


Use Python dictionaries to create a shifted directory of letters that will allow you to map your letters to shifted ones, so that you can use the encrypt and decrypt code provided in test\_caesar.py to encode and decode Caesar cipher messages. Demonstrate that your code works for your own custom message by running the encrypt and decrypt methods within the script provided.

 test\_caesar.py

Let's complete the Caesar cipher script by filling in the missing code to create the shifted dictionaries and demonstrate it with a custom message. Here's the completed version:

python

 Collapse

 Wrap

 Copy

```
# -*- coding: utf-8 -*-
```

```
"""
```

```
Caesar cypher script
```


```
Encode and decode messages by scrambling the letters in your message
```

```
Created on Fri Feb 1 23:06:50 2019
```



How can Grok help?



 DeepSearch

 Think

Grok 3 



```

#create the Caesar cypher
offset = 5 #choose your shift
totalLetters = 26
keys = {} #use dictionary for letter mapping
invkeys = {} #use dictionary for inverse letter mapping
for index, letter in enumerate(letters):
    # cypher setup
    if index < totalLetters: #lowercase
        new_index = (index + offset) % totalLetters
        keys[letter] = letters[new_index]
        invkeys[letters[new_index]] = letter
    else: #uppercase
        new_index = (index + offset) % totalLetters + totalLetters
        keys[letter] = letters[new_index]
        invkeys[letters[new_index]] = letter
print("Cypher Dict:", keys)

#encrypt
encryptedMessage = []
for letter in message:
    if letter == ' ': #spaces
        encryptedMessage.append(letter)
    else:
        encryptedMessage.append(keys[letter])
print("Encrypted Message:", ''.join(encryptedMessage)) #join is used to put list into st

#decrypt
decryptedMessage = []
for letter in encryptedMessage:
    if letter == ' ': #spaces
        decryptedMessage.append(letter)
    else:
        decryptedMessage.append(invkeys[letter])

```

How can Grok help?



DeepSearch




Think

Grok 3 ▾



Section I: b)

This type of cipher is easily broken by observing that some letters in the English language occur more often than others, such as the letter 'e'. Use the initial code provided in test\_caesar\_break.py to write an algorithm to break the code and decrypt the message provided.

 test\_caesar\_break.py

Let's complete the Caesar cipher breaking script by filling in the missing code to determine the shift and decrypt the message using frequency analysis. Here's the completed version:

python


 Collapse  Wrap  Copy


```
# -*- coding: utf-8 -*-  
"""  
Determine the shift of the Caesar Cypher  
  
Created on Sat Feb  2 23:03:02 2019  
  
@author: shakes  
"""  
from collections import Counter
```



How can Grok help?



 DeepSearch

 Think

Grok 3 



```
@author: shakes
"""

from collections import Counter
import string

message = "Zyp cpxpxmpc ez wzzv fa le esp delcd lyo yze ozhy le jzfc qppe Ehz ypgpc rtg"

#frequency of each letter
letter_counts = Counter(message)
#print(letter_counts)

#find max letter
maxFreq = -1
maxLetter = None
for letter, freq in letter_counts.items():
    print(letter, ":", freq)
    if letter.isalpha() and freq > maxFreq: # Only consider letters, update if freq hi
        maxFreq = freq
        maxLetter = letter
print("Max Occurring Letter:", maxLetter)


#predict shift
#assume max letter is 'e'
letters = string.ascii_letters #contains 'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'
e_position = letters.index('e') # Position of 'e' in alphabet (4 for lowercase)
max_letter_position = letters.index(maxLetter.lower()) # Position of most frequent letter
shift = (max_letter_position - e_position) % 26 # Calculate shift
print("Predicted Shift:", shift)


#decrypt the message
decryptedMessage = []
for letter in message:
    if letter.isalpha():
```



How can Grok help?



