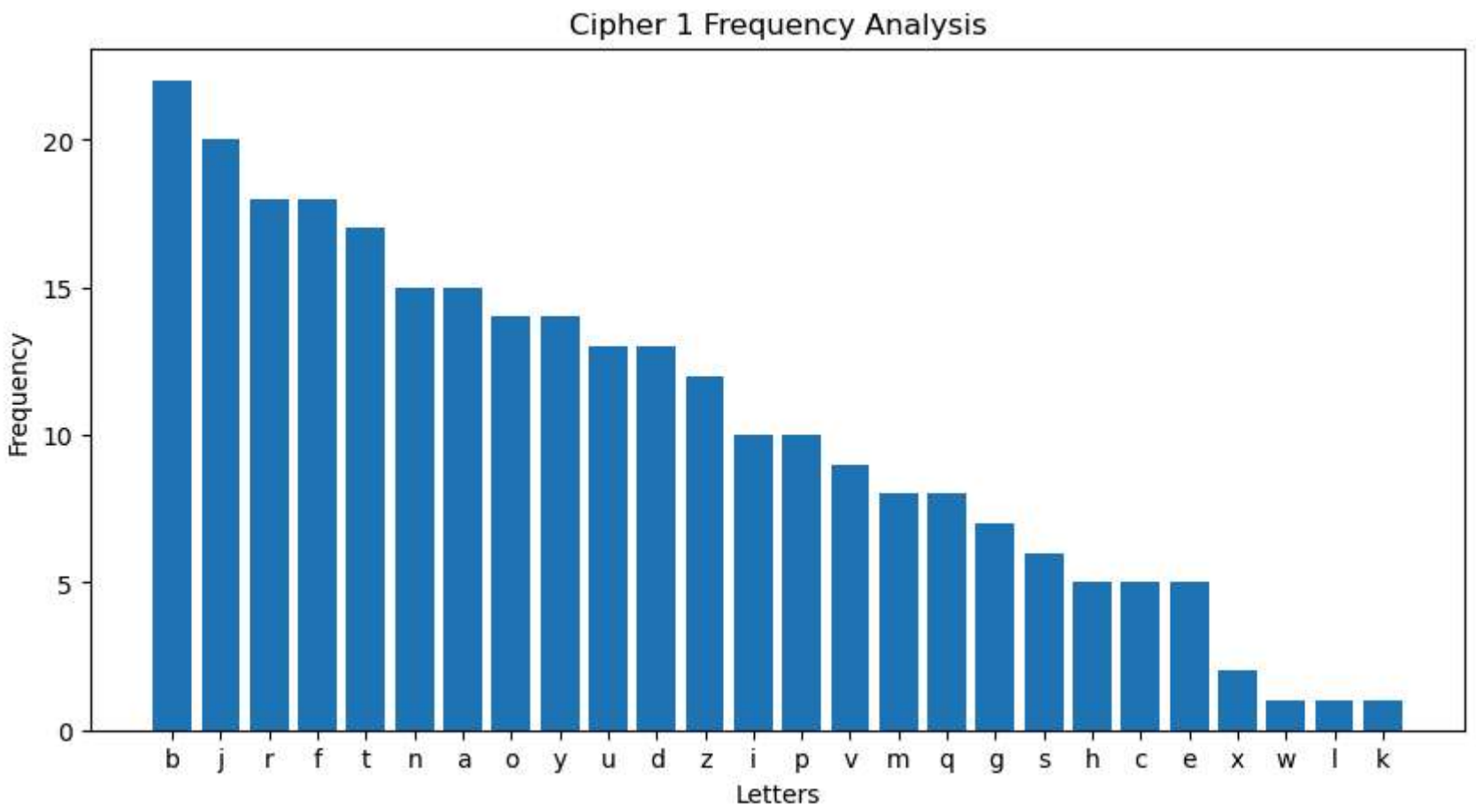
Cipher 1

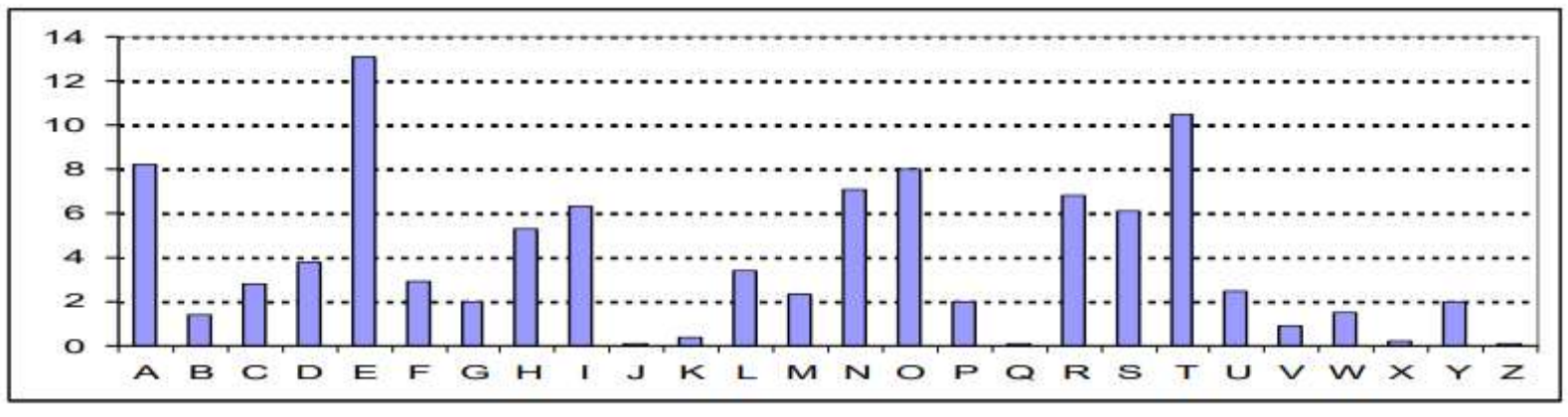
In [1]:
```
## import modules
from collections import Counter
import matplotlib.pyplot as plt
from math import gcd
from functools import reduce

ciphertext = """Oirmf qjff nfrh ub wf n bsbrjab bjvsriffn bzjotnu gzbpcfen ja oir ndujpyn uuztr ybln dbidrmory xvoi opjyyjab d

jzksbqjab uuzje xpzhvaddnojbit ffjygt, vn crdot ptry ja hbat tpcpbgt, zjtg ztczdvvmyt qez-tpcpbgt"""
```

Initial Frequency Analysis

In [2]:
```
## frequency analysis

# strip text, count frequencies, sort by frequency
ciphertext_cleaned = ''.join(filter(str.isalpha, ciphertext)).lower()
frequency = Counter(ciphertext_cleaned)
sorted_frequency = dict(sorted(frequency.items(), key=lambda item: item[1], reverse=True))

# Plot the frequencies
plt.figure(figsize=(10, 5))
plt.bar(sorted_frequency.keys(), sorted_frequency.values())
plt.xlabel('Letters')
plt.ylabel('Frequency')
plt.title('Cipher 1 Frequency Analysis')
plt.show()
```



- Distribution of characters frequencies match that of the english language
- Order of letters does not match that of the English language
- Suggests a substitution cipher has been used and not transposition
- Frequencies of letters in typical English. Source: ELEC6242 Lecture slides - classical ciphers



