Historic Crypto

Questions:

1. What are the advantage and disadvantage of these classical cipher?
    a. These ciphers are easy to implement and run very quickly.
2. Will you consider using these classical ciphers for your content protection? Why or why not?
    a. No, since they are easy to implement, they are easy to crack. Most of these are pretty much useless because of how easy it is for modern computers to brute force them.
3. Discuss your experience during the crack implementations.
    a. Cracking the Caesar was easy once we started using letter frequencies. We originally tried it by checking to see if there were English words in the output, but this became an issue once we saw the files with no spaces. Letter frequency analysis worked well though.
    b. Mono-Alphabetic Substitution was the hardest of the three. In the end the program we developed could only approximately get the correct solution and needed some correction at the end.
    c. Vigenère was originally difficult because we used the key length solving algorithms found on Wikipedia. These were not very accurate, so it caused issues that made the rest of the program difficult to solve. Finally, we found the Twist+ algorithm which could very accurately find the key size consistently. After the key size the rest was the same as the Caesar cipher, so not much work.

Caesar:

Description:

The Caesar Cipher works by rotating every character in the plaintext by a specific amount to produce the ciphertext. For example, if the key is 5 then `a` -> `f`, `z` -> `e`, etc.

Encryption Implementation:

To implement this in Python, loop through the entire input and replace each character with the appropriate offset character. This can be done by creating a dictionary with the plaintext characters as keys and the corresponding ciphertext characters as values.

Decryption Implementation:

For each of the 26 possible shifts (including no shift) get the letter frequencies statistics and compare them with the baseline English letter frequencies. Pick whichever one has the closest correlation. Without any adjustment this works for all the provided ciphertexts.

Vigenère:

Description:

Vigenère works by having some key phrase that is used to offset all the characters in the plaintext. If the key phrase is shorter than the plaintext then it can be added to itself until they are both the same length (i.e. key: ‘dog’ text: ‘Hello, world!’, long_key: ‘dogdo, gdogd!`). The longer key skips over non-letter characters and shifts letters by whatever letter is in the key (i.e. `d` is a shift of three).

Encryption Implementation:

To implement this in Python take the key and make it the same length as the plaintext. Then iterate through both the plaintext and the long key at the same time. Every time you encounter a letter shift it by the current letter from the key.

Decryption Implementation:

The way we implemented the cracking was using the Twist+ algorithm to find the key length then using letter frequency analysis to find each of the letters of the key.

Twist+

Twist+ is really complicated so we’ve included the link to the research paper about it.

Mono-Alphabetic Substitution:

Description:

Mono-Alphabetic Substitution works by mapping each character from the plaintext to a random character in the ciphertext. Each letter consistently maps to another letter but there is no order.

Encryption Implementation:

The key for a Mono-Alphabetic Substitution Cipher is the shuffled alphabet which maps its plaintext to cipher text. In order to implement this, we can create a dictionary that has plaintext characters as its keys and the ciphertext characters as its values. Then just iterate through the input and replace all the letters.

Decryption Implementation:

This one was tricky to decrypt. Even using letter frequency analysis, it is very difficult to find the right mapping because there are a total of 26! possibilities. The approach we used was a genetic algorithm. The way it works is to generate 50 totally random keys and improve on each of them until we can't get any better. We do this by randomly swapping two characters in the key and checking to see if that makes the resulting ciphertext more "Englishly". In this case we checked by comparing the trigraph frequency between the decoded text and the expected English values. Then we keep swapping characters until we have made 5000 useless swaps in a row. At this point we can conclude that we probably won't get much better, so we move to the next key. Once we have all 50 keys, we can take the best one of the set and present it for evaluation. At the end the program provides the decoded text according to the key and lets the user swap letters manually until it gets to a better state.

