><th>Description</th></tr><tr><td>Value</td><td>4baf210000000000 </td><td>8 bytes</td><td>The value of the output in satoshis.</td></tr><tr><td>ScriptPubKey Size</td><td>19</td><td>Variable</td><td>Indicates the upcoming size of the locking code.</td></tr><tr><td>ScriptPubKey</td><td>76a9...88ac</td><td></td><td>A script that locks the output.</td></tr></table>			Field	Data	Size	Description	Value	4baf210000000000 	8 bytes	The value of the output in satoshis.	ScriptPubKey Size	19	Variable	Indicates the upcoming size of the locking code.	ScriptPubKey	76a9...88ac		A script that locks the output.								
	Field	Data	Size	Description																							
	Value	4baf210000000000 	8 bytes	The value of the output in satoshis.																							
	ScriptPubKey Size	19	Variable	Indicates the upcoming size of the locking code.																							
	ScriptPubKey	76a9...88ac		A script that locks the output.																							
Locktime	00000000 	4 bytes	Set a minimum block height or Unix time that this transaction can be included in.																								

- All of the data in a transaction is in [hexadecimal](#).
- The following icon indicates that the data is in [reverse byte order](#): 

All of the data in a transaction is in hexadecimal.

example for Coinbase

01000000 : Version

01 : input Count(vairable)

00 : TXID

ffffff : VOUT

45 : SIG_SIZE

03 : NUMBER OF BYTES BEING PUSH BASED ON BIP34 ec5906 == 416236

2f48616f4254432f53756e204368756e2059753a205a6875616e67205975616e2c2077696c6c20796f7520

6d61727279206d653f2f06fcc9cacc19c5f278560300 : SCRIPT_SIG

ffffff : SEQUENCE

01 : OUTPUT_COUNT

529c6d9800000000 : VALUE

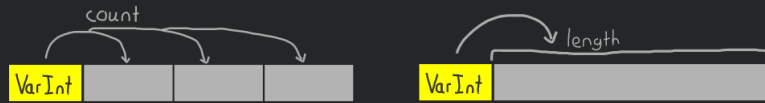
19 : SCRIPT_PUB_KEY SIZE

76a914bfd3ebb5485b49a6cf1657824623ead693b5a45888ac : SCRIPT_PUB_KEY

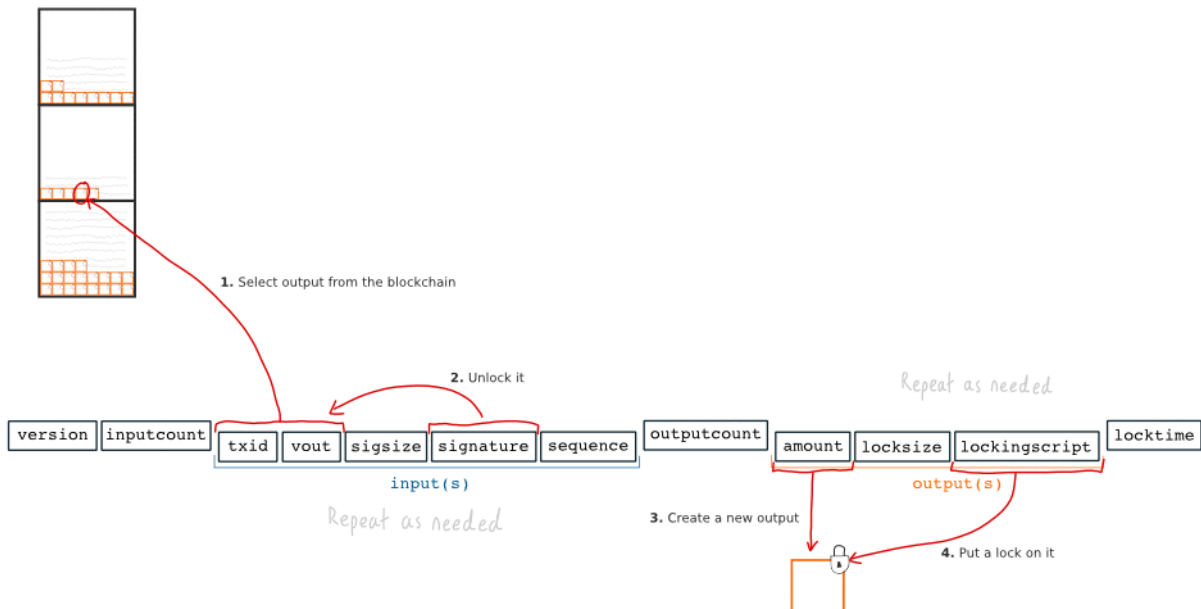
00000000 : LOCK_TIME

VARINT

A format for indicating the size of upcoming data.



