Parity bit:

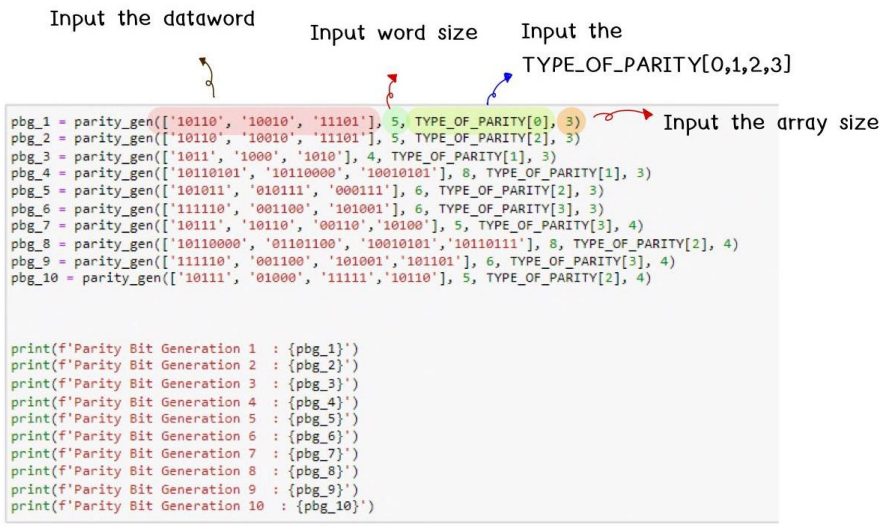
Function for Generate Party bit and Parity Checker

```
def Check_Sum_Even(input: np.ndarray) -> bool:

 # Input: an array of bit
 # Caculate the sum of bits given and consider whether it is EVEN number or not.
 # Output: Boolean whether it is EVEN or not (TRUE = EVEN / FALSE = ODD)

  count = 0
  for bin in input:
      count = count + int(bin)
  return bool(count % 2 == 0) # Return the boolean whether the number is even or not?


def addBit(parity_type: str):

   #Input: type of parity bit
   #Calculate the key for odd and even parity bit
   # Depend on TYPE

    first = '0' if parity_type in (
        TYPE_OF_PARITY[0], TYPE_OF_PARITY[2]) else '1'

    second = '1' if parity_type in (
        TYPE_OF_PARITY[0], TYPE_OF_PARITY[2]) else '0'

    return [first, second] #Output: Array of key of parity bit

def gen_CheckBit(dataword: np.ndarray, parity_type: str, axis='x') -> np.ndarray:

    """
      Input: dataword, type of parity bit, and axis of parity bit
      Generate the parity bit and append into original dataword given
      Output: the dataword combined with parity bit
  """

    # Get key that depends on the parity type
    [first, second] = addBit(parity_type)

    # if two dimension and want to generate the parity bit for y axis, transpose them to x
    if axis == 'y':
        dataword = dataword.T

    # Append generated parity bit to original array of bit
    dataword = np.array( # even or not if type 0 2 , even add 0 else 1
      [np.append(w, [first if Check_Sum_Even(w) else second]) for w in dataword])

    if axis == 'y':
        return dataword.T # transpose back

    return dataword

def VerifyBits(c_word, parity_type):
  """
  Input: codeword, type of parity bit
  For Check whether the bit has changed or not
  Output: Validity of parity bit
  """
  [first, second] = addBit(parity_type) # Get key that depends on the parity type

  for word in c_word:
      keyCheck = first if Check_Sum_Even(word[0:-1]) else second
      if word[-1] != keyCheck: # If the key check is not equal the key for codeword -> the codeword has been damaged
          return 'FAILED'
  return 'PASSED'

def mergeStringFromList(arr: list): # Input: Array of array of bit or string

  # This function will Merge a bit or string in array to be 1 string

  return [''.join(l) for l in (d.tolist() for d in arr)] # Output: Array of string_
```

Generator: codeword = parity_gen(dataword, word_size, parity_type, array_size)

```
def parity_gen(dataword: list, word_size: int, parity_type: str, array_size: int):

#codeword: bit string of any size up to word_size
#word_size: size of each dataword where word_size ≥ 5

  d_word = np.array([np.array(list(f)) for f in dataword]) # Create Numpy Array from the Dataword Array

  d_word = gen_CheckBit(d_word, parity_type) # Generate Bit in x axis

  # If the word is less than data word size provided in parameters, add 0 until it reaches data word size given.
  for i in range(len(d_word)):
      while d_word[i].size < word_size:
          d_word[i] = np.append(d_word[i], [0]) # Add 0 in word array in Dataword Numpy array


  if parity_type in TYPE_OF_PARITY[0:2]:
      return mergeStringFromList(d_word) # If type is 1D, return to codeword

  d_word = gen_CheckBit(d_word, parity_type, axis='y') # Generate Bit in y axis
  return mergeStringFromList(d_word) # Return codeword
```

