**Additional Experiment Notes**

1. **Preparation**

**a) Environment (If Needed)**

To avoid Cobbler error when creating a conda env on Mac

conda create -n py37 or py312

conda activate py37

## use x86\_64 architecture channel(s)

conda config --env --set subdir osx-64

## install python, numpy, etc. (add more packages here...)

conda install python=3.7 numpy

# Add these packages

pip install jupyter lab

pip install notebook

pip install langchain-core

pip install langchain-community

**b) Install HumanEval**

1. Clone repo
2. Replace evaluation.py and execution.py with modified files – to run the same code without multiprocessing.
3. Then run “pip install -e human-eval.”

**Appendix**

**1. Examples of hallucinations when using non—chat optimized model ”meta/meta-llama-3-8b”**

# HumanEval/11 (hallucinations, incomplete code in the end)

return ''.join([str(int(c1 ) ^ int(c2)) for c1, c2 in zip (a, b)])

def xor \_strings(a: str, b: str) -> str:

""" Input are two strings a and b consisting only of 1s and 0s.

Perform binary XOR on these inputs and return result also as a string.

>>> xor\_strings ('010', '110')

'100'

"""

return ''.join([str(int(c1 ) ^ int(c2)) for c1, c2 in zip (a, b)])

def xor \_strings\_list(strings: List[str]) -> str:

""" Input are two strings a and b consisting only of 1s and 0s .

Perform binary XOR on these inputs and return result also as a string.

>>> xor\_strings\_list (['010', '110'])

'100'

"""

return ''.join([str(int(c1 ) ^ int(c2)) for c1, c2 in zip (\*strings)])

def xor\_strings \_list2(strings: List[str]) -> str:

""" Input are two strings a and b consisting only of 1s and 0 s.

Perform binary XOR on these inputs and return result also as a string.

>>> xor\_strings \_list2(['010', '110 '])

'100'

"""

return ''.join([str(int (c1) ^ int(c2 )) for c1, c2 in zip(\*strings)])

def xor\_strings\_list3(strings: List [str]) -> str:

""" Input are two strings a and b consisting only of 1s and 0s.

Perform binary XOR on these inputs and return result also as a string.

>>> xor\_strings\_list3(['010', '110'])

'100'

"""

return ''.join([ str(int(c1) ^ int (c2)) for c1, c2 in zip(\*strings)] )

def xor\_strings\_list4(strings : List[str]) -> str:

""" Input are two strings a and b consisting only of 1s and 0s.

Perform binary XOR on these inputs and return result also as a string.

>>> xor\_strings\_list4(['010 ', '110'])

'100 '

"""

return ''.join ([str(int(c1) ^ int(c2)) for c1 , c2 in zip(\*strings )])

def xor\_strings\_list5 (strings: List[str]) -> str :

""" Input are two strings a and b consisting only of 1s and 0s.

Perform binary XOR on these inputs and return result also as a string.

>>> xor\_strings\_list5(['010 ', '110'])

'100 '

"""

return ''.join ([str(int(c1) ^ int(c2)) for c1 , c2 in zip(\*strings )])

def xor\_strings\_list6 (strings: List[str]) -> str :

""" Input are two strings a and b consisting only of 1s and 0s.

Perform binary XOR on these inputs and return result also as a string .

>>> xor\_strings\_list6 (['010', '110'])

'100'

"""

return ''.join([str(int(c1 ) ^ int(c2)) for c 1, c2 in zip(\* strings)])

def xor\_strings\_list 7(strings: List[str]) -> str:

""" Input are two strings a and b consisting only of 1s and 0s .

Perform binary XOR on these inputs and return result also as a string.

>>> xor\_strings\_list 7(['010', '110'])

'100'

"""

return ''.join([str(int(c 1) ^ int(c2)) for c1, c2 in zip(\*strings)])

def xor \_strings\_list8(strings: List[str ]) -> str:

""" Input are two strings a and b consisting only of 1s and 0s.

Perform binary XOR on these inputs and return result also as a string.

>>> xor \_strings\_list8(['010', ' 110'])

'100'

"""

return ''.join([str (int(c1) ^ int(c 2)) for c1, c 2 in zip(\*strings)])

def xor\_strings\_list9(strings: List[str]) -> str:

""" Input are two strings a and b consisting only of 1s and 0 s.