 DeepSearch

 Think

Grok 3 ⌵



```

#predict shift
#assume max letter is 'e'
letters = string.ascii_letters #contains 'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ
e_position = letters.index('e') # Position of 'e' in alphabet (4 for lowercase)
max_letter_position = letters.index(maxLetter.lower()) # Position of most frequent lett
shift = (max_letter_position - e_position) % 26 # Calculate shift
print("Predicted Shift:", shift)

#decrypt the message
decryptedMessage = []
for letter in message:
    if letter.isalpha():
        # Determine case and base position
        base = 0 if letter.islower() else 26
        # Convert to 0-25 range, shift back, and wrap around
        pos = letters.index(letter) - base
        new_pos = (pos - shift) % 26
        # Convert back to letter
        decryptedMessage.append(letters[base + new_pos])
    else:
        decryptedMessage.append(letter) # Keep non-letters unchanged
print("Decrypted Message:", ''.join(decryptedMessage))

```

## Explanation of Changes and How It Works:

### 1. Finding the Max Letter:

- Added `if letter.isalpha()` and `freq > maxFreq:` to:

How can Grok help?



DeepSearch



Think

Grok 3 ▾



Extend your script to break any Caesar cipher message of an arbitrary shift if you haven't already done so. Test out your code with your neighbour or mate by exchanging ciphered messages of unknown shifts between yourselves.

The script from the previous response already works for breaking any Caesar cipher message with an arbitrary shift, as it uses frequency analysis to determine the shift dynamically based on the most frequent letter (assuming it corresponds to 'e'). However, I'll enhance it slightly to make it more robust and user-friendly for testing with others, including handling input and providing a clean way to exchange and break messages. Here's the extended version:

python

✕ Collapse   ⇅ Wrap   📄 Copy

```
# -*- coding: utf-8 -*-  
"""  
Break any Caesar Cipher message using frequency analysis  
  
Extended on March 11, 2025  
"""  
from collections import Counter  
import string  
  
def break_caesar_cipher(encrypted_message):  
    # Frequency of each letter
```



How can Grok help?



🔍 DeepSearch



💡 Think

Grok 3 ▾



```

# Find max letter
maxFreq = -1
maxLetter = None
print("\nLetter frequencies:")
for letter, freq in letter_counts.items():
    if letter.isalpha():
        print(f"{letter} : {freq}")
        if freq > maxFreq:
            maxFreq = freq
            maxLetter = letter
print("Max Occurring Letter:", maxLetter)

# Predict shift (assume max letter is 'e')
letters = string.ascii_letters
e_position = letters.index('e') # Position of 'e' (4)
max_letter_position = letters.index(maxLetter.lower()) # Position of most frequent
shift = (max_letter_position - e_position) % 26
print("Predicted Shift:", shift)

# Decrypt the message
decryptedMessage = []
for letter in encrypted_message:
    if letter.isalpha():
        base = 0 if letter.islower() else 26
        pos = letters.index(letter) - base
        new_pos = (pos - shift) % 26
        decryptedMessage.append(letters[base + new_pos])
    else:
        decryptedMessage.append(letter)
return ''.join(decryptedMessage)

# Test with example messages
def test_with_friend():

```

How can Grok help?



DeepSearch


Think

Grok 3



Shakes the Horrible has decided to purchase a set of Enigma machines to ensure communication with his armed forces (consisting mainly of Spider Monkeys) is secure as possible.

a) Demonstrate a working Enigma machine with the given simple example script.

 test\_enigma\_simple.py

Let's tackle this step-by-step based on the provided Python script and the task at hand. The script demonstrates an Enigma machine implementation using the `pyEnigma` library, and we need to complete the missing parts to decrypt messages as requested.



Here's the complete script with both parts filled in:

python

✕ Collapse    ≡ Wrap    📋 Copy

```
# -*- coding: utf-8 -*-
import enigma
import rotor

# Set up the Enigma machine
engine = enigma.Enigma(rotor.ROTOR_Reflector_A, rotor.ROTOR_I,
                      rotor.ROTOR_II, rotor.ROTOR_III, key="ABC",
                      plugs="AA BB CC DD EE")

# Part a) Encrypt and Decrypt "Hello World"
message = "Hello World"
print("Message:", message)
secret = engine.encipher(message)
print("Encoded Message:", secret)
engine.set_key("ABC") # Reset to initial position
decoded = engine.encipher(secret)
print("Decoded Message:", decoded)

# Part b) Decrypt ShakesHorribleMessage
ShakesHorribleMessage = "Vxye ajgh D yf? Ptn uluo yjgco L ws nznde czidn. Bs j ccj qdbk q"
print("Encrypted Message:", ShakesHorribleMessage)
engine.set_key("ABC") # Reset to initial position
decrypted_message = engine.encipher(ShakesHorribleMessage)
print("Decrypted Message:", decrypted_message)
```

This demonstrates a working Enigma machine for Shakes the Horrible's secure communication needs. To get the exact decrypted message for Part b), you'd need to run this with the `pyEnigma`

How can Grok help?



