Problem 1

0.1 Explanation of Code

Given a single 64 bit number our goal was to calculate the number of ones in the binary representation of that same 64 bit number. First I initialize the lookup table, which will be revisited in just a moment. Next, I define the function countOnes(). I initialize the total number of ones to zero and begin a while loop that will allow us to count our ones. The condition n != 0 terminates as we continue to right shift n by four bits each loop iteration (line 14). Upon each loop iteration we mask the four most right bits, lookup the corresponding number of ones associated with the base 10 digit for those four bits, and then shift our input number by four to the right. Eventually the loop terminates and we are left with the correct number of ones from the input number's binary representation.

```
1 # table for lookup
2 table = {0: 0, 1: 1, 2: 1, 3: 2, 4: 1, 5: 2, 6: 2, 7: 3, 8: 1, 9: 2,
      0xa: 2, 0xb: 3, 0xc: 2, 0xd: 3, 0xe: 3, 0xf: 4}
3
4 # function that uses bitwise operations to determine number of 1
     bits in number
5 def countOnes(n):
6   total = 0
```

```
7
8   while(n != 0):
9     #mask first four bits
10    temp = n & 0xf
11    #lookup num of ones and add to total
12    total += table[temp]
13    #shift binary right by 4 bits
14    n = n >> 4
15
16  return total
```

Algorithm 1: **Ones bit counter in Python 3.9**

Problem 2

0.2 Explanation of Code

In this problem we are asked to implement a a function that when given a 64 bit number, returns that number with all bits inverted. I start my code by initializing a lookup table for efficiency. I figured if I approach the problem from 4 bits at a time the lookup table only needs entries from 0 - 15 (base 16). Then I define the actual function invert(). First, I initialize my final digit and the stack where I will push all my conversions onto. The first while loop

examines all of the input digit n and converts each right most 4 bits using masking. The second loop then takes the conversions that have been pushed onto the stack and assembles them back together in reverse order by ORing. Then I return the newly built digit.

In terms of efficiency, I am sure there is a more efficient function that only requires one loop and more bit wise operation; however, in my solution we still use only bit wise operations and the .append() action on the stack (which is still a very low overhead operation). Since my implementation works and uses bit wise operation, I would say that this is a successful implementation.

```
# table for lookup
table = {0: 0xf, 1: 0xe, 2: 0xd, 3: 0xc, 4: 0xb, 5: 0xa, 6: 9, 7: 8,
    8: 7, 9: 6, 0xa: 5, 0xb: 4, 0xc: 3, 0xd: 2, 0xe: 1, 0xf: 0}

def invert(n):
  final = 0
  stack = []

  while(n != 0):
    # mask 4 right-most bits
    temp = n & 0xf
    # lookup inverted version and 'push onto stack'
    stack.append(table[temp])
```

```
13      # right shift by 4 bits
14      n = n >> 4
15
16  for i in range(len(stack) - 1, -1, -1):
17      # left shift by 4 bits
18      final = final << 4
19      # OR stack element with final digit
20      final = final | stack[i]
21
22  return final
```

Algorithm 2: **Bit inverter in Python 3.9**

Problem 3

0.3 Explanation of Code

In this problem we are given a Sudoku board with a certain arrangement. From this board we must determine whether the solution given is valid or not. I approached this problem from a brute force approach (as I couldn't figure out the binary trick). First, I check all rows updating a table if a number has been seen or not. Second, I do the same for columns. Lastly, I check each row of squares. So basically I check a 3 by 9 board each time.