Solutions:

caesar_easy_encrypted.txt -> 8

Scrooge took his melancholy dinner in his usual melancholy tavern; and having read all the newspapers, and beguiled the rest of the evening with his banker's-book, went home to bed. He lived in chambers which had once belonged to his deceased partner. They were a gloomy suite of rooms, in a lowering pile of building up a yard, where it had so little business to be, that one could scarcely help fancying it must have run there when it was a young house, playing at hide-and-seek with other houses, and forgotten the way out again. It was old enough now, and dreary enough, for nobody lived in it but Scrooge, the other rooms being all let out as offices. The yard was so dark that even Scrooge, who knew its every stone, was fain to grope with his hands. The fog and frost so hung about the black old gateway of the house, that it seemed as if the Genius of the Weather sat in mournful meditation on the threshold.

caesar_easy_2_encrypted.txt -> 15

Oneofthephenomenawhichhadpeculiarlyattractedmyattentionwasthestructureofthehumanframe,and,indeed,anyanimalenduedwithlife.Whence,Ioftenaskedmyself,didtheprincipleoflifeproceed?Itwasaboldquestion,andonewhichhaseverbeenconsideredasamystery;yetwithhowmanythingsareweuponthebrinkofbecomingacquainted,ifcowardiceorcarelessnessdidnotrestrainourinquiries.Irevolvedthesecircumstancesinmymindanddeterminedthenceforthtoapplymyselfmorepartic ularlytothosebranchesofnaturalphilosophywhichrelatetophysiology.UnlessIhadbeenanimatedby analmostsupernaturalenthusiasm,myapplicationtothisstudywouldhavebeenirksomeandalmosti ntolerable.Toexaminethecausesoflife,wemustfirsthaverecoursetodeath.Ibecameacquaintedwit hthescienceofanatomy,butthiswasnotsufficient;Imustalsoobservethenaturaldecayandcorruptio nofthehumanbody.Inmyeducationmyfatherhadtakenthegreatestprecautionsthatmymindshould beimpressedwithnosupernaturalhorrors.Idonoteverremembertohavetrembledatatataleofsuperstitionortohavefearedtheapparitionofaspirit.Darknesshadnoeffectuponmyfancy,andachurchyardw astomemerelythereceptacleofbodiesdeprivedoflife,which,frombeingtheseatofbeautyandstrengt h,hadbecomefoodfortheworm.NowIwasledtoexaminethecauseandprogressofthisdecayandforce dtospenddaysandnightsinvaultsandcharnel-
houses.Myattentionwasfixeduponeveryobjectthemostinsupportabletothedelicacyofthehumanf eelings.Isawhowthefineformofmanwasdegradedandwasted;Ibeheldthecorruptionofdeathsucce edtothebloomingcheekoflife;Isawhowtheworminheritedthewondersoftheeyeandbrain.Ipaused, examiningandanalysingalltheminutiaeofcausation,asexemplifiedinthechangefromlifetodeath,an ddeathtolife,untilfromthemidstofthisdarknessasuddenlightbrokeinuponme—
alightsobrilliantandwondrous,yetsosimple,thatwhileIbecamedizzywiththeimmensityoftheprosp ectwhichitillustrated,Iwassurprisedthatamongsomanymenofgeniuswhohaddirectedtheirinquirie stowardsthesamescience,thatIaloneshouldbereservedtodiscoversoastonishingasecret.

caesar_hard_encrypted.txt -> 20

But this is not a claim that Man should stay always youthful. Supposing that that famous Spaniard, landing upon Florida's coral strands, had found that mythical Fountain of Youth; what a calamity for mankind! A world without maturity of thought; without man's full-grown muscular ability to construct mighty buildings, railroads and ships; a world without authors,

doctors, savants, musicians; nothing but Youth! I can think of but a solitary approval of such a condition; for such a horror as war would not,—could not occur; for a child is, naturally, a small bunch of sympathy. I know that boys will "scrap;" also that "spats" will occur amongst girls; but, at such a monstrosity as killings by bombing towns, sinking ships, or mass annihilation of marching troops, childhood would stand aghast. Not a tiny bird would fall; nor would any form of gun nor facility for manufacturing it, insult that almost Holy purity of youthful thought. Anybody who knows that wracking sorrow brought upon a child by a dying puppy or cat, knows that childhood can show us that our fighting, our policy of "a tooth for a tooth," is abominably wrong.