DeepSearch



Think

Grok 3



In his arrogance however, he has decided to force his military to always use the window positions SSC as those are the initials of his name. He is also a cheapskate! He hasn't even bothered to purchase the plug board or any additional rotors, meaning he only has access to rotors I, II and III.

b) Decipher Shakes' message to his military given in the example script using the Enigma machine simulator.

Let's decipher Shakes the Horrible's message with the new constraints he's imposed due to his arrogance and thriftiness. We'll adjust the Enigma machine configuration to reflect his decisions and then decrypt the message from the original script:

text

✕ Collapse   ⇅ Wrap   📋 Copy

```
ShakesHorribleMessage = "Vxye ajgh D yf? Ptn uluo yjgco L ws nznde czidn. Bs j ccj qdbk q
```

## New Constraints

1. **Window Positions (Key):** Shakes forces his military to use the rotor window positions `"SSC"` (his initials) instead of the original `"ABC"`.
2. **No Plugboard:** As a cheapskate, he didn't buy the plugboard, so we'll omit the `plugins="AA BB CC DD EE"` setting.

How can Grok help?



🔍 DeepSearch

💡 Think

Grok 3



## Final Answer

Here's the complete script tailored to Shakes' new constraints:

python

✕ Collapse   ⇅ Wrap   📄 Copy

```
# -*- coding: utf-8 -*-
import enigma
import rotor

# Set up the Enigma machine with Shakes' constraints
engine = enigma.Enigma(rotor.ROTOR_Reflector_A, rotor.ROTOR_I,
                      rotor.ROTOR_II, rotor.ROTOR_III, key="SSC",
                      plugs="")

# Part b) Decrypt ShakesHorribleMessage
ShakesHorribleMessage = "Vxye ajgh D yf? Ptn uluo yjgco L ws nznde czidn. Bsj ccj qdbk q
print("Encrypted Message:", ShakesHorribleMessage)
decrypt = engine.encipher(ShakesHorribleMessage)
print("Decrypted Message:", decrypt)
```

To get the exact decrypted message, you'd need to run this script with the `pyEnigma` library installed (e.g., via `pip install pyenigma` if available). If you've run it and have the output, share it with me, and I can help interpret or adjust further! Otherwise, this setup correctly simulates Shakes' Enigma machine to decipher his military orders.

How can Grok help?



🔍 DeepSearch

💡 Think

Grok 3 ▾



After realising that his military forces are taking heavy losses, he decides to take the advice of his

generals and allows a different fixed, but hidden window positions for all further communications. He

however still refuses to buy the plugboard. He does decree though that all messages are to end with

the phrase "Hail Shakes!".

c) Using the known end phrase, phrases like which are called cribs, write an algorithm to break

Shakes' new code into the provided script and decrypt the message provided.

Let's tackle this new challenge: Shakes the Horrible has changed his Enigma machine's window positions (key) to a fixed but unknown setting, still refuses to use a plugboard, and now mandates that all messages end with "Hail Shakes!". We're tasked with using this known ending (a "crib") to break his code and decrypt the message from the original script:

text

✕ Collapse   ⇄ Wrap   📋 Copy

```
ShakesHorribleMessage = "Vxye ajgh D yf? Ptn uluo yjgco L ws nznde czidn. Bs j ccj qdbk q
```

## Problem Setup

- **Goal:** Determine the unknown key (window positions) and decrypt the message.

How can Grok help?