- b =? e, j =? t

In [3]:
```
## single character word frequency analysis

# find single char words
single_char_words = [word for word in ciphertext.split() if len(word) == 1]

# count frequencies
single_char_word_frequency = Counter(single_char_words)
sorted_single_char_word_frequency = dict(sorted(single_char_word_frequency.items(), key=lambda item: item[1], reverse=True))

print(sorted_single_char_word_frequency)
```

```
{'n': 1}
```

- Only 1 single letter word
- monoalphabetic substitution is plausible
- n =? {a, i}

In [4]:
```
# homogeneous diagraph frequency analysis (repeating chars)

# find homogenous digraphs (repeating characters)
homogenous_digraphs = []
for i in range(len(ciphertext_cleaned)-1):
    if ciphertext_cleaned[i] == ciphertext_cleaned[i+1]:
        homogenous_digraphs.append(ciphertext_cleaned[i]*2)

# count frequencies
homogenous_digraph_freq = Counter(homogenous_digraphs)
sorted_homogenous_digraph_freq = dict(sorted(homogenous_digraph_freq.items(), key=lambda item: item[1], reverse=True))

for digraph, freq in sorted_homogenous_digraph_freq.items():
    print(f"{digraph}: {freq},", end=' ')
```

```
ff: 4, uu: 3, bb: 1, yy: 1, rr: 1, aa: 1, dd: 1, tt: 1, vv: 1,
```