caesar_hard_2_encrypted.txt -> 3

But this is not a claim that Man should stay always youthful. Supposing that that famous Spaniard, landing upon Florida's coral strands, had found that mythical Fountain of Youth; what a calamity for mankind! A world without maturity of thought; without man's full-grown muscular ability to construct mighty buildings, railroads and ships; a world without authors, doctors, savants, musicians; nothing but Youth! I can think of but a solitary approval of such a condition; for such a horror as war would not,—could not occur; for a child is, naturally, a small bunch of sympathy. I know that boys will "scrap;" also that "spats" will occur amongst girls; but, at such a monstrosity as killings by bombing towns, sinking ships, or mass annihilation of marching troops, childhood would stand aghast. Not a tiny bird would fall; nor would any form of gun nor facility for manufacturing it, insult that almost Holy purity of youthful thought. Anybody who knows that wracking sorrow brought upon a child by a dying puppy or cat, knows that childhood can show us that our fighting, our policy of "a tooth for a tooth," is abominably wrong.

mono_easy_encrypt.txt -> zyorbunxlpmjtisqvgfkwcahde

MEGALONYX

MEGALONYX (GREAT CLAW) IS THE GREEK NAME FOR ANOTHER OF THE GIANT GROUND SLOTHS. THE NAME WAS PROPOSED BY THOMAS JEFFERSON IN 1797, BASED ON FOSSIL SPECIMENS FOUND IN A CAVE IN WEST VIRGINIA. MEGALONYX JEFFERSONII, OF THE FAMILY MEGALONYCHIDAE, WAS A LARGE, HEAVILY BUILT ANIMAL ABOUT 8 TO 10 FEET (2.53 M) LONG. ITS MAXIMUM WEIGHT MAY HAVE BEEN AS MUCH AS 800 POUNDS. THIS IS MEDIUM-SIZED AMONG THE GIANT GROUND SLOTHS.

LIKE OTHER GROUND SLOTHS IT HAD A BLUNT SNOUT, MASSIVE JAW ND LARGE, PEG-LIKE TEETH. THE HIND LIMBS WERE PLANTIGRADE (FLAT-FOOTED) AND THIS, ALONG WITH ITS STOUT TAIL, ALLOWED IT TO REAR UP INTO A SEMI-ERECT POSITION TO FEED ON TREE LEAVES. THE FORELIMBS HAD THREE HIGHLY DEVELOPED CLAWS THAT WERE PROBABLY USED TO STRIP LEAVES AND TEAR OFF BRANCHES.

M. JEFFERSONII WAS APPARENTLY THE MOST WIDE-RANGING GIANT GROUND SLOTH. FOSSILS ARE KNOWN FROM MANY PLEISTOCENE SITES IN THE UNITED STATES, INCLUDING MOST OF

THE STATES EAST OF THE ROCKY MOUNTAINS AS WELL AS ALONG THE WEST COAST. IT WAS THE ONLY GROUND SLOTH TO RANGE AS FAR NORTH AS THE PRESENT-DAY YUKON AND ALASKA.

IN 2010, THE FIRST SPECIMEN EVER FOUND IN COLORADO WAS DISCOVERED AT THE ZIEGLER RESERVOIR SITE NEAR SNOWMASS VILLAGE (IN THE ROCKY MOUNTAINS AT AN ELEVATION OF 8,874 FEET). WHY THE GIANT GROUND SLOTH, AS WITH OTHER MEGAFAUNA OF THE MIOCENE EPOCH, GREW TO SUCH ENORMOUS SIZE IS A MYSTERY.

BESIDES THEIR BULK, THESE SLOTHS WERE DISTINGUISHED BY THEIR SIGNIFICANTLY LONGER FRONT THAN HIND LEGS, A CLUE THAT THEY USED THEIR LONG FRONT CLAWS TO ROPE IN COPIOUS AMOUNTS OF VEGETATION. AS BIG AS IT WAS, THOUGH, MEGALONYX WAS A MERE PUP COMPARED TO THE TRULY GIANT MEGATHERIUM.

MEGATHERIUM AND MEGALONYX ARE DISTANT RELATIVES OF TODAYS MODERN TWO- AND THREE-FINGERED SLOTHS THAT LIVE IN CENTRAL AND SOUTH AMERICA.

mono_medium_encrypt.txt -> vimexbphstwgfjclqdyarnozuk

TWAS BRILLIG, AND THE SLITHY TOVES
    DID GYRE AND GIMBLE IN THE WABE:
ALL MIMSY WERE THE BOROGOVES,
    AND THE MOME RATHS OUTGRABE.