DeepSearch

Think

Grok 3



```

# Break the code via brute force search
# Function to test a key and check if it decrypts to end with "Hail Shakes!"
def test_key(key, ciphertext):
    engine = enigma.Enigma(rotor.ROTOR_Reflector_A, rotor.ROTOR_I,
                           rotor.ROTOR_II, rotor.ROTOR_III, key=key, plugs="")
    decrypted = engine.encrypt(ciphertext)
    # Check if the last 13 characters match "Hail Shakes!" (case-insensitive)
    decrypted_end = decrypted[-13:]
    return decrypted, decrypted_end.lower() == crib.lower()

# Generate all possible 3-letter keys (A-Z)
alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
keys = [''.join(k) for k in itertools.product(alphabet, repeat=3)]

print("Encrypted Message:", ShakesHorribleMessage)
print("Searching for the key...")

found_key = None
decrypted_message = None
for key in keys:
    decrypted, is_match = test_key(key, ShakesHorribleMessage)
    if is_match:
        found_key = key
        decrypted_message = decrypted
        break

# Print the Decoded message
if found_key:
    print(f"Key found: {found_key}")
    print("Decoded Message:", decrypted_message)
else:
    print("No key found that matches the crib 'Hail Shakes!'")

```

How can Grok help?



DeepSearch

Think

Grok 3 ▾



```

# Generate all possible 3-letter keys (A-Z)
alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
keys = [''.join(k) for k in itertools.product(alphabet, repeat=3)]

print("Encrypted Message:", ShakesHorribleMessage)
print("Searching for the key...")

found_key = None
decrypted_message = None
for key in keys:
    decrypted, is_match = test_key(key, ShakesHorribleMessage)
    if is_match:
        found_key = key
        decrypted_message = decrypted
        break

# Print the Decoded message
if found_key:
    print(f"Key found: {found_key}")
    print("Decoded Message:", decrypted_message)
else:
    print("No key found that matches the crib 'Hail Shakes!'")

```

#### Explanation

How can Grok help?



DeepSearch



Think

Grok 3 ▾



d) Add a counter to your script to keep track of the number of tries. How many attempts does it take to crack the code? How long did it take on your computer? How long do you think it would've taken for a computer in the 1940s?!

Let's address part (d) of your task: adding a counter to track the number of attempts, reporting how many attempts it takes to crack the code, measuring the time it took on your computer (AMD Ryzen 7), and estimating how long it might have taken on a 1940s-era computer. Your script already includes a counter and timer, so we'll build on that, analyze the results, and make an educated guess for the 1940s scenario.

### Updated Script with Counter and Timing

Your original script already has a `counter` variable and timing via `time.time()`, which is perfect for this task. I've refined it to ensure it aligns with Shakes' constraints (no plugboard, crib at the end) and provides clear output for part (d). Here's the script with these elements emphasized:

python

✕ Collapse   ⇌ Wrap   📋 Copy

```
# -*- coding: utf-8 -*-  
"""  
Create and test an Enigma machine encryption and decoding machine
```



How can Grok help?



🔍 DeepSearch



Think

Grok 3 ▾





## Break the code via brute force search

```
def shakes_enigma(ShakesHorribleMessage, crib, crib_substring):
    alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
    counter = 0 # Counter for number of attempts

    # Use full crib if no substring provided
    if not crib_substring:
        target = crib
    else:
        target = crib_substring

    for rotor1 in alphabet:
        for rotor2 in alphabet:
            for rotor3 in alphabet:
                key = f"{rotor1}{rotor2}{rotor3}"
                counter += 1 # Increment counter for each attempt
                # No plugboard (Shakes refuses to buy it)
                engine = enigma.Enigma(
                    rotor.ROTOR_Reflector_A,
                    rotor.ROTOR_I,
                    rotor.ROTOR_II,
                    rotor.ROTOR_III,
                    key=key,
                    plugs="" # Assuming "" is valid for no plugboard
                )

                # Decrypt the message
                decrypted = engine.encipher(ShakesHorribleMessage)

                # Check if the decrypted message ends with the crib
                if decrypted[-13:] == target: # Last 13 chars must be "Hail Shakes!"
                    print(f"Found key: {key}")
```



How can Grok help?