- 8 homogeneous digraphs is high but monoalphabetic is still plausible
- Most frequent homogeneous digraphs in standard English (source = https://blogs.sas.com/content/iml/2014/10/03/double-letter-bigrams.html):

L	S	E	O	T	F	P	R	M	C	N	D	G
0.577%	0.405%	0.378%	0.210%	0.171%	0.146%	0.137%	0.121%	0.096%	0.083%	0.073%	0.043%	0.025%

I	B	A	Z	X	U	H	Q	J	W	K	Y	V
0.023%	0.011%	0.003%	0.003%	0.003%	0.001%	0.001%	.	.	.	.	.	.

- ff =? ll, uu =? ss, bb =? ee

In [5]:
```
# homogeneous trigraph frequency analysis (repeating chars)

# find homogenous trigraphs (repeating characters)
homogenous_trigraphs = []
for i in range(len(ciphertext_cleaned)-2):
    if ciphertext_cleaned[i] == ciphertext_cleaned[i+1] == ciphertext_cleaned[i+2]:
        homogenous_trigraphs.append(ciphertext_cleaned[i]*3)

# count frequencies
homogenous_trigraph_freq = Counter(homogenous_trigraphs)
sorted_homogenous_trigraph_freq = dict(sorted(homogenous_trigraph_freq.items(), key=lambda item: item[1], reverse=True))

if sorted_homogenous_trigraph_freq:
    for trigraph, freq in sorted_homogenous_trigraph_freq.items():
        print(f"{trigraph}: {freq},", end=' ')
else:
    print("There are no homogenous trigraphs.")
```

```
There are no homogenous trigraphs.
```

- No homogeneous trigraphs
- Monoalphabetic still plausible
- We can use index of coincedence to check if mono or poly alphabetic substitution

Index of Coincidence

In [6]:
```
# Index of coincidence

def calculate_ic(text):
    frequency = Counter(text)
```

```
    N = len(text)
    ic = sum(f * (f - 1) for f in frequency.values()) / (N * (N - 1))
    return ic

# Calculate and print IC for monoalphabetic cipher (key_size = 1)
ic_mono = calculate_ic(ciphertext_cleaned)
print(f"Monoalphabetic:\n\tIC = {ic_mono}")

# Calculate and print IC for other key sizes
print("Polyalphabetic:")
for key_size in range(2, 6):
    split_texts = [''.join(ciphertext_cleaned[i::key_size]) for i in range(key_size)]
    ic_values = [calculate_ic(part) for part in split_texts]
    average_ic = sum(ic_values) / key_size
    print(f" k = {key_size}: IC = {average_ic}")
```

```
Monoalphabetic:
        IC = 0.047883260278532985
Polyalphabetic:
 k = 2: IC = 0.04866311716272439
 k = 3: IC = 0.07165664131956268
 k = 4: IC = 0.04648881262138505
 k = 5: IC = 0.0476751061656722
```

- The IC of random text ~= 0.038 (Classic Cipher Slides)
- The IC of written English ~= 0.066 (Classic Cipher Slides)
- There is a clear outlier for n = 3 with an IC of 0.72. This value is close the the value of written English - given the short length of the ciperhtext.
- The other IC values for each value of n all hover around the same value which is close the the IC value of random text.
- Given this, it is likely that the encryption method was not a monoalphabetic subsitution and is likely a polyalphabetic substitution
- We can check this further using the kasiski test