Perform binary XOR on these inputs and return result also as a string.

>>> xor\_strings \_list9(['010', '110'])

'100'

"""

return ''.join([str(int(c1 ) ^ int(c2)) for c1, c2 in zip (\*strings)])

def xor\_strings \_list10(strings: List[str]) -> str:

""" Input are two strings a and b consisting only of 1s and 0 s.

Perform binary XOR on these inputs and return result also

====================================================================================================

# HumanEval/12 (hallucination - unnecessary doctest in main())

if not strings:

return None

max\_len = 0

best\_str = strings[0 ]

for s in strings[ 1:]:

if len(s) > max\_len:

max\_len = len(s)

best\_str = s

elif len(s ) == max\_len:

best \_str = s

return best \_str

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

import doctest

doctest.testmod()

====================================================================================================

# HumanEval/13 (hallucinations, incomplete code in the end)

if b == 0:

return a

else:

return greatest\_common\_divisor(b, a % b)

def least\_common \_multiple(a: int, b: int) -> int:

""" Return a least common multiple of two integers a and b

>>> least\_common\_multiple(3, 5)

15

>>> least\_common\_multiple(25, 15)

75

"""

return (a \* b) // greatest\_common\_divisor (a, b)

def is\_prime (n: int) -> bool:

""" Return True if the given integer n is prime

>>> is\_prime(3)

True

>>> is\_prime(5 )

True

>>> is \_prime(6)

False

"""

if n == 1:

return False

for i in range(2, n):

if n % i == 0:

return False

return True

def prime \_factor\_decomposition(n: int) -> List[int]:

""" Return a list of prime factors of the given integer n

>>> prime \_factor\_decomposition(1)

[]

>>> prime\_factor\_decomposition (2)

[2]

>>> prime\_factor\_decomposition( 6)

[2, 3]

>>> prime\_factor\_de composition(15)

[3 , 5]

"""

factors = []

i = 2

while i \* i <= n:

if n % i:

i += 1

else:

n //= i

factors.append(i)

if n > 1:

factors.append(n)

return factors

def prime\_decomposition(n: int) -> List[int]:

""" Return a list of prime factors of the given integer n

>>> prime\_decomposition(1)

[]

>>> prime\_decomposition (2)

[2]

>>> prime\_decomposition(6 )

[2, 3 ]

>>> prime\_decomposition( 15)

[3, 5]

"""

factors = []

i = 2

while i \* i <= n :

if n % i:

i += 1

else:

n //= i

factors.append(i)

if n > 1:

factors .append(n)

return factors

def prime\_factorization(n: int ) -> Dict[int, int]:

""" Return a dictionary of prime factors of the given integer n

>>> prime\_factorization(1 )

{}

>>> prime\_factor ization(2)

{2 : 1}

>>> prime \_factorization(6)

{ 2: 1, 3 : 1}

>>> prime \_factorization(15)

{ 3: 1, 5 : 1}

"""

factors = {}

i = 2

while i \* i <= n:

if n % i:

i += 1

else:

n //= i

factors[i] = factors.get(i, 0) + 1

if n > 1:

factors[n ] = factors.get(n, 0) + 1

return factors

def prime\_factorization (n: int) -> Dict[int , int]:

""" Return a dictionary of prime factors of the given integer n

>>> prime\_factor ization(1)

{}

>>> prime\_factorization(2)

{2: 1}

>>> prime\_factorization(6 )

{2: 1 , 3: 1}

>>> prime\_factorization(15 )

{3: 1 , 5: 1}

"""

factors = {}

i = 2

while i \* i <= n:

if n % i:

i += 1

else:

n //= i

factors [i] = factors.get(i, 0) + 1

if n > 1:

factors[n] = factors.get (n, 0) + 1

return factors

def prime \_factorization(n: int) -> Dict[int, int]:

""" Return a dictionary of prime factors of the given integer n

>>> prime\_factor ization(1)

{}

>>> prime\_factorization(2)

{2: 1}

>>> prime\_factorization(6 )

{2: 1, 3: 1}

>>> prime\_factorization(15)

{ 3: 1, 5 : 1}

"""

factors = {}

i = 2

while i \* i <= n :

if n % i:

i += 1

else :

n //= i

factors[i] = factors.get(i , 0) + 1

if n > 1 :

factors[n] = factors .get(n