DeepSearch



Think

Grok 3 ▾





)

*# Decrypt the message*

decrypted = engine.encipher(ShakesHorribleMessage)

*# Check if the decrypted message ends with the crib*

if decrypted[-13:] == target: *# Last 13 chars must be "Hail Shakes!"*

print(f"Found key: {key}")

print(f"Decoded Message: {decrypted}")

print(f"Number of attempts: {counter}")

if crib\_substring and crib in decrypted:

print(f"Full crib '{crib}' also matches!")

return key, decrypted, counter

print("No key found with this rotor setup.")

return None, None, counter

*# Part d, add timer and run the attack*

start\_time = time.time()

key, decrypted, attempts = shakes\_enigma(ShakesHorribleMessage, crib, crib\_substring)

end\_time = time.time()

duration = end\_time - start\_time

if key:

print(f"Brute force attack completed in {round(duration, 2)} seconds on AMD Ryzen 7"

print(f"Attempts to crack the code: {attempts}")

else:

print(f"Failed to find key in {round(duration, 2)} seconds on AMD Ryzen 7 after {att

*## Part e, 3 rotors, additional 2 more rotor*

*# rotor = 5 \* 4 \* 3 = 60*

*# There are 60 unique rotors for 3 rotors from a set of 5*

How can Grok help?



DeepSearch



Think

Grok 3



### Final Answer for Part (d)

- **Counter and Attempts:** The script successfully added a counter, and it took 11,772 attempts to crack the code.
- **Time on Your Computer:** The brute-force attack completed in 10.77 seconds on your AMD Ryzen 7.
- **1940s Estimate:** On a hypothetical 1940s computer performing 1 decryption per second, it would have taken approximately 3.27 hours to crack the code with 11,772 attempts. For the full keyspace (17,576 attempts), it would take about 4.88 hours, though real Enigma cracking with bombes was much faster due to specialized hardware and crib techniques.

If Shakes the Horrible wasn't so ignorant and worried about money, he would have purchased both the extra 2 rotors and the plugboard. How much longer would have the cracking his code taken on your computer? An estimate as a number of tries or minutes/hours is sufficient.

Let's estimate how much longer it would take to crack Shakes the Horrible's code if he had purchased the extra two rotors and the plugboard, using your AMD Ryzen 7 computer as the baseline. We'll calculate the increase in the number of possible configurations and extrapolate the time based on your previous result of 10.77 seconds for 11,772 attempts.

### Original Setup

- **Rotors:** 3 fixed rotors (I, II, III) with 26 starting positions each, giving  $26^3 = 17,576$  possible key settings.
- **Plugboard:** None, so no additional complexity.
- **Attempts:** 11,772 (actual number to find the key).
- **Time:** 10.77 seconds on AMD Ryzen 7.
- **Time per Attempt:**  $10.77 / 11,772 \approx 0.000915$  seconds/attempt.

### New Setup with Extra Rotors and Plugboard

If Shakes had purchased the extra two rotors (let's call them IV and V, making a total of five roto. ▼

How can Grok help?



DeepSearch



Think

Grok 3 ▼



## 1. Rotor Combinations

- **Number of Rotors:** 5 total.
- **Choose 3 Rotors:** The order of rotors matters in an Enigma machine (left to right affects wiring), so we calculate permutations of 3 rotors from 5:
  - $\text{Permutations} = 5 \times 4 \times 3 = 60$  ways to arrange 3 rotors.
- **Starting Positions:** Each rotor has 26 possible starting positions, and with 3 rotors, this is  $26^3 = 17,576$  per rotor order.
- **Total Rotor Configurations:**  $60 \times 17,576 = 1,054,560$  possible settings (rotor order + starting positions).

## 2. Plugboard Configurations

- **Plugboard Capacity:** A standard Enigma plugboard allowed up to 10 pairs of letters to be swapped (20 letters total, leaving 6 unused).
- **Calculation:** The number of ways to choose 10 pairs from 26 letters is complex due to permutations and the fact that each pair is undirected (A-B is the same as B-A), and the order of pairing doesn't matter beyond the first 10 choices. The formula is:
  - Total ways to choose 10 pairs:  $\frac{26!}{2^{10} \cdot 10! \cdot 6!}$  (where  $2^{10}$  accounts for ordering within pairs,  $10!$  for ordering the pairs, and  $6!$  for the unused letters).
  - This simplifies to approximately 150,738,274,937,250 (as you calculated in Part e), but let's verify:

## 2. Plugboard Configurations

- **Plugboard Capacity:** A standard Enigma plugboard allowed up to 10 pairs of letters to be swapped (20 letters total, leaving 6 unused).
- **Calculation:** The number of ways to choose 10 pairs from 26 letters is complex due to permutations and the fact that each pair is undirected (A-B is the same as B-A), and the order of pairing doesn't matter beyond the first 10 choices. The formula is:
  - Total ways to choose 10 pairs:  $\frac{26!}{2^{10} \cdot 10! \cdot 6!}$  (where  $2^{10}$  accounts for ordering within pairs,  $10!$  for ordering the pairs, and  $6!$  for the unused letters).
  - This simplifies to approximately 150,738,274,937,250 (as you calculated in Part e), but let's verify:
    - Number of ways to choose 20 letters for 10 pairs:  $\binom{26}{20}$ .
    - Pair them:  $\frac{\binom{26}{20}}{10! \cdot 2^{10}}$  (since each pair can be swapped, and order of pairs doesn't matter).
    - Exact calculation:  $\frac{26!}{(6! \cdot 10! \cdot 2^{10})} \approx 150,738,274,937,250$ .
- This is the standard Enigma plugboard complexity, confirmed by historical data.

## 3. Total Configurations

- **Total Possibilities:** Rotor settings  $\times$  Plugboard settings =  $1,054,560 \times 150,738,274,937,250$ .
- **Approximation:**  $1,054,560 \times 150,738,274,937,250 \approx 1.59 \times 10^{17}$  (158,962,555,217,826,360,000, as you noted in Part e).
- This is the theoretical maximum, but in practice, Shakes might not use all plugboard pairs (e.g., 0 to 10 pairs). Let's assume a realistic scenario with, say, 5 pairs (a common setup):
  - 5 pairs from 26 letters:  $\binom{26}{10} \cdot \frac{1}{2^{5 \cdot 5!}}$  (adjusting for pairs)  $\approx 6.5 \times 10^7$ .

How can Grok help?



DeepSearch

Think

Grok 3



- 5 pairs from 26 letters:  $\binom{26}{10} \cdot \frac{1}{2^{5.5}!}$  (adjusting for pairs)  $\approx 6.5 \times 10^7$ .
- Total with 5 pairs:  $1,054,560 \times 6.5 \times 10^7 \approx 6.85 \times 10^{10}$ .

However, for a full estimate matching your Part e) total, we'll use the maximum plugboard complexity (150,738,274,937,250) to reflect Shakes going all out if he weren't cheap.

### Time Estimation

- **Original Attempts:** 11,772 out of 17,576 (67% of the keyspace).
- **New Attempts:** If we scale proportionally, the new total configurations are  $1,054,560 \times 150,738,274,937,250 / 17,576 \approx 9.04 \times 10^{15}$  times more possibilities than the original 17,576.
- **Adjusted Attempts:**  $11,772 \times 9.04 \times 10^{15} \approx 1.06 \times 10^{17}$  attempts to find the key at a similar position in the new keyspace.
- **Time on Ryzen 7:** At 0.000915 seconds/attempt:
  - $1.06 \times 10^{17} \times 0.000915 \approx 9.7 \times 10^{13}$  seconds.
  - Convert to years:  $9.7 \times 10^{13} / (365 \times 24 \times 60 \times 60) \approx 3,075$  years.

A number of sophisticated encryption schemes can be broken with some simple prior knowledge of the context in which the message was composed, but without knowing anything about the encryption scheme or device. Crack the following message intercepted by the United States navy that was addressed to a Japanese naval officer in 1941 to reveal its contents. It is actually doable by hand, but you may use Python as well. Once solved by hand, determine if an AI tool solve it for you.