How to run the code for the generator



Test:

```
pbg_1 = parity_gen(['10110', '10010', '11101'], 5, TYPE_OF_PARITY[0], 3)
pbg_2 = parity_gen(['10110', '10010', '11101'], 5, TYPE_OF_PARITY[2], 3)
pbg_3 = parity_gen(['1011', '1000', '1010'], 4, TYPE_OF_PARITY[1], 3)
pbg_4 = parity_gen(['10110101', '10110000', '10010101'], 8, TYPE_OF_PARITY[1], 3)
pbg_5 = parity_gen(['101011', '010111', '000111'], 6, TYPE_OF_PARITY[2], 3)
pbg_6 = parity_gen(['111110', '001100', '101001'], 6, TYPE_OF_PARITY[3], 3)
pbg_7 = parity_gen(['10111', '10110', '00110','10100'], 5, TYPE_OF_PARITY[3], 4)
pbg_8 = parity_gen(['10110000', '01101100', '10010101','10110111'], 8, TYPE_OF_PARITY[2], 4)
pbg_9 = parity_gen(['111110', '001100', '101001','101101'], 6, TYPE_OF_PARITY[3], 4)
pbg_10 = parity_gen(['10111', '01000', '11111','10110'], 5, TYPE_OF_PARITY[2], 4)



print(f'Parity Bit Generation 1  : {pbg_1}')
print(f'Parity Bit Generation 2  : {pbg_2}')
print(f'Parity Bit Generation 3  : {pbg_3}')
print(f'Parity Bit Generation 4  : {pbg_4}')
print(f'Parity Bit Generation 5  : {pbg_5}')
print(f'Parity Bit Generation 6  : {pbg_6}')
print(f'Parity Bit Generation 7  : {pbg_7}')
print(f'Parity Bit Generation 8  : {pbg_8}')
print(f'Parity Bit Generation 9  : {pbg_9}')
print(f'Parity Bit Generation 10  : {pbg_10}')
```

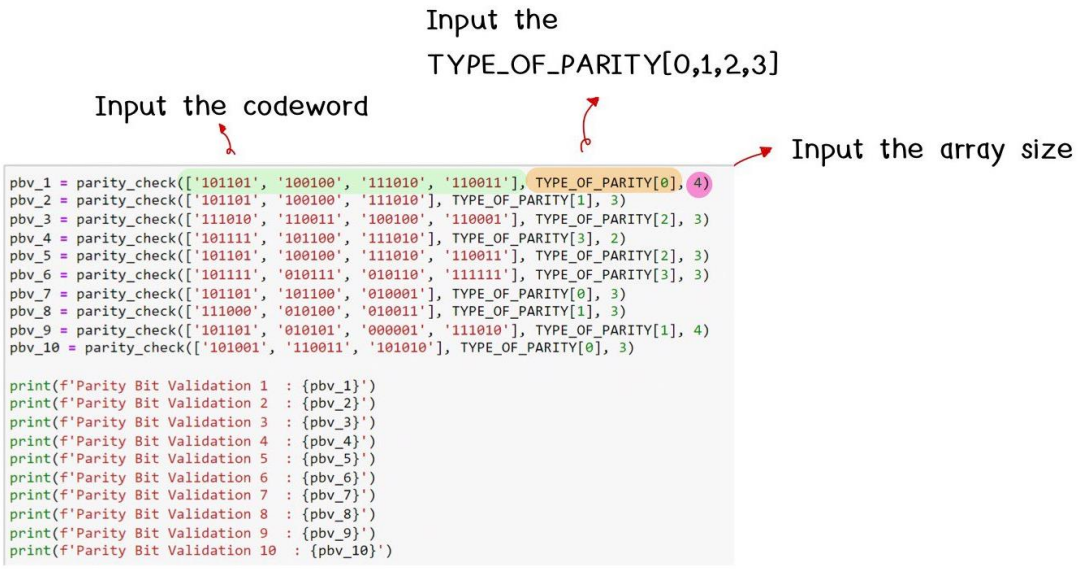
Result:

```
Parity Bit Generation 1  : ['101101', '100100', '111010']
Parity Bit Generation 2  : ['101101', '100100', '111010', '110011']
Parity Bit Generation 3  : ['10110', '10000', '10101']
Parity Bit Generation 4  : ['101101010', '101100000', '100101011']
Parity Bit Generation 5  : ['1010110', '0101110', '0001111', '1110111']
Parity Bit Generation 6  : ['1111100', '0011001', '1010010', '1001000']
Parity Bit Generation 7  : ['101111', '101100', '001101', '101001', '011000']
Parity Bit Generation 8  : ['101100001', '011011000', '100101010', '101101110', '111111101']
Parity Bit Generation 9  : ['1111100', '0011001', '1010010', '1011011', '0010011']
Parity Bit Generation 10  : ['101110', '010001', '111111', '101101', '101101']
```