Clearly this is rather inefficient, but it does work. I hope to improve upon this design by you using binary operations.

```
1
2  table = [0,0,0,0,0,0,0,0,0]
3
4  # function that checks the validity of a sudoku solution
5  def checkSol(board):
6    global table
7
8    # check columns
9    for i in range(0, 9):
10     for j in range(0, 9):
11       currNum = board[j][i]
12       # check if currNum has already been seen
13       if(currNum == 0):
14         pass
15       else:
16         if(table[currNum - 1] & 1):
17           return False
18         else:
19           # show that we have seen the number
```

```
20          table[currNum - 1] = 1
21      # reset the table to all zeros
22      table = reset(table)
23
24  # check rows
25  for i in range(0, 9):
26      for j in range(0, 9):
27          currNum = board[i][j]
28              # check if currNum has already been seen
29          if(currNum == 0):
30              pass
31          else:
32              if(table[currNum - 1] & 1):
33                  return False
34              else:
35                  # show that we have seen the number
36                  table[currNum - 1] = 1
37      # reset the table to all zeros
38      table = reset(table)
39
40  # check house 1 (3x3 subgrid)
41
```

```
42  for i in range(3):
43    for j in range(3):
44      currNum = board[i][j]
45        # check if currNum has already been seen
46      if(currNum == 0):
47        pass
48      else:
49        if(table[currNum - 1] == 1):
50          return False
51        else:
52          # show that we have seen the number
53          table[currNum - 1] = 1
54
55  # reset table to all zeros
56  table = reset(table)
57
58  # check house 2 (3x3 subgrid)
59
60  for i in range(3,6,1):
61    for j in range(3):
62      currNum = board[j][i]
63        # check if currNum has already been seen
```

```
64      if(currNum == 0):
65        pass
66      else:
67        if(table[currNum - 1] == 1):
68          return False
69        else:
70          # show that we have seen the number
71          table[currNum - 1] = 1
72
73  # reset table to all zeros
74  table = reset(table)
75
76  # check house 3 (3x3 subgrid)
77
78  for i in range(6,9,1):
79    for j in range(3):
80      currNum = board[j][i]
81        # check if currNum has already been seen
82      if(currNum == 0):
83        pass
84      else:
85        if(table[currNum - 1] == 1):
```

```
86            return False
87          else:
88            # show that we have seen the number
89            table[currNum - 1] = 1
90
91  # reset table to all zeros
92  table = reset(table)
93
94  # check house 4 (3x3 subgrid)
95
96  for i in range(3):
97    for j in range(3,6,1):
98      currNum = board[j][i]
99        # check if currNum has already been seen
100     if(currNum == 0):
101       pass
102     else:
103       if(table[currNum - 1] == 1):
104         return False
105       else:
106         # show that we have seen the number
107         table[currNum - 1] = 1
```

```
108
109 # reset table to all zeros
110 table = reset(table)
111
112
113 # check house 5 (3x3 subgrid)
114
115 for i in range(3,6,1):
116   for j in range(3,6,1):
117     currNum = board[j][i]
118       # check if currNum has already been seen
119     if(currNum == 0):
120       pass
121     else:
122       if(table[currNum - 1] == 1):
123         return False
124       else:
125         # show that we have seen the number
126         table[currNum - 1] = 1
127
128 # reset table to all zeros
129 table = reset(table)
```

```
130
131 # check house 6 (3x3 subgrid)
132
133 for i in range(6,9,1):
134   for j in range(3,6,1):
135     currNum = board[j][i]
136       # check if currNum has already been seen
137     if(currNum == 0):
138       pass
139     else:
140       if(table[currNum - 1] == 1):
141         return False
142       else:
143         # show that we have seen the number
144         table[currNum - 1] = 1
145
146 # reset table to all zeros
147 table = reset(table)
148
149
150 # check house 7 (3x3 subgrid)
151
```

```
152 for i in range(3):
153   for j in range(6,9,1):
154     currNum = board[j][i]
155       # check if currNum has already been seen
156     if(currNum == 0):
157       pass
158     else:
159       if(table[currNum - 1] == 1):
160         return False
161       else:
162         # show that we have seen the number
163         table[currNum - 1] = 1
164
165 # reset table to all zeros
166 table = reset(table)
167
168
169 # check house 8 (3x3 subgrid)
170
171 for i in range(3,6,1):
172   for j in range(6,9,1):
173     currNum = board[j][i]
```

```
174         # check if currNum has already been seen
175       if(currNum == 0):
176         pass
177       else:
178         if(table[currNum - 1] == 1):
179           return False
180         else:
181           # show that we have seen the number
182           table[currNum - 1] = 1
183
184 # reset table to all zeros
185 table = reset(table)
186
187   # check house 8 (3x3 subgrid)
188
189 for i in range(6,9,1):
190   for j in range(6,9,1):
191     currNum = board[j][i]
192         # check if currNum has already been seen
193     if(currNum == 0):
194       pass
195     else:
```

```
196         if(table[currNum - 1] == 1):
197           return False
198         else:
199           # show that we have seen the number
200           table[currNum - 1] = 1
201
202   # reset table to all zeros
203   table = reset(table)
204
205   #partial solution is valid
206   return True
207
208 # function that resets the lookup table
209 def reset(table):
210   clean = [0,0,0,0,0,0,0,0,0]
211
212   for i in range(0, len(table)):
213     clean[i] = 0 & table[i]
214
215   return clean
216
217
```

```
testBoard = [[0,5,0,0,0,8,0,4,0],
        [0,4,0,3,0,0,0,7,0],
        [0,3,1,7,2,0,8,9,0],
        [3,0,0,0,0,0,7,8,0],
              [0,0,5,0,0,0,1,0,0],
              [0,6,2,0,0,0,0,0,3],
              [0,2,6,0,4,7,9,3,0],
              [0,8,0,0,0,6,0,2,0],
              [0,9,0,8,0,0,0,1,0]]

print(checkSol(testBoard))
```

Algorithm 3: **Sudoku solution confirm in Python 3.9**