Hint: Use the context given above to essentially construct a crib that helps you solve it. TIP ONE: note that is not 1 to 1 mapping to alphabet, TWO: use logic rather than just blindly mapping

Let's tackle this problem with a fresh perspective, focusing on the given context, hints, and tips to crack the cipher logically by hand. The message was intercepted by the U.S. Navy in 1941, addressed to a Japanese naval officer, and consists of the numbers: 19, 17, 17, 19, 14, 20, 23, 18, 19, 8, 12, 16, 19, 8, 3, 21, 8, 25, 18, 14, 18, 6, 3, 18, 8, 15, 18, 22, 18, 11 (30 numbers total). The hints are: the plaintext starts with "ATTACK," the mapping is not a 1-to-1 correspondence with the alphabet, and we should use logic rather than blind mapping.

### Step 1: Leverage the Context and Crib

- **Context:** In 1941, a message to a Japanese naval officer likely pertains to military operations. The most significant event involving the Japanese Navy in 1941 was the attack on Pearl Harbor.

How can Grok help?



DeepSearch



Think

Grok 3 ▾



## Step 2: Analyze the Numbers and Patterns

- **Ciphertext:** 19, 17, 17, 19, 14, 20, 23, 18, 19, 8, 12, 16, 19, 8, 3, 21, 8, 25, 18, 14, 18, 6, 3, 18, 8, 15, 18, 22, 18, 11.
- **Length:** 30 numbers.
- **Frequency of Numbers:**
  - 3: 2 times (positions 15, 22)
  - 6: 1 time (position 21)
  - 8: 5 times (positions 10, 14, 17, 24, 27)
  - 11: 1 time (position 30)
  - 12: 1 time (position 11)
  - 14: 2 times (positions 5, 20)
  - 15: 1 time (position 25)
  - 16: 1 time (position 12)
  - 17: 3 times (positions 2, 3, 28)
  - 18: 7 times (positions 8, 18, 19, 23, 26, 28, 29)
  - 19: 5 times (positions 1, 4, 9, 13, 19)
  - 20: 1 time (position 6)
  - 21: 1 time (position 16)
  - 22: 1 time (position 27)
  - 23: 1 time (position 7)
  - 25: 1 time (position 18)
- **Range:** 3 to 25 (missing 1, 2, 4, 5, 7, 9, 10, 13, 24, 26).





- **Range:** 3 to 25 (missing 1, 2, 4, 5, 7, 9, 10, 13, 24, 26).
- **Patterns:**
  - Repeated pairs: 17, 17 (positions 2-3); 19, 19 (positions 1, 4; 9, 13); 18, 18 (positions 18-19, 28-29).
  - High frequencies: 18 (7 times), 8 (5 times), 19 (5 times).

### Step 3: Hypothesize the Full Plaintext

- **Plaintext:** "ATTACK PEARL HARBOR" (17 letters, including spaces as separators for clarity: A T T A C K \_ P E A R L \_ H A R B O R).
- **Padding:** 13 remaining numbers, likely "XXXXXXXXXXXXX" (X repeated 13 times), a common padding technique.
- **Plaintext Frequency** (excluding spaces for now, 15 letters without spaces):
  - A: 3 (positions 1, 4, 10, 14)
  - T: 2 (positions 2, 3)
  - C: 1 (position 5)
  - K: 1 (position 6)
  - P: 1 (position 8)
  - E: 1 (position 9)
  - R: 3 (positions 11, 15, 17)
  - L: 1 (position 12)
  - H: 1 (position 14)
  - B: 1 (position 16)



- O: 1 (position 17)
- X: 13 (positions 18-30)
- **First 6 Numbers (Crib):** 19, 17, 17, 19, 14, 20 → A, T, T, A, C, K.
- **Observations:**
  - 19 appears at positions 1, 4 (both A), but also at 9, 13, 19 (E, space, X).
  - 17 appears at positions 2, 3 (both T), but also at 28 (X).
  - This confirms the mapping is not 1-to-1: 19 maps to A, E, and X; 17 maps to T and X.

#### Step 4: Interpret "Not 1-to-1 Mapping"

- **Many-to-One:** Multiple numbers map to the same letter (e.g., 19 maps to A, E, X; 17 maps to T, X).
- **Structural Encoding:**
  - **Polybius Square:** Numbers might represent row/column pairs (e.g., 19 = row 1, column 9), but the range 3-25 and single numbers suggest otherwise.
  - **Vigenère Cipher:** The same letter maps to different numbers based on a key (e.g., A + key1 = 19, A + key2 = 14).
  - **Additive Cipher with Codebook:** Numbers might be codebook values with additives, common in Japanese ciphers like JN-25, but we lack the codebook.
- Given the context (1941 Japanese naval message), a Vigenère cipher with a key (e.g., "TOKYO" or "JAPAN") is plausible, as it creates a many-to-one mapping when the key shifts the same letter to different numbers.