Kasisky Test

In [7]:
```
# find repeating sequences of length >3

def kasiski_test(ciphertext, min_length=3):
    sequences = {}
    for seq_len in range(min_length, 6):
        for i in range(len(ciphertext) - seq_len):
            seq = ciphertext[i:i + seq_len]
            if seq in sequences:
                sequences[seq].append(i)
            else:
                sequences[seq] = [i]

    repeated_sequences = {seq: locs for seq, locs in sequences.items() if len(locs) > 1}
    return repeated_sequences

repeated_sequences = kasiski_test(ciphertext_cleaned)

# Sort sequences by the number of repetitions in descending order
sorted_repeated_sequences = sorted(repeated_sequences.items(), key=lambda item: len(item[1]), reverse=True)

# Print the first 10 sequences in descending order of repetitions
for seq, locs in sorted_repeated_sequences[:10]:
    print(f"Sequence: {seq}, Locations: {locs}")
```

```
Sequence: jab, Locations: [22, 88, 115, 136, 193]
Sequence: uuz, Locations: [61, 118, 196]
Sequence: yja, Locations: [87, 114, 231]
Sequence: oir, Locations: [0, 51]
Sequence: ffn, Locations: [7, 31]
Sequence: frh, Locations: [10, 106]
Sequence: fnb, Locations: [16, 32]
Sequence: brj, Locations: [20, 134]
Sequence: rja, Locations: [21, 135]
Sequence: try, Locations: [64, 229]
```

- "jab" is the most repeated sequence
- We can caculate the gcd of the distances between the repititions to find the key length (Lecture slides)
- This should be 3 based on the IC values

In [8]:
```
# Find the distances between the repeated sequences of "jav"
jav_locations = repeated_sequences['jab']
distances = [jav_locations[i] - jav_locations[i - 1] for i in range(1, len(jav_locations))]

# Calculate the GCD of the distances
kasiski_value = reduce(gcd, distances)
print(f"Distances: {distances}")
print(f"GCD: {kasiski_value}")
```

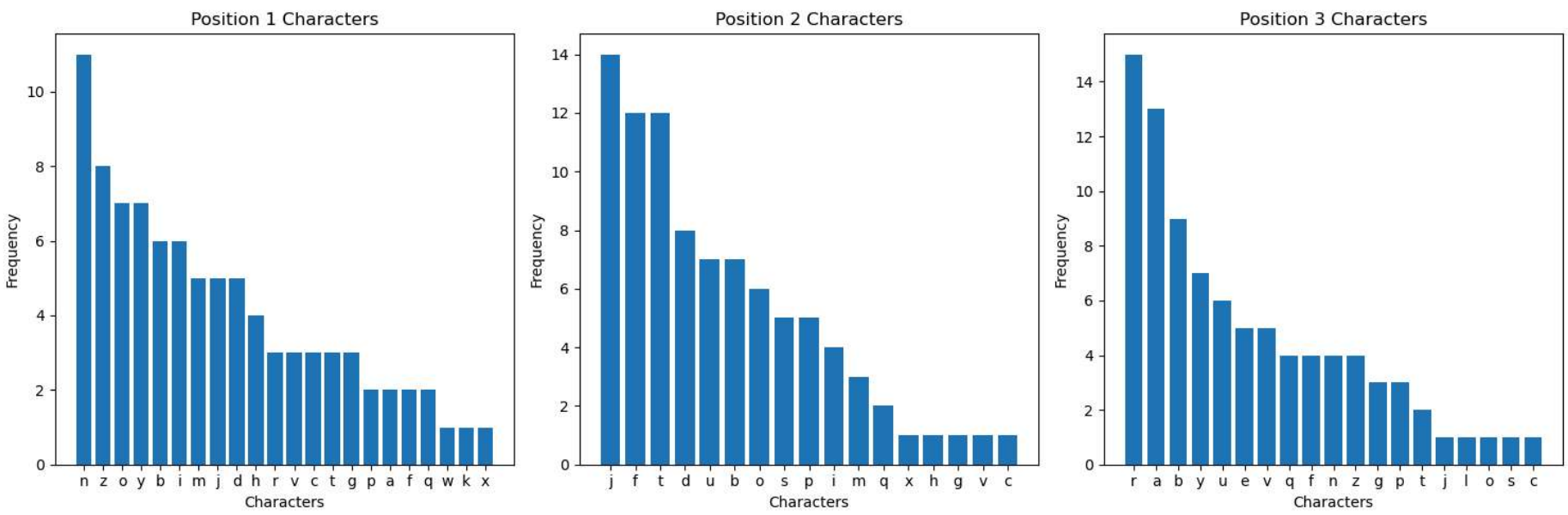
```
Distances: [66, 27, 21, 57]
GCD: 3
```

- GCD is three, concluding that the key length is 3

Tri-Alphabetic Subsitution

- 3 alphabets used
- Possibly tri-alphabetic substituion (every 3rd character) or a Vignere Cipher with Key size 3.
- These are 2 encryption algorithms from the lecture slides that could apply to this scenario