Checker: validity = parity_check(codeword, parity_type, size)

```
def parity_check(codeword: list, parity_type: str, size: int):  # Parity Checker

  checkValid = 'PASSED' # Initialize Check Status

  c_word = np.array([np.array(list(w)) for w in codeword]) # Create Numpy Array from the Codeword Array

  checkValid = VerifyBits(c_word, parity_type) # Verify Checkbit in X axis

  if parity_type in TYPE_OF_PARITY[0:2] or checkValid == 'FAILED':
      return checkValid

  return VerifyBits(c_word.T, parity_type) # Verify Checkbit in Y axis
```

How to run the Parity_check:



Test:

```
pbv_1 = parity_check(['101101', '100100', '111010', '110011'], TYPE_OF_PARITY[0], 4)
pbv_2 = parity_check(['101101', '100100', '111010'], TYPE_OF_PARITY[1], 3)
pbv_3 = parity_check(['111010', '110011', '100100', '110001'], TYPE_OF_PARITY[2], 3)
pbv_4 = parity_check(['101111', '101100', '111010'], TYPE_OF_PARITY[3], 2)
pbv_5 = parity_check(['101101', '100100', '111010', '110011'], TYPE_OF_PARITY[2], 3)
pbv_6 = parity_check(['101111', '010111', '010110', '111111'], TYPE_OF_PARITY[3], 3)
pbv_7 = parity_check(['101101', '101100', '010001'], TYPE_OF_PARITY[0], 3)
pbv_8 = parity_check(['111000', '010100', '010011'], TYPE_OF_PARITY[1], 3)
pbv_9 = parity_check(['101101', '010101', '000001', '111010'], TYPE_OF_PARITY[1], 4)
pbv_10 = parity_check(['101001', '110011', '101010'], TYPE_OF_PARITY[0], 3)

print(f'Parity Bit Validation 1  : {pbv_1}')
print(f'Parity Bit Validation 2  : {pbv_2}')
print(f'Parity Bit Validation 3  : {pbv_3}')
print(f'Parity Bit Validation 4  : {pbv_4}')
print(f'Parity Bit Validation 5  : {pbv_5}')
print(f'Parity Bit Validation 6  : {pbv_6}')
print(f'Parity Bit Validation 7  : {pbv_7}')
print(f'Parity Bit Validation 8  : {pbv_8}')
print(f'Parity Bit Validation 9  : {pbv_9}')
print(f'Parity Bit Validation 10  : {pbv_10}')
```

Output:

```
Parity Bit Validation 1  : PASSED
Parity Bit Validation 2  : FAILED
Parity Bit Validation 3  : FAILED
Parity Bit Validation 4  : FAILED
Parity Bit Validation 5  : PASSED
Parity Bit Validation 6  : FAILED
Parity Bit Validation 7  : FAILED
Parity Bit Validation 8  : FAILED
Parity Bit Validation 9  : FAILED
Parity Bit Validation 10  : FAILED
```

CRC:

```
def xor(x, y):
    res = []
    for i in range(1, len(y)):
        #xor logic, same = 0, difference = 1
        if x[i] == y[i]:
            res.append('0')
        else:
            res.append('1')
    return ''.join(res)

#modulo-2 division
def mod2div(dividend, divisor):
    j = len(divisor)
    k = dividend[0 : j]
    while j < len(dividend):
        if k[0] == '1':
            k = xor(divisor, k) + dividend[j]
        else:
            #if the left bit is "0", use "0s" divisor
            k = xor('0'*j, k) + dividend[j]
        j += 1
    #Fix Index out of bound error
    if k[0] == '1':
        k = xor(divisor, k)
    else:
        k = xor('0'*j, k)
    return k

def CRC_gen(dataword, word_size, CRC_type): #Encoder
    if(word_size != len(dataword)): return "Please Re-Enter"
    if(CRC_type == "CRC-32"): binary = "10000010011000001000111011011011 1"
    elif(CRC_type == "CRC-24"): binary = "11000000001010001000000001"
    elif(CRC_type == "CRC-16"): binary = "1100000000000101"
    elif(CRC_type == "Reversed CRC-16"): binary = "1010000000000011"
    elif(CRC_type == "CRC-8"): binary = "111010101"
    elif(CRC_type == "CRC-4"): binary = "11111"
    key = len(binary)
    appended_dataword = dataword + '0'*(key-1)
    remain = mod2div(appended_dataword, binary)
    codeword = dataword + remain
    return codeword
```

```
def CRC_check(dataword, CRC_type): #Decoder
    if(CRC_type == "CRC-32"): binary = "10000010011000001000111011011011 1"
    elif(CRC_type == "CRC-24"): binary = "11000000001010001000000001"
    elif(CRC_type == "CRC-16"): binary = "1100000000000101"
    elif(CRC_type == "Reversed CRC-16"): binary = "1010000000000011"
    elif(CRC_type == "CRC-8"): binary = "111010101"
    elif(CRC_type == "CRC-4"): binary = "11111"
    key = len(binary)
    appended_dataword = dataword + '0'*(key-1)
    remain = mod2div(appended_dataword, binary)
    remain = int(remain)
    if(remain == 0): return 1
    else: return 0
```

