Bit Stuffing

def bit\_stuff(data):

stuffed\_data = ""

count = 0

for bit in data:

if bit == '1':

count += 1

stuffed\_data += bit

else:

count = 0

stuffed\_data += bit

if count == 5:

stuffed\_data += '0'

count = 0

return stuffed\_data

def bit\_destuff(data):

destuffed\_data = ""

count = 0

for bit in data:

if bit == '1':

count += 1

destuffed\_data += bit

else:

count = 0

destuffed\_data += bit

if count == 5 and len(destuffed\_data) > 1: # Skip if the first bit is stuffed

if destuffed\_data[-1] == '0':

count = 0

else:

destuffed\_data = destuffed\_data[:-1] # Remove the stuffed bit

count -= 1

return destuffed\_data

# Example usage

data = input() # Original data

stuffed\_data = bit\_stuff(data)

print("Stuffed data:", stuffed\_data)

destuffed\_data = bit\_destuff(stuffed\_data)

print("Destuffed data:", destuffed\_data)

111110100111100110111101111110

Stuffed data: 11111001001111001101111011111010

Destuffed data: 111100100111100110111101111010

Bit destuffing

# Function for bit de-stuffing

def bit\_destuffing(data):

destuffed\_data = ''

i = 0

j = 0

# Loop to traverse the data string

while i < len(data):

# If the current bit is a set bit

if data[i] == '1':

destuffed\_data += data[i]

# Loop to check for the next 5 bits

count = 1

k = i + 1

while k < len(data) and data[k] == '1' and count < 5:

destuffed\_data += data[k]

count += 1

# If 5 consecutive set bits are found, skip the next bit

if count == 5:

k += 1

i = k

# Otherwise, insert the current bit into the destuffed\_data

else:

destuffed\_data += data[i]

i += 1

j += 1

# Print Destuffed data

print("Destuffed data:", destuffed\_data)

# Driver Code

if \_\_name\_\_ == '\_\_main\_\_':

data = "111110100111100110111101111110" # Example input string

bit\_destuffing(data)

Destuffed data: 11111111111001111110011111111111011111111110

Byte Stuffing:

list1=[]

flag='@'

esc='/'

a=input('Enter your message=')

list1.append(flag)

len1=len(a)

for i in a:

if i=='@':

list1.append(esc)

list1.append(i)

else:

list1.append(i)

list1.append(flag)

print ('At senders side=',list1)

len1=len(a)

list2=[]

del list1[0],list1[len(list1)-1]

for i in range(0,len1,1):

if a[i]=='#':

list1.remove('#')

else:

list2.append(a[i])

print ('At recievers side=',list2)

Enter your message=ravikarthik@gmail.com

At senders side= ['@', 'r', 'a', 'v', 'i', 'k', 'a', 'r', 't', 'h', 'i', 'k', '/', '@', 'g', 'm', 'a', 'i', 'l', '.', 'c', 'o', 'm', '@']

At recievers side= ['r', 'a', 'v', 'i', 'k', 'a', 'r', 't', 'h', 'i', 'k', '@', 'g', 'm', 'a', 'i', 'l', '.', 'c', 'o', 'm']

Parity Check Implementations

# Python3 code to get parity of a binary string.

# Function to get parity of binary string.

# It returns 1 if the binary string has odd parity,

# and returns 0 if it has even parity

def getParity(binary\_string):

# Convert binary string to integer

n = int(binary\_string, 2)

parity = 0

while n:

parity = ~parity

n = n & (n - 1)

return parity

# Driver program to test getParity()

binary\_string = input("Enter a binary string: ")

print("Parity of binary string", binary\_string, " = ",

("odd" if getParity(binary\_string) else "even"))

Enter a binary string: 0101101

Parity of binary string 0101101 = even

Main parity check code

def get\_vrc(data):

vrc = ''

for i in range(len(data[0])): # Iterate over each bit position

count = sum(int(dataword[i]) for dataword in data) # Count the number of set bits

vrc += str(count % 2) # Append the parity bit

return vrc

def get\_lrc(data):

lrc = ''

for i in range(7): # Iterate over each bit position

count = sum(int(dataword[i]) for dataword in data) # Count the number of set bits

lrc += str(count % 2) # Append the parity bit

return lrc

# Function to get parity of binary string.

# It returns 1 if the binary string has odd parity,

# and returns 0 if it has even parity

def get\_parity(binary\_string):