In [9]:
```python
## tri-alphabetic frequency analysis

# Plot frequencies for each part
plt.figure(figsize=(15, 5))

for i in range(3):
    c = ciphertext_cleaned[i::3]
    freq = Counter(c)
    freq = dict(sorted(freq.items(), key=lambda item: item[1], reverse=True))

    plt.subplot(1, 3, i+1)
    plt.bar(freq.keys(), freq.values())
    plt.title(f"Position {i+1} Characters")
    plt.xlabel('Characters')
    plt.ylabel('Frequency')

plt.tight_layout()
plt.show()
```



- Once again all 3 plots follow follow the same distribution as the English language with the letters substituted
- Most common letters for every 3n chars: n, 3n+1: j, 3n+2: r

In [10]:
```python
## substitute letters

# Initialize the substitution dictionary for the trio-alphabetic substitution
substitutions = {
    'n': 'e',  # Most common letter in c1
    'j': 'e',  # Most common letter in c2
    'r': 'e',  # Most common letter in c3
}

def substitute_text(text, substitutions, replace_non_substituted=False):
    substituted_text = []
    for char in text:
        if char in substitutions:
            substituted_text.append(substitutions[char])
        elif replace_non_substituted and char.isalpha():
            substituted_text.append('-')
        else:
            substituted_text.append(char)
    return ''.join(substituted_text)

print(substitutions, end='\n\n')

# Substitute letters in the ciphertext
substituted_text = substitute_text(ciphertext, substitutions)
#print(substituted_text, end="\n\n")
print(ciphertext, end="\n\n")

# Substitute letters in the ciphertext and replace non-substituted letters with '-'
substituted_text_with_dashes = substitute_text(ciphertext, substitutions, replace_non_substituted=True)
print(substituted_text_with_dashes, end="\n\n")
```

```
{'n': 'e', 'j': 'e', 'r': 'e'}

Oirmf qjff nfrh ub wf n bsbrjab bjvsriffn bzjotnu gzbpcfen ja oir ndujpyn uuztr ybln dbidrmory xvoi opjyyjab dudmqmfa nfya ffof
rh. Srrbeyjab uuz dudmqmfa ape nibrjab qnojridr voq  fjayornt, sjs ydtgzovih nouriuvqfyt, bay gbm

jzksbqjab uuzje xpzhvaddnojbit ffjygt, vn crdot ptry ja hbat tpcpbgt, zjtg ztczdvvmyt qez-tpcpbgt

--e-- -e-- e-e- -- -- e ---ee-- -e--e---e --e--e- -------e e- --e e--e--e ----e ---e ----e--e- ---- --e--e-- -------- e--- ----
e-. -ee---e-- --- -------- --- e--ee-- -e-ee--e ---  -e---ee-, -e- --------- e--e-------, --- ---

e-----e-- ---e- --------e-e--- --e---, -e -e--- --e- e- ---- -------, -e-- ---------- -----------
```

- By simply substituting e for the most frequent in all three languages, the result contains 'e-' twice.
- This could be Ed, Em, En, Eh. Definitely plausible but not commonly used words
- Also results in E being a single char word - impossible
- Continuing the manual substitution is very time expensive
- It would be more time efficient to try Vignere first, then attempt the manual substitution if it is not Vignere

Vignere

- Vignere uses 26 alpahbets all shifted by a different amount and is managed using a Vignere Square (lecture Slides)
- Below is the Vignere Square for the English language (Lecture Slides). The top row is the plaintext, left is the key and the intersection of both is the ciphertext.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
A	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
B	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A
C	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B
D	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C
E	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D
F	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E
G	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F
H	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G
I	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H
J	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I
K	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J
L	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K
M	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L
N	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M
O	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N
P	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
Q	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
R	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
S	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R
T	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
U	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T
V	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
W	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V
X	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W
Y	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X
Z	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y

- Key size = 3
- By taking the most frequent letters from the tri-alphabet frequency analysis: n,j,r. We can make a guess of the starting key using the table and substituting for E.
- Starting key = JFN