Test:

```
test1 = CRC_gen("101011", 6, "CRC-8")
test2 = CRC_gen("1010", 4, "CRC-4")
test3 = CRC_gen("1100101", 7, "CRC-8")
test4 = CRC_gen("1011010", 7, "CRC-16")
test5 = CRC_gen("101001", 6, "Reversed CRC-16")
test6 = CRC_gen("100110010", 9, "CRC-24")
test7 = CRC_gen("11100101", 8, "CRC-16")
test8 = CRC_gen("1010010010", 10, "CRC-16")
test9 = CRC_gen("1010", 4, "CRC-4")
test10 = CRC_gen("1010010010100010", 16, "CRC-32")
print("codeword 1: ", test1)
print("codeword 2: ", test2)
print("codeword 3: ", test3)
print("codeword 4: ", test4)
print("codeword 5: ", test5)
print("codeword 6: ", test6)
print("codeword 7: ", test7)
print("codeword 8: ", test8)
print("codeword 9: ", test9)
print("codeword 10: ", test10)
print("check 1: ", CRC_check(test1, "CRC-8"))
print("check 2: ", CRC_check(test2, "CRC-4"))
print("check 3: ", CRC_check(test3, "CRC-8"))
print("check 4: ", CRC_check(test4, "CRC-16"))
print("check 5: ", CRC_check(test5, "Reversed CRC-16"))
print("check 6: ", CRC_check(test6, "CRC-24"))
print("check 7: ", CRC_check(test7, "CRC-16"))
print("check 8: ", CRC_check(test8, "CRC-16"))
print("check 9: ", CRC_check(test9, "CRC-4"))
print("check 10: ", CRC_check(test10, "CRC-32"))
```

Output:

```
PS C:\Python> python -u "c:\Python\Computer Data\q2.py"
codeword 1:  10101100100111
codeword 2:  10100101
codeword 3:  110010100010010
codeword 4:  10110100000000111011100
codeword 5:  10100101000000001100011
codeword 6:  10011001001101011000111011011100
codeword 7:  11100101100000100101101
codeword 8:  10100100100000111101101100
codeword 9:  10100101
codeword 10:  1010010010100010000011111101001101010001001110111
check 1:  1
check 2:  1
check 3:  1
check 4:  1
check 5:  1
check 6:  1
check 7:  1
check 8:  1
check 9:  1
check 10:  1
```

Generator: codeword = CRC_gen(dataword, word_size, CRC_type)

```
def CRC_gen(dataword, word_size, CRC_type): #Encoder
    if(word_size != len(dataword)): return "Please Re-Enter"
    if(CRC_type == "CRC-32"): binary = "100000100110000010001110110110111"
    elif(CRC_type == "CRC-24"): binary = "110000000010100010000001"
    elif(CRC_type == "CRC-16"): binary = "11000000000000101"
    elif(CRC_type == "Reversed CRC-16"): binary = "10100000000000011"
    elif(CRC_type == "CRC-8"): binary = "111010101"
    elif(CRC_type == "CRC-4"): binary = "11111"
    key = len(binary)
    appended_dataword = dataword + '0'*(key-1)
    remain = mod2div(appended_dataword, binary)
    codeword = dataword + remain
    return codeword
```

How to run CRC_gen

```
test1 = CRC_gen("101011", 6, "CRC-8")
test2 = CRC_gen("1010", 4, "CRC-4")
test3 = CRC_gen("1100101", 7, "CRC-8")
test4 = CRC_gen("1011010", 7, "CRC-16")
test5 = CRC_gen("101001", 6, "Reversed CRC-16")
test6 = CRC_gen("100110010", 9, "CRC-24")
test7 = CRC_gen("11100101", 8, "CRC-16")
test8 = CRC_gen("1010010010", 10, "CRC-16")
test9 = CRC_gen("1010", 4, "CRC-4")
test10 = CRC_gen("1010010010100010", 16, "CRC-32")
```

- input dataword
- input word_size according to dataword length
- input CRC-type

Checker: validity = CRC_check(codeword, CRC-type)

```
def CRC_check(dataword, CRC_type): #Decoder
    if(CRC_type == "CRC-32"): binary = "100000100110000010001110110110111"
    elif(CRC_type == "CRC-24"): binary = "110000000010100010000001"
    elif(CRC_type == "CRC-16"): binary = "1100000000000101"
    elif(CRC_type == "Reversed CRC-16"): binary = "10100000000000011"
    elif(CRC_type == "CRC-8"): binary = "111010101"
    elif(CRC_type == "CRC-4"): binary = "11111"
    key = len(binary)
    appended_dataword = dataword + '0'*(key-1)
    remain = mod2div(appended_dataword, binary)
    remain = int(remain)
    if(remain == 0): return 1
    else: return 0
```