BEWARE THE JABBERWOCK, MY SON!
    THE JAWS THAT BITE, THE CLAWS THAT CATCH!
BEWARE THE JUBJUB BIRD, AND SHUN
    THE FRUMIOUS BANDERSNATCH!

HE TOOK HIS VORPAL SWORD IN HAND;
    LONG TIME THE MANXOME FOE HE SOUGHT
SO RESTED HE BY THE TUMTUM TREE
    AND STOOD AWHILE IN THOUGHT.

AND, AS IN UFFISH THOUGHT HE STOOD,
    THE JABBERWOCK, WITH EYES OF FLAME,
CAME WHIFFLING THROUGH THE TULGEY WOOD,
    AND BURBLED AS IT CAME!

ONE, TWO! ONE, TWO! AND THROUGH AND THROUGH
    THE VORPAL BLADE WENT SNICKER-SNACK!
HE LEFT IT DEAD, AND WITH ITS HEAD
    HE WENT GALUMPHING BACK.

AND HAST THOU SLAIN THE JABBERWOCK?

COME TO MY ARMS, MY BEAMISH BOY!
O FRABJOUS DAY! CALLOOH! CALLAY!
HE CHORTLED IN HIS JOY.

TWAS BRILLIG, AND THE SLITHY TOVES
DID GYRE AND GIMBLE IN THE WABE:
ALL MIMSY WERE THE BOROGOVES,
AND THE MOME RATHS OUTGRABE.

vigenere_easy_encrypt.txt -> mfxjt

the room displayed a modest and pleasant color-scheme, after one of the best standard designs of the decorator who “did the interiors” for most of the speculative-builders’ houses in zenith. the walls were gray, the woodwork white, the rug a serene blue; and very much like mahogany was the furniture—the bureau with its great clear mirror, mrs. babbitt’s dressing-table with toilet-articles of almost solid silver, the plain twin beds, between them a small table holding a standard electric bedside lamp, a glass for water, and a standard bedside book with colored illustrations—what particular book it was cannot be ascertained, since no one had ever opened it. the mattresses were firm but not hard, triumphant modern mattresses which had cost a great deal of money; the hot-water radiator was of exactly the proper scientific surface for the cubic contents of the room. the windows were large and easily opened, with the best catches and cords, and holland roller-shades guaranteed not to crack. it was a masterpiece among bedrooms, right out of cheerful modern houses for medium incomes. only it had nothing to do with the babbitts, nor with any one else. if people had ever lived and loved here, read thrillers at midnight and lain in beautiful indolence on a sunday morning, there were no signs of it. it had the air of being a very good room in a very good hotel. one expected the chambermaid to come in and make it ready for people who would stay but one night, go without looking back, and never think of it again.

vigenere_medium_encrypt.txt -> dbdubdfbf

thebellman’sspeech
thebellmanhimselftheyallpraisedtotheskies—
suchacarriage,sucheaseandsuchgrace!
suchsolemnity,too!onecouldseehewaswise,
themomentonelookedinhisface!
hehadboughtalargemaprepresentingthesea,
withouttheleastvestigeofland:
andthecrewweremuchpleasedwhentheyfoundittobe
amaptheycouldallunderstand.
“what’sthegoodofmercator’snorthpolesandequators,
tropics,zones,andmeridianlines?”
sothebellmanwouldcry:andthecrewwouldreply
“theyaremerelyconventionalsigns!
“othermapsaresuchshapes,withtheirislandsandcapes!