In [11]:
```python
## vignere

def vignere_decrypt(ciphertext, key):
    key = key.lower()
    key_length = len(key)
    key_as_int = [ord(i) - 97 for i in key]
    plaintext = ''
    positions = []

    j = 0  # index for key
    for i in range(len(ciphertext)):
        if ciphertext[i].isalpha():
            offset = 65 if ciphertext[i].isupper() else 97
            ciphertext_int = ord(ciphertext[i]) - offset
            value = (ciphertext_int - key_as_int[j % key_length]) % 26
            plaintext += chr(value + offset)
```

```
            positions.append(j % key_length + 1)  # store the position (1, 2, or 3)
            j += 1  # increment key index only if the character is alphabetic
        else:
            plaintext += ciphertext[i]
            positions.append(0)  # non-alphabetic characters get position 0

    return plaintext, positions

starting_key = 'jfn'
decrypted_text, positions = vignere_decrypt(ciphertext, starting_key)

print("Cipher Text:")
print(ciphertext, end='\n\n')

# Print the decrypted text
print("Decrypted Text:")
print(decrypted_text, end='\n\n')

# Print each of the 3 alphabets on their own with '-' for every other letter
print("Alphabet 1:")
for char, pos in zip(decrypted_text, positions):
    if pos == 1:
        print(char, end='')
    elif char.isalpha():
        print('-', end='')
    else:
        print(char, end='')
print('\n')

print("Alphabet 2:")
for char, pos in zip(decrypted_text, positions):
    if pos == 2:
        print(char, end='')
    elif char.isalpha():
        print('-', end='')
    else:
        print(char, end='')
print('\n')

print("Alphabet 3:")
for char, pos in zip(decrypted_text, positions):
    if pos == 3:
        print(char, end='')
    elif char.isalpha():
        print('-', end='')
    else:
        print(char, end='')
print()
```

```
Cipher Text:
Oirmf qjff nfrh ub wf n bsbrjab bjvsriffn bzjotnu gzbpcfen ja oir ndujpyn uuztr ybln dbidrmory xvoi opjyyjab dudmqmfa nfya ffof
rh. Srrbeyjab uuz dudmqmfa ape nibrjab qnojridr voq  fjayornt, sjs ydtgzovih nouriuvqfyt, bay gbm

jzksbqjab uuzje xpzhvaddnojbit ffjygt, vn crdot ptry ja hbat tpcpbgt, zjtg ztczdvvmyt qez-tpcpbgt

Decrypted Text:
Fdeda daas eaey po na a snoiens wwmnezase wmajgep tqwctare en fde eyhakle phqoe pwye yozyedjep sifd bgelpens yhuhddan ealr asfa
ey. Neiwrpens phq yhuhddan rkr edoiens lafeezye mjd  wenpjeeo, fan luotqjizc afpezpihalk, wnp bod

embnohens phqer okmyqnuyafeozo swelxo, ie xeujg goep en ywnk octkoxo, maot qopqyimhlk lrq-octkoxo

Alphabet 1:
F--d- -a-- e--y -- n- - s--i--s --m--z--e --a--e- -q--t--e -- f-- e--a--e --q-- p--e --z--d--p --f- -g--p--s --u--d-- e--r --f-
-y. --i--p--s --q --u--d-- r-- e--i--s --f--z-- m--  w--p--e-, -a- -u--q--z- -f--z--h--k, --p --d

--b--h--s --q-- o--y--u--f--z- -w--x-, -e --u-- g--p -- y--k --t--x-, -a-- q--q--m--k --q---t--x-

Alphabet 2:
-d--a --a- -a-- p- -a - -n--e-- w--n--a-- w--j--p --w--a-- e- -d- -y--k-- p--o- -w-- y--y--j-- s--d --e--e-- y--h--a- -a-- a--a
--. N--w--e-- p-- y--h--a- -k- -d--e-- l--e--y- -j-  -e--j--o, --n --o--j--c --p--p--a--, w-- b--

e--n--e-- p--e- -k--q--y--e--o --e--o, -- x--j- -o-- e- -w-- o--k--o, --o- -o--y--h-- l---o--k--o

Alphabet 3:
--e-- d--s --e- -o -- a --o--n- -w--e--s- -m--g-- t--c--r- -n --e --h--l- -h--e --y- -o--e--e- -i-- b--l--n- -h--d--n --l- -s--
e-. -e--r--n- -h- -h--d--n --r --o--n- -a--e--e --d  --n--e--, f-- l--t--i-- a--e--i--l-, -n- -o-

-m--o--n- -h--r --m--n--a--o-- s--l--, i- -e--g --e- -n --n- -c--o--, m--t --p--i--l- -r---c--o--
```