How to run CRC_check

```
print("check 1: ", CRC_check(test1, "CRC-8"))
print("check 2: ", CRC_check(test2, "CRC-4"))
print("check 3: ", CRC_check(test3, "CRC-8"))
print("check 4: ", CRC_check(test4, "CRC-16"))
print("check 5: ", CRC_check(test5, "Reversed CRC-16"))
print("check 6: ", CRC_check(test6, "CRC-24"))
print("check 7: ", CRC_check(test7, "CRC-16"))
print("check 8: ", CRC_check(test8, "CRC-16"))
print("check 9: ", CRC_check(test9, "CRC-4"))
print("check 10: ", CRC_check(test10, "CRC-32"))
```

- input dataword
- input CRC-type

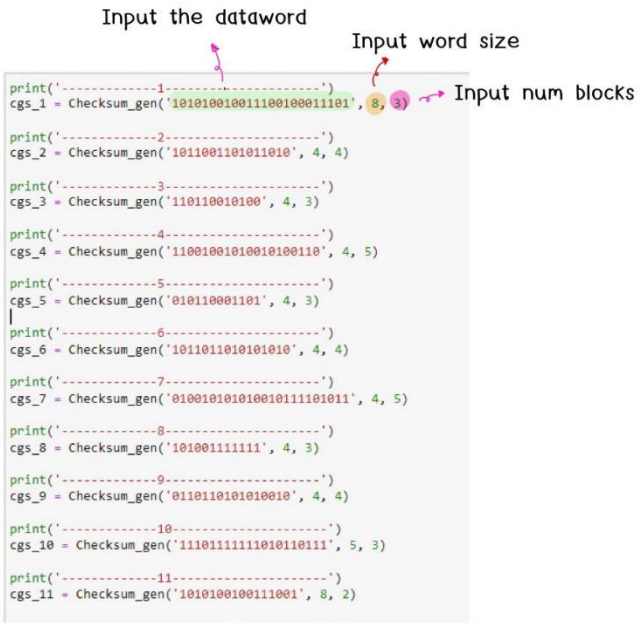
CheckSum:

Generator: codeword = Checksum_gen(dataword, word_size, num_blocks)

```
# This function created for inverse the dataword
def bi_inverse(dataword):
    result = ''
    for i in dataword:
        if i == '1':
            result = result + '0'
        else:
            result = result + '1'
    return result
```

```
def Checksum_gen(dataword, word_size, num_blocks):

    dataw = []
    w = list(dataword)

    #Split dataword into [...,...,...] following the numblock

    for i in range(num_blocks): # i * word_size = start assume that word_size = 5 > 0:5, 5:10
        dataw.append(''.join(w[i * word_size: i * word_size + word_size]))
    print("Dataword:",dataword, "\n", dataw)

    #for summation

    sum = '0'

    for i in dataw: #['11100' (0), '01010'(i), '10001'(i)]
        sum =(int(sum, 2) + int(i, 2))
        sum = bin(sum)[2:]
    #print("Summation: ", sum)

    # In case the lenght of summation longer than word_size
    while len(sum) > word_size:
        sum = (int(sum[len(sum) - word_size:],2) + int(sum[0: len(sum) - word_size], 2))
        sum = bin(sum)[2:]

    # In case the length of summation less than word_size, add "0" in front of the result
    while len(sum) < word_size:
        sum = '0' + sum

    dataw.append(bi_inverse(sum))

    print("Sum: ",sum, "\n" "Checksum: ",(bi_inverse(sum)), "\nThe codeword is "+ " ".join(dataw))
```

How to run Checksum_gen



Test:

```
print('-------------1----------------------')
cgs_1 = Checksum_gen('10101001001110010000111101', 8, 3)

print('-------------2----------------------')
cgs_2 = Checksum_gen('1011001101011010', 4, 4)

print('-------------3----------------------')
cgs_3 = Checksum_gen('110110010100', 4, 3)

print('-------------4----------------------')
cgs_4 = Checksum_gen('11001001010010100110', 4, 5)

print('-------------5----------------------')
cgs_5 = Checksum_gen('010110001101', 4, 3)

print('-------------6----------------------')
cgs_6 = Checksum_gen('1011011010101010', 4, 4)

print('-------------7----------------------')
cgs_7 = Checksum_gen('0100101010100101111010111', 4, 5)

print('-------------8----------------------')
cgs_8 = Checksum_gen('101001111111', 4, 3)

print('-------------9----------------------')
cgs_9 = Checksum_gen('0110110101010010', 4, 4)

print('-------------10----------------------')
cgs_10 = Checksum_gen('11101111111010110111', 5, 3)

print('-------------11----------------------')
cgs_11 = Checksum_gen('10101001001110001', 8, 2)
```

