program for SHA-3 option with a block size of 1024 bits and assume that each of the lanes in the first message block (P0) has at least one nonzero bit. To start, all of the lanes in the internal state matrix that correspond to the capacity portion of the initial state are all zeros. Show how long it will take before all of these lanes have at least one nonzero bit. Note: Ignore the permutation. That is, keep track of the original zero lanes even after they have changed position in the matrix. # Keccak lane-visit simulation (simple)

# Assumptions:

# - 25 lanes indexed as idx = x + 5\*y where x,y in 0..4

# - Lane size = 64 bits, rate r = 1024 => 16 rate lanes (indices 0..15)

# - Capacity lanes are indices 16..24 (9 lanes)

# - Initial state: rate lanes filled (nonzero) after absorbing P0; capacity lanes start zero

# - We ignore bit-level transformations; we only use the pi (lane permutation) mapping

# - After each absorption, new message (with all rate lanes nonzero) is XORed into the rate lanes.

# That effectively ensures those rate lanes are nonzero immediately after absorption.

# - After absorption we apply the pi permutation to move lanes, then check which original capacity

# lane indices have been nonzero at least once (we "mark" them permanently).

# - Repeat until all capacity lanes have been marked nonzero.

def idx\_to\_xy(idx):

return (idx % 5, idx // 5)

def xy\_to\_idx(x, y):

return x + 5 \* y

def pi\_map\_index(idx):

# Pi mapping: B[x, y] = A[(x + 3\*y) % 5, x]

# So lane at (u,v) in A moves to (x,y) in B where

# x = v

# y = (2\*u + 3\*v)? <-- careful: common formula used in Keccak literature:

# We will use the standard: B[x,y] = A[(x+3\*y) mod 5, x].

# So lane at A[(x+3\*y) mod 5, x] goes to B[x,y].

# For inverse mapping (where does A[u,v] go?), solve for (x,y) such that (u,v) = ((x+3\*y)%5, x)

# That gives x = v and u = (v + 3\*y) mod 5 => 3\*y = (u - v) mod 5 => y = 2\*(u - v) mod 5 (since 3 inverse mod5 = 2).

# So mapping (u,v) -> (x = v, y = (2\*(u - v)) mod 5)

u, v = idx\_to\_xy(idx)

x = v

y = (2 \* ((u - v) % 5)) % 5

return xy\_to\_idx(x, y)

def simulate\_blocks(rate\_indices, capacity\_indices, max\_blocks=1000):

# Initially after absorbing P0, rate lanes are nonzero

# We track for each lane index whether it has been nonzero at any time

seen = set() # indices of original capacity lanes that have been filled at least once

state\_nonzero = set(rate\_indices) # set of indices currently nonzero in the state

# Mark any capacity indices that might already intersect (unlikely since rate and capacity disjoint)

for idx in capacity\_indices:

if idx in state\_nonzero:

seen.add(idx)

# If already all filled

if set(capacity\_indices).issubset(seen):

return 0, seen

blocks = 1 # we already absorbed P0 (block 1)

while blocks <= max\_blocks:

# Apply pi permutation to current nonzero lanes (they move)

new\_state = set()

for idx in state\_nonzero:

new\_state.add(pi\_map\_index(idx))

# After permutation, check if any of the original capacity indices now contain nonzero

for cap\_idx in capacity\_indices:

if cap\_idx in new\_state:

seen.add(cap\_idx)

if set(capacity\_indices).issubset(seen):

return blocks, seen

# Now absorb next message block: XOR message (with all rate lanes nonzero) into the rate lanes.

# XOR with nonzero just makes sure those rate indices become nonzero (we treat nonzero as boolean).

new\_state.update(rate\_indices)

# Update for next round

state\_nonzero = new\_state

blocks += 1

return None, seen # didn't finish within max\_blocks

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

# rate: lanes 0..15, capacity: lanes 16..24

rate\_indices = list(range(16))

capacity\_indices = list(range(16, 25))

blocks\_needed, seen = simulate\_blocks(rate\_indices, capacity\_indices, max\_blocks=1000)

if blocks\_needed is None:

print("Did not fill all capacity lanes within 1000 blocks. Seen so far:", sorted(seen))

else:

print(f"All capacity lanes (indices {capacity\_indices}) became nonzero after {blocks\_needed} block(s).")

print("Order of capacity lanes observed (indices):", sorted(seen))

