# 作业一

# 马辰越 3180102882

## 1 代码实现

本次作业采用python进行SHA256算法的实现。

维基百科中关于SHA256算法的介绍、实现过程以及伪代码都有非常详尽的阐述和图解，详情可见https://en.wikipedia.org/wiki/SHA-2

主要步骤包括常量初始化、信息预处理、逻辑运算、计算信息摘要等，鉴于各类网站对此介绍已经十分详细，实在没有复制粘贴的必要，在此给出相关参考网站https://zhuanlan.zhihu.com/p/94619052，不再赘述。

下面给出算法实现的代码。

1. **import** struct
2. **def** out\_hex(list1):
3. **for** i **in** list1:
4. **print** ("%08x" % i)
5. **print** ("\n")
7. **def** rotate\_left(a, k):
8. k = k % 32
9. **return** ((a << k) & 0xFFFFFFFF) | ((a & 0xFFFFFFFF) >> (32 - k))
10. **def** rotate\_right(a, k):
11. k = k % 32
12. **return** (((a >> k) & 0xFFFFFFFF) | ((a & 0xFFFFFFFF) << (32 - k))) & 0xFFFFFFFF
14. **def** rotate\_shift(a,k):
15. k = k%32
16. **return** ((a >> k) & 0xFFFFFFFF);
18. **def** P\_0(X):
19. **return** (rotate\_right(X, 7)) ^ (rotate\_right(X, 18)) ^ (rotate\_shift(X,3))
21. **def** P\_1(X):
22. **return** (rotate\_right(X, 17)) ^ (rotate\_right(X, 19)) ^ (rotate\_shift(X,10))
24. IV="0x6A09E667 0xBB67AE85 0x3C6EF372 0xA54FF53A 0x510E527F 0x9B05688C 0x1F83D9AB 0x5BE0CD19"
26. IV=IV.replace("0x","")
27. IV=int(IV.replace(" ",""),16)
29. K = """0x428a2f98, 0x71374491, 0xb5c0fbcf, 0xe9b5dba5, 0x3956c25b, 0x59f111f1, 0x923f82a4, 0xab1c5ed5,
30. 0xd807aa98, 0x12835b01, 0x243185be, 0x550c7dc3, 0x72be5d74, 0x80deb1fe, 0x9bdc06a7, 0xc19bf174,
31. 0xe49b69c1, 0xefbe4786, 0x0fc19dc6, 0x240ca1cc, 0x2de92c6f, 0x4a7484aa, 0x5cb0a9dc, 0x76f988da,
32. 0x983e5152, 0xa831c66d, 0xb00327c8, 0xbf597fc7, 0xc6e00bf3, 0xd5a79147, 0x06ca6351, 0x14292967,
33. 0x27b70a85, 0x2e1b2138, 0x4d2c6dfc, 0x53380d13, 0x650a7354, 0x766a0abb, 0x81c2c92e, 0x92722c85,
34. 0xa2bfe8a1, 0xa81a664b, 0xc24b8b70, 0xc76c51a3, 0xd192e819, 0xd6990624, 0xf40e3585, 0x106aa070,
35. 0x19a4c116, 0x1e376c08, 0x2748774c, 0x34b0bcb5, 0x391c0cb3, 0x4ed8aa4a, 0x5b9cca4f, 0x682e6ff3,
36. 0x748f82ee, 0x78a5636f, 0x84c87814, 0x8cc70208, 0x90befffa, 0xa4506ceb, 0xbef9a3f7, 0xc67178f2 """
38. K=K.replace("\n","")
39. K=K.replace(", ","")
40. K=K.replace(" ","")
41. K=K.replace("0x","")
42. K=int(K,16)
44. a = []
45. **for** i **in** range(0, 8):
46. a.append(0)
47. a[i] = (IV >> ((7 - i) \* 32)) & 0xFFFFFFFF
48. IV = a
50. k = []
51. **for** i **in** range(0,64):
52. k.append(0)
53. k[i]= (K >> ((63-i)\*32)) & 0xFFFFFFFF
54. K = k
56. **def** CF(V\_i, B\_i):
57. W = []
58. **for** j **in** range(0, 16):
59. W.append(0)
60. unpack\_list = struct.unpack(">I", B\_i[j\*4:(j+1)\*4])
61. W[j] = unpack\_list[0]
62. **for** j **in** range(16, 64):
63. W.append(0)
64. s0 = P\_0(W[j-15])
65. s1 = P\_1(W[j-2])
66. W[j] = (W[j-16] + s0 + W[j-7] + s1) & 0xFFFFFFFF
67. str1 = "%08x" % W[j]
68. W\_1 = []

71. A, B, C, D, E, F, G, H = V\_i
73. **for** j **in** range(0, 64):
74. SS1 = rotate\_right(E,6) ^ rotate\_right(E,11) ^ rotate\_right(E,25)
75. SS0 = rotate\_right(A,2) ^ rotate\_right(A,13) ^ rotate\_right(A,22)
76. ch  = (E & F) ^ ((~E) & G)
77. temp1 = (H + SS1 + ch + K[j] + W[j]) & 0xFFFFFFFF
78. maj = (A & B) ^ (A & C) ^ ( B & C)
79. temp2 = (SS0 + maj) & 0xFFFFFFFF
81. H = G
82. G = F
83. F = E
84. E = (D + temp1)
85. D = C
86. C = B
87. B = A
88. A = (temp1 + temp2)