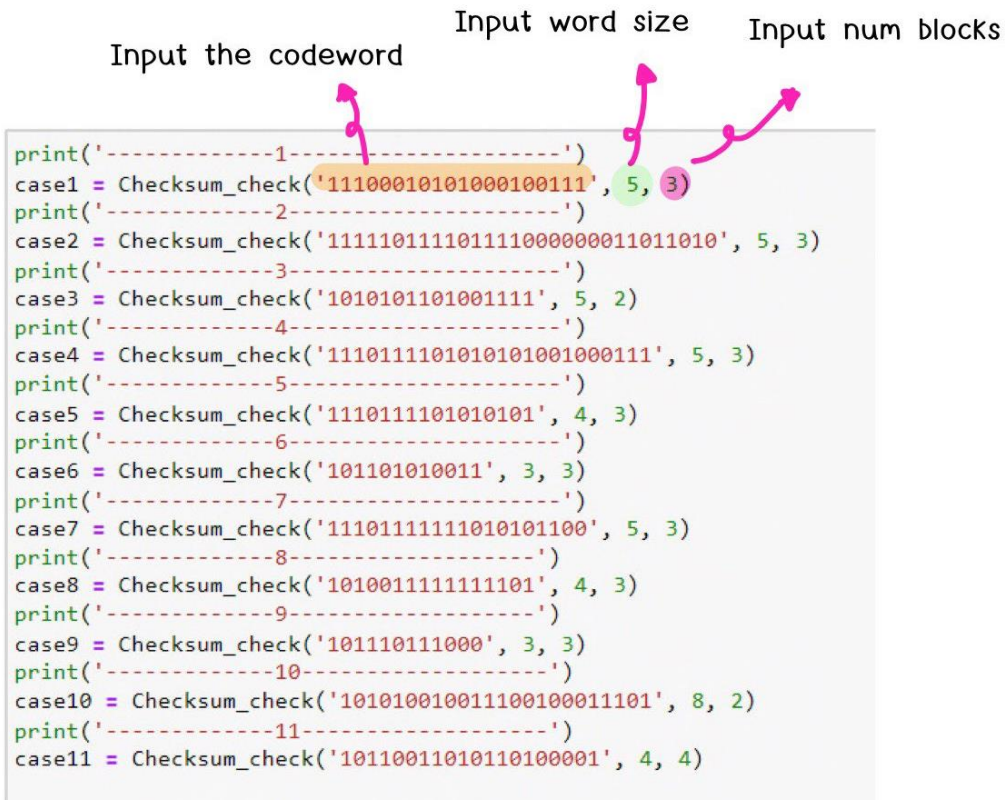
Output:

```
-------------1---------------------
Dataword: 10101001001111001000111101
 ['10101001', '00111001', '00011101']
Sum:  11111111
Checksum:  00000000
The codeword is 10101001 00111001 00011101 00000000
-------------2---------------------
Dataword: 1011001101011010
 ['1011', '0011', '0101', '1010']
Sum:  1110
Checksum:  0001
The codeword is 1011 0011 0101 1010 0001
-------------3---------------------
Dataword: 110110010100
 ['1101', '1001', '0100']
Sum:  1011
Checksum:  0100
The codeword is 1101 1001 0100 0100
-------------4---------------------
Dataword: 11001001010010100110
 ['1100', '1001', '0100', '1010', '0110']
Sum:  1011
Checksum:  0100
The codeword is 1100 1001 0100 1010 0110 0100
-------------5---------------------
Dataword: 010110001101
 ['0101', '1000', '1101']
Sum:  1011
Checksum:  0100
The codeword is 0101 1000 1101 0100
-------------6---------------------
Dataword: 1011011010101010
 ['1011', '0110', '1010', '1010']
Sum:  0111
Checksum:  1000
The codeword is 1011 0110 1010 1010 1000
-------------7---------------------
Dataword: 010010101010010111101011
 ['0100', '1010', '1010', '0101', '1110']
Sum:  1101
Checksum:  0010
The codeword is 0100 1010 1010 0101 1110 0010
-------------8---------------------
Dataword: 101001111111
 ['1010', '0111', '1111']
Sum:  0010
Checksum:  1101
The codeword is 1010 0111 1111 1101
-------------9---------------------
Dataword: 0110110101010010
 ['0110', '1101', '0101', '0010']
Sum:  1011
Checksum:  0100
The codeword is 0110 1101 0101 0010 0100
-------------10--------------------
Dataword: 11101111111010110111
 ['11101', '11111', '10101']
Sum:  10011
Checksum:  01100
The codeword is 11101 11111 10101 01100
-------------11--------------------
Dataword: 1010100100111001
 ['10101001', '00111001']
Sum:  11100010
Checksum:  00011101
The codeword is 10101001 00111001 00011101
```

