Name: Daniel Throop
MUID: 006165173

Programming Assignment 2

COSC 3100
February 8, 2021

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
```

```

  while(n != 0):
    #mask first four bits
    temp = n & 0xf
    #lookup num of ones and add to total
    total += table[temp]
    #shift binary right by 4 bits
    n = n >> 4

  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.

```python
# 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])
    # right shift by 4 bits
    n = n >> 4

  for i in range(len(stack) - 1, -1, -1):
    # left shift by 4 bits
    final = final << 4
    # OR stack element with final digit
    final = final | stack[i]

  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.

```python

table = [0,0,0,0,0,0,0,0,0]

# function that checks the validity of a sudoku solution
def checkSol(board):
  global table

  # check columns
  for i in range(0, 9):
    for j in range(0, 9):
      currNum = board[j][i]
      # check if currNum has already been seen
      if(currNum == 0):
        pass
      else:
        if(table[currNum - 1] & 1):
          return False
        else:
          # show that we have seen the number
          table[currNum - 1] = 1
    # reset the table to all zeros
    table = reset(table)

  # check rows
  for i in range(0, 9):
    for j in range(0, 9):
      currNum = board[i][j]
        # check if currNum has already been seen
      if(currNum == 0):
        pass
      else:
        if(table[currNum - 1] & 1):
          return False
        else:
          # show that we have seen the number
          table[currNum - 1] = 1
    # reset the table to all zeros
    table = reset(table)

  # check house 1 (3x3 subgrid)

```

```
for i in range(3):
  for j in range(3):
    currNum = board[i][j]
      # check if currNum has already been seen
    if(currNum == 0):
      pass
    else:
      if(table[currNum - 1] == 1):
        return False
      else:
        # show that we have seen the number
        table[currNum - 1] = 1

# reset table to all zeros
table = reset(table)

# check house 2 (3x3 subgrid)

for i in range(3,6,1):
  for j in range(3):
    currNum = board[j][i]
      # check if currNum has already been seen
    if(currNum == 0):
      pass
    else:
      if(table[currNum - 1] == 1):
        return False
      else:
        # show that we have seen the number
        table[currNum - 1] = 1

# reset table to all zeros
table = reset(table)

# check house 3 (3x3 subgrid)

for i in range(6,9,1):
  for j in range(3):
    currNum = board[j][i]
      # check if currNum has already been seen
    if(currNum == 0):
      pass
    else:
      if(table[currNum - 1] == 1):
        return False
      else:
        # show that we have seen the number
        table[currNum - 1] = 1

# reset table to all zeros
table = reset(table)

# check house 4 (3x3 subgrid)

for i in range(3):
  for j in range(3,6,1):
    currNum = board[j][i]
      # check if currNum has already been seen
    if(currNum == 0):
      pass
    else:
      if(table[currNum - 1] == 1):
        return False
      else:
        # show that we have seen the number
        table[currNum - 1] = 1
```

```

# reset table to all zeros
table = reset(table)


# check house 5 (3x3 subgrid)

for i in range(3,6,1):
  for j in range(3,6,1):
    currNum = board[j][i]
      # check if currNum has already been seen
    if(currNum == 0):
      pass
    else:
      if(table[currNum - 1] == 1):
        return False
      else:
        # show that we have seen the number
        table[currNum - 1] = 1

# reset table to all zeros
table = reset(table)
```

```

# check house 6 (3x3 subgrid)

for i in range(6,9,1):
  for j in range(3,6,1):
    currNum = board[j][i]
      # check if currNum has already been seen
    if(currNum == 0):
      pass
    else:
      if(table[currNum - 1] == 1):
        return False
      else:
        # show that we have seen the number
        table[currNum - 1] = 1

# reset table to all zeros
table = reset(table)



# check house 7 (3x3 subgrid)

```

```
for i in range(3):
  for j in range(6,9,1):
    currNum = board[j][i]
      # check if currNum has already been seen
    if(currNum == 0):
      pass
    else:
      if(table[currNum - 1] == 1):
        return False
      else:
        # show that we have seen the number
        table[currNum - 1] = 1

# reset table to all zeros
table = reset(table)



# check house 8 (3x3 subgrid)

for i in range(3,6,1):
  for j in range(6,9,1):
    currNum = board[j][i]
```

```
      # check if currNum has already been seen
    if(currNum == 0):
      pass
    else:
      if(table[currNum - 1] == 1):
        return False
      else:
        # show that we have seen the number
        table[currNum - 1] = 1

# reset table to all zeros
table = reset(table)

  # check house 8 (3x3 subgrid)

for i in range(6,9,1):
  for j in range(6,9,1):
    currNum = board[j][i]
      # check if currNum has already been seen
    if(currNum == 0):
      pass
    else:
        if(table[currNum - 1] == 1):
          return False
        else:
          # show that we have seen the number
          table[currNum - 1] = 1

  # reset table to all zeros
  table = reset(table)

  #partial solution is valid
  return True

# function that resets the lookup table
def reset(table):
  clean = [0,0,0,0,0,0,0,0,0]

  for i in range(0, len(table)):
    clean[i] = 0 & table[i]

  return clean
```

```python
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**