91. A = A & 0xFFFFFFFF
92. B = B & 0xFFFFFFFF
93. C = C & 0xFFFFFFFF
94. D = D & 0xFFFFFFFF
95. E = E & 0xFFFFFFFF
96. F = F & 0xFFFFFFFF
97. G = G & 0xFFFFFFFF
98. H = H & 0xFFFFFFFF

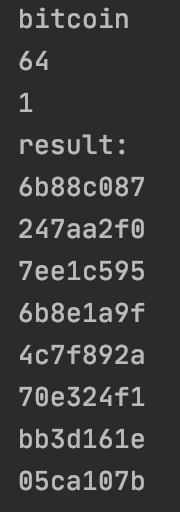
101. V\_i\_1 = []
102. V\_i\_1.append((A + V\_i[0]) & 0xFFFFFFFF)
103. V\_i\_1.append((B + V\_i[1]) & 0xFFFFFFFF)
104. V\_i\_1.append((C + V\_i[2]) & 0xFFFFFFFF)
105. V\_i\_1.append((D + V\_i[3]) & 0xFFFFFFFF)
106. V\_i\_1.append((E + V\_i[4]) & 0xFFFFFFFF)
107. V\_i\_1.append((F + V\_i[5]) & 0xFFFFFFFF)
108. V\_i\_1.append((G + V\_i[6]) & 0xFFFFFFFF)
109. V\_i\_1.append((H + V\_i[7]) & 0xFFFFFFFF)
111. **return** V\_i\_1
113. **def** hash\_msg(msg):
114. len1 = len(msg)
115. reserve1 = len1 % 64
116. msg1 = msg.encode() + struct.pack("B",128)
117. reserve1 = reserve1 + 1
118. **for** i **in** range(reserve1, 56):
119. msg1 = msg1 + struct.pack("B",0)
121. bit\_length = (len1) \* 8
122. bit\_length\_string = struct.pack(">Q", bit\_length)
123. msg1 = msg1 + bit\_length\_string
125. **print** (len(msg1) )
126. group\_count = int(len(msg1) / 64 )
127. **print**(group\_count)
129. m\_1 = B = []
130. **for** i **in** range(0, group\_count):
131. B.append(0)
132. B[i] = msg1[i\*64:(i+1)\*64]
134. V = []
135. V.append(0)
136. V[0] = IV
137. **for** i **in** range(0, group\_count):
138. V.append(0)
139. V[i+1] = CF(V[i], B[i])
141. **return** V[i+1]
143. **print** ("1313141151617118819101100191171615113121")
144. y = hash\_msg("1313141151617118819101100191171615113121")
145. **print** ("result: ")
146. out\_hex(y)

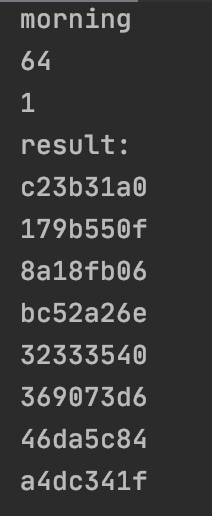
## 2 SHA256在比特币区块链中发挥的作用

## 2.1 单向函数

我们知道，要设计一个安全的哈希算法，就必须满足：通过输入可以很容易地计算输出，但是，反过来，通过输出无法反推输入，只能暴力穷举。

在SHA256算法中，通过常量初始化、信息预处理、逻辑运算、计算信息摘要等步骤，实现从输入到输出的变换。例如对“bitcoin”和“morning”进行算法处理可以得到如下结果：





但如果想通过输出来反推输入，则是极为困难的事情，事实上，想要根据上述结果反推输入，只能由计算机暴力穷举。

## 2.2 碰撞率低

一个安全的哈希算法还需要满足另一个条件：碰撞率低。

在批量生成测试用例时，我们发现很难出现相同的哈希结果。那么是因为我们测试用例的数量不够吗？

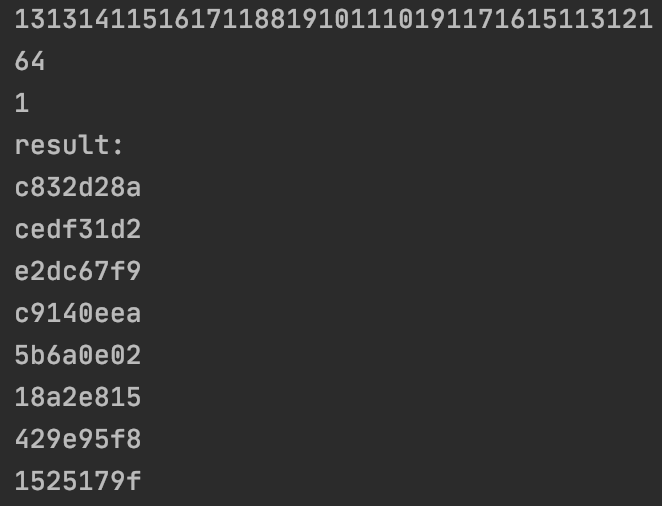
事实上，SHA-256的理论碰撞概率是：尝试2的130次方的随机输入，有99.8%的概率碰撞。注意2的130次方是一个非常大的数字，大约是1361万亿亿亿亿。以现有的计算机的计算能力，是不可能在短期内破解的。