# Convert binary string to integer

n = int(binary\_string, 2)

parity = 0

while n:

parity = ~parity

n = n & (n - 1)

return parity

# Driver program to test getParity()

if \_\_name\_\_ == "\_\_main\_\_":

# Input binary strings

data = []

for i in range(4):

while True:

binary\_string = input(f"Enter binary string {i+1}: ")

if len(binary\_string) != 7 or not all(bit in ['0', '1'] for bit in binary\_string):

print("Invalid binary string. Enter a 7-bit binary string.")

continue

else:

data.append(binary\_string)

break

# Get and print VRC

vrc = get\_vrc(data)

print("Vertical Redundancy Check (VRC):", vrc)

# Get and print LRC

lrc = get\_lrc(data)

print("Longitudinal Redundancy Check (LRC):")

# Get and print parity

for i, binary\_string in enumerate(data):

parity = get\_parity(binary\_string)

print(f"Parity of binary string {i+1} = {'odd' if parity else 'even'}")

# example:

# Enter binary string 1: 0101101

# Enter binary string 2: 1011010

# Enter binary string 3: 1110010

# Enter binary string 4: 0001110

LRC(Logitudinal redundancy check)

def calculate\_lrc(data):

lrc = ''

for i in range(len(data[0])):

c1 = data[0][i]

c2 = data[1][i]

c3 = data[2][i]

e = int(c1) + int(c2) + int(c3)

if e % 2 == 0:

lrc += '1'

else:

lrc += '0'

return lrc

# Main function

if \_\_name\_\_ == "\_\_main\_\_":

# Input datawords

datawords = []

for i in range(3):

while True:

dataword = input(f"Enter {i+1}st Dataword: ")

if len(dataword) != 7 or not all(bit in ['0', '1'] for bit in dataword):

print("Invalid Data. Enter a 7-bit binary dataword.")

continue

else:

datawords.append(dataword)

break

# Calculate LRC

lrc = calculate\_lrc(datawords)

# Concatenate datawords and LRC

transmitted\_data = ''.join(datawords) + lrc

print("LRC:", transmitted\_data)

Enter 1st Dataword: 0101101

Enter 2st Dataword:

Invalid Data. Enter a 7-bit binary dataword.

Enter 2st Dataword: 1011010

Enter 3st Dataword: 1110010

LRC: 0101101101101011100101111010

Find only LRC

def calculate\_lrc(datawords):

lrc = ''

for i in range(len(datawords[0])):

count = sum(int(d[i]) for d in datawords)

if count % 2 == 0:

lrc += '1'

else:

lrc += '0'

return lrc

# Main function to get input and calculate LRC

def main():

datawords = []

for \_ in range(4): # Assuming 3 datawords as per the original C program

dataword = input("Enter a binary dataword (8 bits): ")

if len(dataword)!= 8:

print("Invalid Data")

continue

if not all(c in '01' for c in dataword):

print("Enter binary digits")

continue

datawords.append(dataword)

lrc = calculate\_lrc(datawords)

print(f"\nLRC: {lrc[::-1]}")

if \_\_name\_\_ == "\_\_main\_\_":

main()

# EXAMPLE

# Enter a binary dataword (8 bits): 11100111

# Enter a binary dataword (8 bits): 11011101

# Enter a binary dataword (8 bits): 00111001

# Enter a binary dataword (8 bits): 10101001

CRC

def xor(a, b):

result = []

for i in range(1, len(b)):

if a[i] == b[i]:

result.append('0')

else:

result.append('1')

return ''.join(result)

def mod2div(dividend, divisor):

pick = len(divisor)

tmp = dividend[0: pick]

while pick < len(dividend):

if tmp[0] == '1':

tmp = xor(divisor, tmp) + dividend[pick]

else:

tmp = xor('0'\*pick, tmp) + dividend[pick]