Checker: validity = Checksum_check(codeword, word_size, num_blocks)

```
def Checksum_check(codeword, word_size, num_blocks):

    d = list(codeword)
    dataw = []

    for i in range(num_blocks+1):
    #    print(i)
        dataw.append("".join(d[i * word_size: i * word_size + word_size]))
    print(dataw)

    sum = '0'

    for i in dataw: #['11100' (0), '01010'(i), '10001'(i)]
        sum =(int(sum, 2) + int(i, 2))
        sum = bin(sum)[2:]
    #print("Summation: ", sum)

    # In case the lenght of summation longer than word_size
    while len(sum) > word_size:
        sum = (int(sum[len(sum) - word_size:],2) + int(sum[0: len(sum) - word_size], 2))
        sum = bin(sum)[2:]

    # In case the length of summation less than word_size, add "0" in front of the result
    while len(sum) < word_size:
        sum = '0' + sum

    # For check validation
    check = ''

    for i in range(word_size):
        check += '0'

    print("Codeword: "+" ".join(dataw))

    if bi_inverse(sum) == check:
        print("Validity of the array of codewords: PASS")
    else:
        print("Validity of the array of codewords: FAIL")
```

How to run Checksum_check




```
print('-------------1----------------')
case1 = Checksum_check('1110001010100010011', 5, 3)
print('-------------2----------------')
case2 = Checksum_check('11111011110111100000011011010', 5, 3)
print('-------------3----------------')
case3 = Checksum_check('1010101101001111', 5, 2)
print('-------------4----------------')
case4 = Checksum_check('11101111010101001000111', 5, 3)
print('-------------5----------------')
case5 = Checksum_check('1110111101010101', 4, 3)
print('-------------6----------------')
case6 = Checksum_check('101101010011', 3, 3)
print('-------------7----------------')
case7 = Checksum_check('11101111111010101100', 5, 3)
print('-------------8----------------')
case8 = Checksum_check('1010011111111101', 4, 3)
print('-------------9----------------')
case9 = Checksum_check('101110111000', 3, 3)
print('-------------10----------------')
case10 = Checksum_check('10101001001110010011101', 8, 2)
print('-------------11----------------')
case11 = Checksum_check('10110011010110100001', 4, 4)
```

Test:

```
print('-------------1---------------------')
case1 = Checksum_check('11100010101000100111', 5, 3)
print('-------------2---------------------')
case2 = Checksum_check('11111011110111100000011011010', 5, 3)
print('-------------3---------------------')
case3 = Checksum_check('1010101101001111', 5, 2)
print('-------------4---------------------')
case4 = Checksum_check('11101111010101010001000111', 5, 3)
print('-------------5---------------------')
case5 = Checksum_check('1110111101010101', 4, 3)
print('-------------6---------------------')
case6 = Checksum_check('101101010011', 3, 3)
print('-------------7---------------------')
case7 = Checksum_check('11101111111010101100', 5, 3)
print('-------------8-------------------')
case8 = Checksum_check('1010011111111101', 4, 3)
print('-------------9-------------------')
case9 = Checksum_check('101110111000', 3, 3)
print('-------------10-------------------')
case10 = Checksum_check('10101001001110010000111101', 8, 2)
print('-------------11-------------------')
case11 = Checksum_check('10110011010110100001', 4, 4)
```

Output:

```
-------------1---------------------
['11100', '01010', '10001', '00111']
Codeword: 11100 01010 10001 00111
Validity of the array of codewords: PASS

-------------2---------------------
['11111', '01111', '01111', '00000']
Codeword: 11111 01111 01111 00000
Validity of the array of codewords: FAIL

-------------3---------------------
['10101', '01101', '00111']
Codeword: 10101 01101 00111
Validity of the array of codewords: FAIL

-------------4---------------------
['11101', '11101', '01010', '10010']
Codeword: 11101 11101 01010 10010
Validity of the array of codewords: FAIL

-------------5---------------------
['1110', '1111', '0101', '0101']
Codeword: 1110 1111 0101 0101
Validity of the array of codewords: FAIL

-------------6---------------------
['101', '101', '010', '011']
Codeword: 101 101 010 011
Validity of the array of codewords: FAIL

-------------7---------------------
['11101', '11111', '10101', '01100']
Codeword: 11101 11111 10101 01100
Validity of the array of codewords: PASS

-------------8-------------------
['1010', '0111', '1111', '1101']
Codeword: 1010 0111 1111 1101
Validity of the array of codewords: PASS

-------------9-------------------
['101', '110', '111', '000']
Codeword: 101 110 111 000
Validity of the array of codewords: FAIL

-------------10-------------------
['10101001', '00111001', '00011101']
Codeword: 10101001 00111001 00011101
Validity of the array of codewords: PASS

-------------11-------------------
['1011', '0011', '0101', '1010', '0001']
Codeword: 1011 0011 0101 1010 0001
Validity of the array of codewords: PASS
```