- The resulting text is not English
- Alphabet 3 assigns 'a' to a single characeter word, this is likely to be correct, therefore, N is likely to be the last character in the key
- In alphabet 3 we have the word "-o" which could be "to", this would change the second character in the key to B
- Also have the word "i-", this could be "in", changing the first character in the key to A

In [12]:
```
key = 'abn'
decrypted_text, positions = vignere_decrypt(ciphertext, key)
```

```
print("Cipher Text:")
print(ciphertext, end='\n\n')

# Print the decrypted text
print("Decrypted Text:")
print(decrypted_text, end='\n\n')

# Print each of the 3 alphabets on their own with '-' for every other letter
print("Alphabet 1:")
for char, pos in zip(decrypted_text, positions):
    if pos == 1:
        print(char, end='')
    elif char.isalpha():
        print('-', end='')
    else:
        print(char, end='')
print('\n')

print("Alphabet 2:")
for char, pos in zip(decrypted_text, positions):
    if pos == 2:
        print(char, end='')
    elif char.isalpha():
        print('-', end='')
    else:
        print(char, end='')
print('\n')

print("Alphabet 3:")
for char, pos in zip(decrypted_text, positions):
    if pos == 3:
        print(char, end='')
    elif char.isalpha():
        print('-', end='')
    else:
        print(char, end='')
print()
```

```
Cipher Text:
Oirmf qjff nfrh ub wf n bsbrjab bjvsriffn bzjotnu gzbpcfen ja oir ndujpyn uuztr ybln dbidrmory xvoi opjyyjab dudmqmfa nfya ffof
rh. Srrbeyjab uuz dudmqmfa ape nibrjab qnojridr voq  fjayornt, sjs ydtgzovih nouriuvqfyt, bay gbm

jzksbqjab uuzje xpzhvaddnojbit ffjygt, vn crdot ptry ja hbat tpcpbgt, zjtg ztczdvvmyt qez-tpcpbgt

Decrypted Text:
Oheme djes neeh to we a brorinb awvreiesn amjngnt tzaccern in ohe nchjoln thzse yayn coicemney wioh bpilyinb chdldmen nela esoe
eh. Reraryinb thz chdldmen aor nhorinb paoieice vnd  finynens, fjr ldstzniig aoteitiqelt, any fom

imkroqinb thzir xomhundcaoiois sfilgs, in bedng psey in hant sccoogs, mjst zspzcivllt prz-sccoogs

Alphabet 1:
O--m- -j-- n--h -- w- - b--r--b --v--i--n --j--n- -z--c--n -- o-- n--j--n --z-- y--n --i--m--y --o- -p--y--b --d--m-- n--a --o-
-h. --r--y--b --z --d--m-- a-- n--r--b --o--i-- v--  f--y--n-, -j- -d--z--i- -o--i--q--t, --y --m

--k--q--b --z-- x--h--d--o--i- -f--g-, -n --d-- p--y -- h--t --c--g-, -j-- z--z--v--t --z---c--g-

Alphabet 2:
-h--e --e- -e-- t- -e - -r--i-- a--r--e-- a--n--t --a--e-- i- -h- -c--o-- t--s- -a-- c--c--n-- w--h --i--i-- c--l--e- -e-- e--e
--. R--a--i-- t-- c--l--e- -o- -h--i-- p--i--c- -n-  -i--n--s, --r --s--n--g --t--t--e--, a-- f--

i--r--i-- t--i- -o--u--c--i--s --i--s, -- b--n- -s-- i- -a-- s--o--s, --s- -s--c--l-- p---s--o--s

Alphabet 3:
--e-- d--s --e- -o -- a --o--n- -w--e--s- -m--g-- t--c--r- -n --e --h--l- -h--e --y- -o--e--e- -i-- b--l--n- -h--d--n --l- -s--
e-. -e--r--n- -h- -h--d--n --r --o--n- -a--e--e --d  --n--e--, f-- l--t--i-- a--e--i--l-, -n- -o-

-m--o--n- -h--r --m--n--a--o-- s--l--, i- -e--g --e- -n --n- -c--o--, m--t --p--i--l- -r---c--o--
```