由此我们得到两个推论：

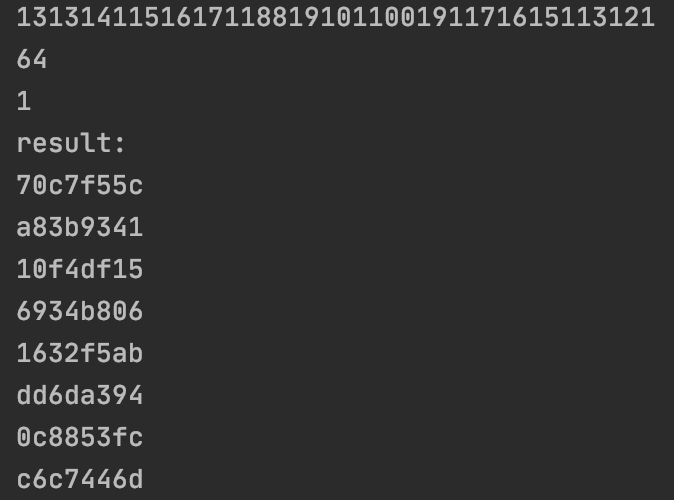
* 推论1：每个区块的哈希都是不一样的，可以通过哈希标识区块。
* 推论2：如果区块的内容变了，它的哈希一定会改变。

## 2.3 不可篡改

我们随机打出一串数字“1313141151617118819101110191171615113121”，它经过SHA256算法处理后的结果如下：



而我们仅仅修改其中的一位数字变为“13131411516171188191011**0**0191171615113121”，它经过SHA256算法处理后的结果如下：



可以看到，虽然在许多位中仅仅修改了一位数字，但是结果发生了巨大的变化，两次输出的结果可以说是毫不相关。

利用哈希算法的这种特性，我们可以使得交易数据依靠Merkle Hash确保无法修改，整个区块依靠Block Hash确保区块无法修改。

比如加密交易地址，它使用的是单向哈希。哈希树的顶部为顶部哈希（top hash），亦称根哈希（root hash）或主哈希（master hash）。它是通过并联两个子哈希来往树上爬直到找到根哈希。

作用：快速定位每笔交易；核实交易数据是否被篡改

同时该特性也使得修改一个区块的成本就已经非常非常高了，要修改后续所有区块，这个攻击者必须掌握全网51%以上的算力才行，这就进一步确保了区块链不可更改的特性。

## 2.4 编号长度固定

在上面测试用例的生成中我们发现，在经过SHA256算法后，生成结果一定是一个64位的16进制数字，达成了形式上的统一。

即不论原始数据有多少位，只要通过hash运算后，得到结果的长度都是一样的。

## 2.5 工作量证明机制

在挖矿过程中，区块头中有个参数叫随机数Nonce。

比特币挖矿过程使用SHA256哈希函数不断运算。挖矿就是重复计算区块头的哈希值，不断修改Nonce值，直到符合目标哈希值过程。找到一个哈希值小于指定的难度值，通过发布这个结果来证明自己完成的工作量。我们在前述分析中指出，哈希函数的结果无法预知，因此也没有特定模式快速算出哈希值。

我们熟知的比特币挖矿，采用的就是PoW机制。比特币网络通过调节计算难度，保证每一次竞争记账都需要全网矿工计算约10分钟，才能算出一个满足条件的结果。当有矿工找到了这个满足条件的结果，便可认定全网矿工完成了指定难度系数的工作量。

再往细了说，矿工们在挖一个新的区块时，必须对SHA-256密码散列函数进行运算，区块中的随机散列值以一个或多个0开始。随着0数目的上升，找到这个解所需要的工作量将呈指数增长，矿工通过反复尝试找到这个解。最先算出正确答案的节点可获得当前区块的记账权，同时获得新发行比特币的奖励。

比特币的工作量证明计算有一个重要的特点，**即是难于计算，却易于验证**，每次完成工作量证明需要大量的哈希计算，但是验证只需要一次哈希计算，使得各节点对于结果能很有效率的达成共识。

总而言之，是我们前述SHA256算法的一系列特点（单向函数、碰撞率低、雪崩效应、编号长度固定、谜题友好）证明了SHA256算法适用于比特币中。

## 2.6 总结SHA-256函数在比特币中应用

•工作量证明

•比特币公钥转换为公钥哈希

•交易的输入输出部分（Transaction id，交易脚本等）

•比特币区块头Previous Block Hash

•Merkle树