pick += 1

if tmp[0] == '1':

tmp = xor(divisor, tmp)

else:

tmp = xor('0'\*pick, tmp)

checkword = tmp

return checkword

def encodeData(data, key):

l\_key = len(key)

appended\_data = data + '0'\*(l\_key-1)

remainder = mod2div(appended\_data, key)

codeword = data + remainder

print("Remainder:", remainder)

print("Encoded Data (Data + Remainder):", codeword)

return codeword

def decodeData(codeword, key):

remainder = mod2div(codeword, key)

print("Remainder:", remainder)

if all(bit == '0' for bit in remainder):

print("No Error Detected")

else:

print("Error Detected")

if \_\_name\_\_ == "\_\_main\_\_":

data = input("Enter data: ")

key = input("Enter key: ")

# Sender side

print("at sender side")

codeword = encodeData(data, key)

print('at reciever side')

# Receiver side

decodeData(codeword, key)

Enter data: 11010110100

Enter key: 101

at sender side

Remainder: 11

Encoded Data (Data + Remainder): 1101011010011

at reciever side

Remainder: 00

No Error Detected

CHECKSUM

def ones\_complement(data):

return ''.join(['1' if bit == '0' else '0' for bit in data])

def checksum(data, block\_size):

n = len(data)

if n % block\_size != 0:

pad\_size = block\_size - (n % block\_size)

data = '0' \* pad\_size + data

result = data[:block\_size]

for i in range(block\_size, n, block\_size):

next\_block = data[i:i+block\_size]

additions = ""

carry = 0

for j in range(block\_size - 1, -1, -1):

sum\_ = (int(next\_block[j]) + int(result[j]) + carry)

carry = sum\_ // 2

additions = str(sum\_ % 2) + additions

final = ""

for digit in additions[::-1]:

sum\_ = int(digit) + carry

carry = sum\_ // 2

final = str(sum\_ % 2) + final

result = final

return ones\_complement(result)

# Driver code

if \_\_name\_\_ == "\_\_main\_\_":

bit\_str1 = input("Enter the first binary string: ")

bit\_str2 = input("Enter the second binary string: ")

checksum\_result = checksum(bit\_str1 + bit\_str2, len(bit\_str1))

print("Checksum:", checksum\_result)

# Function to find the one's complement of the given binary string

# Enter the first binary string: 1110011001100110

# Enter the second binary string: 1101010101010101

# Checksum: 0100010001000011

Hamming Code

def ham\_calc(position, code):

count = 0

i = position - 1

while i < len(code):

for j in range(i, min(i + position, len(code))):

if code[j] == '1':

count += 1

i = i + 2 \* position

return '1' if count % 2 != 0 else '0'

def solve(input\_bits):

p\_n = 0

i = 0

while len(input\_bits) > (2 \*\* i) - (i + 1):

p\_n += 1

i += 1

c\_l = p\_n + len(input\_bits)

code = ['0'] \* c\_l

j = k = 0

for i in range(c\_l):

if i == (2 \*\* k) - 1:

k += 1

else:

code[i] = input\_bits[j]

j += 1

for i in range(p\_n):

position = 2 \*\* i

value = ham\_calc(position, code)

code[position - 1] = value

print("The generated Code Word is:", ''.join(code))

# Given input message Bit as string

input\_bits\_str = "0111"

# Convert string to list of bits

input\_bits = list(input\_bits\_str)

# Function Call

solve(input\_bits)

The generated Code Word is: 0001111

Djikstras Algorithm:

import heapq

def dijkstra(graph, source):

# Initialize distances dictionary to store shortest distances from source

distances = {vertex: float('inf') for vertex in graph}

distances[source] = 0

# Priority queue to store vertices with their corresponding distances

priority\_queue = [(0, source)]

while priority\_queue:

# Pop vertex with minimum distance from the priority queue

current\_distance, current\_vertex = heapq.heappop(priority\_queue)