- This is close to English containing some English words and words that are close to English
- Decrypted text contains "vnd". This is likely to be "and".
- The v is encoded by the first letter of the key A. We can change this to V to form the word "and" in the text

In [13]:
```
key = 'vbn'
decrypted_text, positions = vignere_decrypt(ciphertext, key)

print("Cipher Text:")
print(ciphertext, end='\n\n')

# Print the decrypted text
print("Decrypted Text:")
print(decrypted_text, end='\n\n')

# Print each of the 3 alphabets on their own with '-' for every other letter
print("Alphabet 1:")
for char, pos in zip(decrypted_text, positions):
```

```
    if pos == 1:
        print(char, end='')
    elif char.isalpha():
        print('-', end='')
    else:
        print(char, end='')
print('\n')

print("Alphabet 2:")
for char, pos in zip(decrypted_text, positions):
    if pos == 2:
        print(char, end='')
    elif char.isalpha():
        print('-', end='')
    else:
        print(char, end='')
print('\n')

print("Alphabet 3:")
for char, pos in zip(decrypted_text, positions):
    if pos == 3:
        print(char, end='')
    elif char.isalpha():
        print('-', end='')
    else:
        print(char, end='')
print()
```

```
Cipher Text:
Oirmf qjff nfrh ub wf n bsbrjab bjvsriffn bzjotnu gzbpcfen ja oir ndujpyn uuztr ybln dbidrmory xvoi opjyyjab dudmqmfa nfya ffof
rh. Srrbeyjab uuz dudmqmfa ape nibrjab qnojridr voq  fjayornt, sjs ydtgzovih nouriuvqfyt, bay gbm

jzksbqjab uuzje xpzhvaddnojbit ffjygt, vn crdot ptry ja hbat tpcpbgt, zjtg ztczdvvmyt qez-tpcpbgt

Decrypted Text:
There does seem to be a growing awareness amongst teachers in the schools these days concerned with building children self este
em. Rewarding the children for showing patience and  kindness, for listening attentively, and for

improving their communications skills, is being used in many schools, most especially pre-schools

Alphabet 1:
T--r- -o-- s--m -- b- - g--w--g --a--n--s --o--s- -e--h--s -- t-- s--o--s --e-- d--s --n--r--d --t- -u--d--g --i--r-- s--f --t-
-m. --w--d--g --e --i--r-- f-- s--w--g --t--n-- a--  k--d--s-, -o- -i--e--n- -t--n--v--y, --d --r

--p--v--g --e-- c--m--i--t--n- -k--l-, -s --i-- u--d -- m--y --h--l-, -o-- e--e--a--y --e---h--l-

Alphabet 2:
-h--e --e- -e-- t- -e - -r--i-- a--r--e-- a--n--t --a--e-- i- -h- -c--o-- t--s- -a-- c--c--n-- w--h --i--i-- c--l--e- -e-- e--e
--. R--a--i-- t-- c--l--e- -o- -h--i-- p--i--c- -n-  -i--n--s, --r --s--n--g --t--t--e--, a-- f--

i--r--i-- t--i- -o--u--c--i--s --i--s, -- b--n- -s-- i- -a-- s--o--s, --s- -s--c--l-- p---s--o--s

Alphabet 3:
--e-- d--s --e- -o -- a --o--n- -w--e--s- -m--g-- t--c--r- -n --e --h--l- -h--e --y- -o--e--e- -i-- b--l--n- -h--d--n --l- -s--
e-. -e--r--n- -h- -h--d--n --r --o--n- -a--e--e --d  --n--e--, f-- l--t--i-- a--e--i--l-, -n- -o-

-m--o--n- -h--r --m--n--a--o-- s--l--, i- -e--g --e- -n --n- -c--o--, m--t --p--i--l- -r---c--o--
```

Success!

- The ciphertext has been successfully decrypted.
- The plaintext was encrypted using the Vigenere Cipher with the key VBN
- Plaintext: "There does seem to be a growing awareness amongst teachers in the schools these days concerned

with building children self esteem. Rewarding the children for showing patience and kindness, for listening attentively, and for improving their communications skills, is being used in many schools, most especially pre-schools"