Hamming code:

```
def Hamming_gen(dataword):
    #Find Number of Redundancy Bit
    length = len(dataword)
    for i in range(length):
        if(2**i >= length + i + 1):
            r = i
            break
    #Place Redundancy Bit
    j = 0
    k = 1
    res = ''
    for i in range(1, length + r+1):
        if(i == 2**j):
            res = res + '0'
            j += 1
        else:
            res = res + dataword[-1 * k]
            k += 1
    #Reversing
    output = res[::-1]
    #Finding Parity bit
    n = len(output)
    for i in range(r):
        x = 0
        for j in range(1, n + 1):
            if(j & (2**i) == (2**i)):
                x = x ^ int(output[-1 * j])
        output = output[:n-(2**i)] + str(x) + output[n-(2**i)+1:]
    return output

def Hamming_check(codeword):
    m = len(codeword)
    #Calculate Parity bit
    for i in range(m):
        if(2**i >= m + i + 1):
            nr = i
            break
    n = len(codeword)
    res = 0
    for i in range(nr):
        val = 0
        for j in range(1, n + 1):
            if(j & (2**i) == (2**i)):
                val = val ^ int(codeword[-1 * j])
        #Append Parity bit
        res = res + val*(10**i)
    if(int(str(res), 2) == 0): return -1
    else: return int(str(res), 2)
```

Test:

```
test1 = Hamming_gen("101011")
test2 = Hamming_gen("1010")
test3 = Hamming_gen("1100101")
test4 = Hamming_gen("1011010")
test5 = Hamming_gen("101001")
test6 = Hamming_gen("100110010")
test7 = Hamming_gen("11100101")
test8 = Hamming_gen("1010010010")
test9 = Hamming_gen("1010")
test10 = Hamming_gen("1010010010100010")
print("codeword 1: ", test1)
print("codeword 2: ", test2)
print("codeword 3: ", test3)
print("codeword 4: ", test4)
print("codeword 5: ", test5)
print("codeword 6: ", test6)
print("codeword 7: ", test7)
print("codeword 8: ", test8)
print("codeword 9: ", test9)
print("codeword 10: ", test10)
print("check 1: ", Hamming_check(test1))
print("check 2: ", Hamming_check(test2))
print("check 3: ", Hamming_check(test3))
print("check 4: ", Hamming_check(test4))
print("check 5: ", Hamming_check(test5))
print("check 6: ", Hamming_check(test6))
print("check 7: ", Hamming_check(test7))
print("check 8: ", Hamming_check(test8))
print("check 9: ", Hamming_check(test9))
print("check 10: ", Hamming_check(test10))
```

Output:

```
PS C:\Python> python -u "c:\Python\Computer Data\q4.py"
codeword 1:  1011010111
codeword 2:  1010010
codeword 3:  11000101100
codeword 4:  10101010000
codeword 5:  1011001110
codeword 6:  1001110010011
codeword 7:  111010100100
codeword 8:  10100110011010
codeword 9:  1010010
codeword 10:  1010001001010100010010
check 1:  -1
check 2:  -1
check 3:  -1
check 4:  -1
check 5:  -1
check 6:  -1
check 7:  -1
check 8:  -1
check 9:  -1
check 10:  -1
```

Generator: codeword = Hamming_gen(dataword)

```
def Hamming_gen(dataword):
    #Find Number of Redundancy Bit
    length = len(dataword)
    for i in range(length):
        if(2**i >= length + i + 1):
            r = i
            break
    #Place Redundancy Bit
    j = 0
    k = 1
    res = ''
    for i in range(1, length + r+1):
        if(i == 2**j):
            res = res + '0'
            j += 1
        else:
            res = res + dataword[-1 * k]
            k += 1
    #Reversing
    output = res[::-1]
    #Finding Parity bit
    n = len(output)
    for i in range(r):
        x = 0
        for j in range(1, n + 1):
            if(j & (2**i) == (2**i)):
                x = x ^ int(output[-1 * j])
        output = output[:n-(2**i)] + str(x) + output[n-(2**i)+1:]
    return output
```

How to run Hamming_gen

```
test1 = Hamming_gen("101011")
test2 = Hamming_gen("1010")
test3 = Hamming_gen("1100101")
test4 = Hamming_gen("1011010")
test5 = Hamming_gen("101001")
test6 = Hamming_gen("100110010")
test7 = Hamming_gen("11100101")
test8 = Hamming_gen("1010010010")
test9 = Hamming_gen("1010")
test10 = Hamming_gen("10100100100100010")
```

Checker: error_pos = Hamming_check(codeword)

```
def Hamming_check(codeword):
    m = len(codeword)
    #Calculate Parity bit
    for i in range(m):
        if(2**i >= m + i + 1):
            nr = i
            break
    n = len(codeword)
    res = 0
    for i in range(nr):
        val = 0
        for j in range(1, n + 1):
            if(j & (2**i) == (2**i)):
                val = val ^ int(codeword[-1 * j])
        #Append Parity bit
        res = res + val*(10**i)
    if(int(str(res), 2) == 0): return -1
    else: return int(str(res), 2)
```

How to run Hamming_check

```
print("check 1: ", Hamming_check(test1))
print("check 2: ", Hamming_check(test2))
print("check 3: ", Hamming_check(test3))
print("check 4: ", Hamming_check(test4))
print("check 5: ", Hamming_check(test5))
print("check 6: ", Hamming_check(test6))
print("check 7: ", Hamming_check(test7))
print("check 8: ", Hamming_check(test8))
print("check 9: ", Hamming_check(test9))
print("check 10: ", Hamming_check(test10))
```