# If current distance is greater than the stored distance, skip

if current\_distance > distances[current\_vertex]:

continue

# Traverse neighbors of the current vertex

for neighbor, weight in graph[current\_vertex].items():

distance = current\_distance + weight

# If new distance is shorter, update distances dictionary and priority queue

if distance < distances[neighbor]:

distances[neighbor] = distance

heapq.heappush(priority\_queue, (distance, neighbor))

return distances

# Take input for graph data

graph = {}

num\_edges = int(input("Enter the number of edges: "))

for \_ in range(num\_edges):

start, end, weight = input("Enter start, end, and weight separated by space: ").split()

weight = int(weight)

if start not in graph:

graph[start] = {}

if end not in graph:

graph[end] = {}

graph[start][end] = weight

graph[end][start] = weight

# Take input for source vertex

source\_vertex = input("Enter the source vertex: ")

# Call Dijkstra's algorithm

shortest\_distances = dijkstra(graph, source\_vertex)

# Print shortest distances from source vertex

print("Shortest distances from vertex", source\_vertex)

for vertex, distance in shortest\_distances.items():

print("Vertex:", vertex, "Distance:", distance)

Enter the number of edges: 4

Enter start, end, and weight separated by space: A B 5

Enter start, end, and weight separated by space: B C 4

Enter start, end, and weight separated by space: C D 1

Enter start, end, and weight separated by space: D A 4

Enter the source vertex: A

Shortest distances from vertex A

Vertex: A Distance: 0

Vertex: B Distance: 5

Vertex: C Distance: 5

Vertex: D Distance: 4

Bellman Ford Algorithm:

'''Bellman-Ford algorithm is used to find the shortest paths

from a single source vertex to all other vertices in a weighted graph, even

if the graph contains negative weight edges

(as long as there are no negative weight cycles).

Below is a Python implementation of the Bellman-Ford algorithm:'''

class Graph:

def \_\_init\_\_(self, vertices):

self.V = vertices

self.graph = []

def add\_edge(self, u, v, w):

self.graph.append([u, v, w])

def bellman\_ford(self, src):

# Initialize distances from source to all vertices as infinity

dist = [float('inf')] \* self.V

dist[src] = 0

# Relax all edges V-1 times

for \_ in range(self.V - 1):

for u, v, w in self.graph:

if dist[u] != float('inf') and dist[u] + w < dist[v]:

dist[v] = dist[u] + w

# Check for negative weight cycles

for u, v, w in self.graph:

if dist[u] != float('inf') and dist[u] + w < dist[v]:

print("Graph contains negative weight cycle")

return

# Print shortest distances

print("Vertex \tDistance from Source")

for i in range(self.V):

print(f"{i}\t{dist[i]}")

# Example usage

g = Graph(5)

g.add\_edge(0, 1, -1)

g.add\_edge(0, 2, 4)

g.add\_edge(1, 2, 3)

g.add\_edge(1, 3, 2)

g.add\_edge(1, 4, 2)

g.add\_edge(3, 2, 5)

g.add\_edge(3, 1, 1)

g.add\_edge(4, 3, -3)

# Source vertex

src = 0

g.bellman\_ford(src)

Vertex Distance from Source

0 0

1 -1

2 2

3 -2

4 1

FLOYD WARDHALLS ALGO

INF = float('inf')

def floyd\_warshall(graph):

V = len(graph)

dist = [[0]\*V for \_ in range(V)]

# Initialize distance matrix

for i in range(V):

for j in range(V):

dist[i][j] = graph[i][j]

# Iterate through all vertices

for k in range(V):

# Pick all vertices as source one by one

for i in range(V):

# Pick all vertices as destination for the above source

for j in range(V):

# If vertex k is on the shortest path from i to j, then update the value of dist[i][j]

dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j])

# Print the shortest distances

print("Shortest distances between all pairs of vertices:")

for i in range(V):

for j in range(V):

if dist[i][j] == INF:

print("INF", end=" ")

else:

print(dist[i][j], end=" ")

print()

# Example usage

graph = [

[0, 5, INF, 10],

[INF, 0, 3, INF],

[INF, INF, 0, 1],

[INF, INF, INF, 0]

]

floyd\_warshall(graph)

Shortest distances between all pairs of vertices:

0 5 8 9

INF 0 3 4

INF INF 0 1

INF INF INF 0