Size	Example	Description
$\leq 0xfc$	12	
$\leq 0xffff$	fd1234	Prefix with fd, and the next 2 bytes is the VarInt (in little-endian).
$\leq 0xffffffff$	fe12345678	Prefix with fe, and the next 4 bytes is the VarInt (in little-endian).
$\leq 0xffffffffffffffff$	ff1234567890abcdef	Prefix with ff, and the next 8 bytes is the VarInt (in little-endian).



A TXID is always 32 bytes (64 characters) and hexadecimal

You get a TXID by hashing transaction data through SHA256 twice.

If you've just hashed some transaction data and want to search for a TXID in the blockchain, you have to search for it in reverse byte order.

If you have been given a TXID by your bitcoin wallet, it's probably already in its "searchable" format (reverse byte order).

A coinbase transaction must be 100 blocks deep before you can spend its outputs.

Messages in coinbase transactions:

Miners often use the scriptSig for strings of text. decode them (from Hex to ASCII) to read them.

The TXID is all f's. (We're not referencing an existing transaction)

The VOUT is all f's (the maximum hexadecimal value for this field). (Again, because we're not trying to refer to an existing output).

The scriptSig can actually contain any data you like.

Block Body:

```
0100000001000000000000000000000000000000000000000000000000000000000000000000ffffffffff1716383130313939333537536570656872417a6172646172ffffffff0140be4025000000001976a9146171ec3ad99dcc57f33929d9eb532ebcc4f6055788ac00000000
```

addresses

vanity prk: 7f60d9e49f3ee3829687c85e1c935d674c52c6a061f97991f5daaf1bec72f7c0
prk base58(wif): 92Z1sW9bbsUrANc1ZYAFSaVLiL7qsDJYtrokz8Zw99bZen592PL
vanity addr for sp: msep7tvnB1fq4k4cd73X8jsuADVx87vL
addr: msep7tvnB1fq4k4cd73X8jsuADVx87vL
pubk:
04d32efb2a388ad01c5208a3445eb710689e9c74941521c4c7788dcf9071cee537aecb5281c459db9896df
5c0e3c32c96a25e3abbe389ec453f1e4362ab5d402fa
prk: 7f60d9e49f3ee3829687c85e1c935d674c52c6a061f97991f5daaf1bec72f7c0

vanity prk: a22a031135fb9bb416ba65d0bdf706337eafbb4739f0c4ff375aeaf7d189301a
prk wif: 92pLRm8cK9FLAzLRnQxGTyCBwFiri5f65vmnCzYWMvJdCzt8tvC
vanity addr for sp: msepTuQpcFh93KffFp6bUfRcvcgPDu61zy

third one:
vanity prk: 06c85f016e260273cd8361f41e0660ab37e3c83219efd2c5979f5e8a6aeceb16
prk wif: 91duRCxLRu36WSb5UG9bL9MPuJVmz4BChv8ebVMaBz6dgpXHRyW
vanity addr for sp: mspmzEYcoSrZXzvXWwkE3gcyNHUPC9UqFq

mian one:
vanity prk: 0cb1a9189abc27f06f09dd9dad5ffab769d16cd252f5078c838d361db549acea
prk wif: 91gWQy7rGeTJZ2WGK35eisqs21xhk7WoJPtitPhGKQfjzeyoP9F
vanity addr for sp: mspyV97GWXj2ZgbiCmtAAhq1aKQkB43f2

wif: 92tyjesFurdTjfeJu4mZSGrKxVhEuZL1tu92HZCcoeTugDchhY
pbk: mspBonaqNwnZ7dwcN8Pe1WYVCgGwXtHakj
vanity prk: 5582245916e067cb3f4b40fc2da0f7a061c96e2234776ff1eeebd96e84053667

prk wif: 92EaMpD3ZCKs917dbxyXWFO42pesAfqL3HTZm9EqXYGnCefMSjq
vanity addr for sp: mspG86PFVaT1zLeeJYStBLNxC1SZUwUHB4
vanity prk: 8240f80f50cb9af74e529250a9db97d351d0fa87b836c3df47bbb0178f13d855

prk wif: 92aHKbBxhGp9iXuvZFRdZhUHxDYgEwrWuUM2HYTvDdrPyPKNFzb
vanity addr for sp: mspKR0DjG2avwMUjAYsG1z1tUSjAvz7Qtf

vanity prk: 94d0b3b6f49eb393672d2881fe1da86d3d6e792bd7167ecb06c36c9f28839d21
prk wif: 92iTSz8mrm1LEBjtDpUXhJXFRDhhPHXeyZK6Y9pbT6XbBUiLJvd
vanity addr for sp: mspgmvMnYGEWfkYRTuDMuHjySwn5ZBxm2