## Step 5: Test a Vigenère Cipher with the Crib

- **Key Hypothesis:** Use a relevant keyword like "TOKYO" (5 letters, reflecting the Japanese origin).
- **Key Positions (A=0, Z=25):** T=19, O=14, K=10, Y=24, O=14.
- **Ciphertext:** 19, 17, 17, 19, 14, 20, ...
- **Decoding:**  $P = (C - K) \bmod 26$ :
  - Position 1:  $19 - 19 = 0 = A$
  - Position 2:  $17 - 14 = 3 = D$
  - Position 3:  $17 - 10 = 7 = H$
  - Position 4:  $19 - 24 = -5 + 26 = 21 = V$
  - Position 5:  $14 - 14 = 0 = A$
  - Position 6:  $20 - 19 = 1 = B$
- **Result:** ADHVA... (not "ATTACK").
- **Try Key "JAPAN"** (J=9, A=0, P=15, A=0, N=13):
  - $19 - 9 = 10 = K$
  - $17 - 0 = 17 = R$
  - $17 - 15 = 2 = C$
  - $19 - 0 = 19 = T$
  - $14 - 13 = 1 = B$
  - $20 - 9 = 11 = L$
- **Result:** KRCTBL... (not "ATTACK").



Q: How can I test a Vigenère cipher with a crib?

How can Grok help?



DeepSearch



Think

Grok 3



## Step 6: Reassess the Mapping with Logic

- The Vigenère attempt didn't yield "ATTACK," suggesting the key or method is incorrect. Let's use the frequency and crib to deduce the mapping.
- **Crib Mapping:**
  - 19, 17, 17, 19, 14, 20 → A, T, T, A, C, K.
  - 19 → A (positions 1, 4)
  - 17 → T (positions 2, 3)
  - 14 → C (position 5)
  - 20 → K (position 6)
- **Extend to Full Plaintext** (first 17 numbers for "ATTACK PEARL HARBOR"):
  - Positions: A T T A C K \_ P E A R L \_ H A R B O R
  - Numbers: 19, 17, 17, 19, 14, 20, 23, 18, 19, 8, 12, 16, 19, 8, 3, 21, 8
  - Known: 19=A (1, 4), 17=T (2, 3), 14=C (5), 20=K (6), 19=E (9), 8=R (11, 15, 17), 12=L (12), 3=H (14).
- **Conflicts:**
  - 19 maps to A (1, 4) and E (9).
  - 8 maps to R (11, 15, 17) but also appears at 10 (A), 14 (A).
- **Last 13 (Padding):** 18, 14, 18, 6, 3, 18, 8, 15, 18, 22, 18, 11 → likely "XXXXXXXXXXXXXXXX".
  - 18 → X (positions 18, 19, 23, 26, 28, 29)
  - 14 → X (position 20), but 14=C earlier.
  - 8 → X (position 24), but 8=R earlier.

How can Grok help?



DeepSearch



Think

Grok 3 ▾



## Step 9: Final Mapping and Decoding

- **Mapping** (not 1-to-1):
  - $3 \rightarrow H, X$
  - $6 \rightarrow X$
  - $8 \rightarrow A, R, X$
  - $11 \rightarrow X$
  - $12 \rightarrow L$
  - $14 \rightarrow C, X$
  - $15 \rightarrow X$
  - $17 \rightarrow T$
  - $18 \rightarrow P, X$
  - $19 \rightarrow A, E, \text{space}$
  - $20 \rightarrow K$
  - $21 \rightarrow B$
  - $22 \rightarrow X$
  - $23 \rightarrow \text{space}$
- **Decode:**
  - $19, 17, 17, 19, 14, 20 \rightarrow A T T A C K$
  - $23 \rightarrow \_$
  - $18 \rightarrow P$
  - $19 \rightarrow E$