butwe'vegotourbravecaptaintothank:"
(sothecrewwouldprotest)"thathe'sboughtusthebest—
aperfectandabsoluteblank!"
thiswascharming,nodoubt;buttheyshortlyfoundout
thatthecaptaintheytrustedsowell
hadonlyonenotionforcrossingtheocean,
andthatwastotinglehisbell.
hewasthoughtfulandgrave—buttheordershegave
wereenoughtobewilderacrew.
whenhecried"steertostarboard,butkeepherheadlarboard!"
whatonearthwasthehelmsmantodo?
thenthebowspritgotmixedwiththeruddersometimes:
athing,asthebellmanremarked,
thatfrequentlyhappensintropicalclimes,
whenavesselis,sotospeak,"snarked."
buttheprincipalfailingoccurredinthesailing,
andthebellman,perplexedanddistressed,
saidhehadhoped,atleast,whenthewindblewdueeast,
thattheshipwouldnottravelduewest!
butthedangerwaspast—theyhadlandedatlast,
withtheirboxes,portmanteaus,andbags:
yetatfirstsightthecrewwerenotpleasedwiththeview,
whichconsistedofchasmsandcrags.
thebellmanperceivedthattheirspiritswerelow,
andrepeatedinmusicaltone
somejokeshehadkeptforaseasonofwoe—
butthecrewwoulddonothingbutgroan.
heservedoutsomegrogwithaliberalhand,
andbadethemsitdownonthebeach:
andtheycouldnotbutownthattheircaptainlookedgrand,
ashestoodanddeliveredhisspeech.
"friends,romans,andcountrymen,lendmeyourears!"
(theywereallofthemfondofquotations:
sotheydranktohishealth,andtheygavehimthreecheers,
whileheservedoutadditionalrations).
"wehavesailedmanymonths,wehavesailedmanyweeks,
(fourweekstothemonthyoumaymark),
butneverasyet('tisyourcaptainwhospeaks)
havewecaughttheleastglimpseofasnark!
"wehavesailedmanyweeks,wehavesailedmanydays,
(sevendaystotheweekiallow),
butasnark,onthewhichwemightlovinglygaze,
wehaveneverbeheldtillnow!
"come,listen,mymen,whileitellyouagain

thefiveunmistakablemarks
bywhichyoumayknow,wheresoeveryougo,
thewarrantedgenuinesnarks.
“letustaketheminorder.thefirstisthetaste,
whichismeagreandhollow,butcrisp:
likeacoatthatisrathertootightinthewaist,
withaflavourofwill-o’-the-wisp.
“itshabitofgettinguplateyou’llagree
thatitcarriestoofar,whenisay
thatitfrequentlybreakfastsatfive-o’clocktea,
anddinesonthefollowingday.
“thethirdisitsslownessintakingajest.
shouldyouhappentoventureonone,
itwillsighlikeathingthatisdeeplydistressed:
anditalwayslooksgraveatapun.
“thefourthisitsfondnessforbathing-machines,
whichisconstantlycarriesabout,
andbelievesthattheyaddtothebeautyofscenes—
asentimentopentodoubt.
“thefifthisambition.itnextwillberight
todescribeeachparticularbatch:
distinguishingthosethathavefeathers,andbite,
andthosethathavewhiskers,andscratch.
“for,althoughcommonsnarksdonomannerofharm,
yet,ifeelitmydutytosay,
somearеboojums—”thebellmanbrokeoffinalarm,
forthebakerhadfaintedaway.

vigenere_hard_encrypt.txt -> tobsltbsfsfbm
acanthaceae
achariaceae
achatocarpaceae
acoraceae
actinidiaceae
adoxaceae
aextoxicaceae
aizoaceae
akaniaceae
alismataceae
alseuosmiaceae
alstroemeriaceae
altingiaceae
alzateaceae
amaranthaceae

amaryllidaceae
amborellaceae
anacardiaceae
anarthriaceae
ancistrocladaceae
anisophylleaceae
annonaceae
aphanopetalaceae
aphloiaceae
apiaceae
apocynaceae
apodanthaceae
aponogetonaceae
aquifoliaceae
araceae
araliaceae
arecaceae
argophyllaceae
aristolochiaceae
asparagaceae
asteliaceae
asteropeiaceae
atherospermataceae
austrobaileyaceae
balanopaceae
balanophoraceae
balsaminaceae
barbeuiaceae
barbeyaceae
basellaceae
bataceae
begoniaceae
berberidaceae
berberidopsidaceae
betulaceae
biebersteiniaceae
bignoniaceae
bixaceae
blandfordiaceae
bonnetiaceae
boraginaceae
boryaceae
brassicaceae
bromeliaceae

brunelliaceae
bruniaceae
burmanniaceae
burseraceae
butomaceae
buxaceae
byblidaceae
cabombaceae
cactaceae
calceolariaceae
calophyllaceae
calycanthaceae
calyceraceae
campanulaceae
campyneumataceae
canellaceae
cannabaceae
cannaceae
capparaceae
caprifoliaceae
cardiopteridaceae
caricaceae
carlemanniaceae
caryocaraceae
caryophyllaceae
casuarinaceae
celastraceae
centrolepidaceae
centroplacaceae
cephalotaceae
ceratophyllaceae
cercidiphyllaceae
chloranthaceae
chrysobalanaceae
circaeasteraceae
cistaceae
cleomaceae
clethraceae
clusiaceae
colchicaceae
columelliaceae
combretaceae
commelinaceae
compositae

connaraceae
convolvulaceae
coriariaceae
cornaceae
corsiaceae
corynocarpaceae
costaceae
crassulaceae
crossosomataceae
crypteroniaceae
ctenolophonaceae
cucurbitaceae
cunoniaceae
curtisiaceae
cyclanthaceae
cymodoceaceae
cynomoriaceae
cyperaceae
cyrillaceae
cytinaceae
daphniphyllaceae
dasypogonaceae
datiscaceae
degeneriaceae
diapensiaceae
dichapetalaceae
didiereaceae
dilleniaceae
dioncophyllaceae
dioscoreaceae
dipentodontaceae
dipterocarpaceae
dirachmaceae
doryanthaceae
droseraceae
drosophyllaceae
ebenaceae
ecdeiocoleaceae
elaeagnaceae
elaeocarpaceae
elatinaceae
emblingiaceae
ericaceae
eriocaulaceae

erythroxylaceae
escalloniaceae
eucommiaceae
euphorbiaceae
euphroniaceae
eupomatiaceae
eupteleaceae
fagaceae
flagellariaceae
fouquieriaceae
frankeniaceae
garryaceae
geissolomataceae
gelsemiaceae
gentianaceae
geraniaceae
gesneriaceae
gisekiaceae
gomortegaceae
goodeniaceae
goupiaceae
griseliniaceae
grossulariaceae
grubbiaceae
gunneraceae
gyrostemonaceae
haemodoraceae
halophytaceae
haloragaceae
hamamelidaceae
hanguanaceae
haptanthaceae
heliconiaceae
helwingiaceae
hernandiaceae
himantandraceae
huaceae
humiriaceae
hydatellaceae
hydnoraceae
hydrangeaceae
hydrocharitaceae
hydroleaceae
hydrostachyaceae

hypericaceae
hypoxidaceae
icacinaceae
iridaceae
irvingiaceae
iteaceae
ixioliriaceae
ixonanthaceae
joinvilleaceae
juglandaceae
juncaceae
juncaginaceae
kirkiaceae
koeberliniaceae
krameriaceae
lacistemataceae
lactoridaceae
lamiaceae
lanariaceae
lardizabalaceae
lauraceae
lecythidaceae
leguminosae
lentibulariaceae
lepidobotryaceae
liliaceae
limeaceae
limnanthaceae
linaceae
linderniaceae
loasaceae
loganiaceae
lophiocarpaceae
loranthaceae
lowiaceae
lythraceae
magnoliaceae
malpighiaceae
malvaceae
marantaceae
marcgraviaceae
martyniaceae
mayacaceae
melanthiaceae

melastomataceae
meliaceae
melianthaceae
menispermaceae
menyanthaceae
metteniusaceae
misodendraceae
mitrastemonaceae
molluginaceae
monimiaceae
montiaceae
montiniaceae
moraceae
moringaceae
muntingiaceae
musaceae
myodocarpaceae
myricaceae
myristicaceae
myrothamnaceae
myrtaceae
nartheciaceae
nelumbonaceae
nepenthaceae
neuradaceae
nitrariaceae
nothofagaceae
nyctaginaceae
nymphaeaceae
ochnaceae
olacaceae
oleaceae
onagraceae
oncothecaceae
opiliaceae
orchidaceae
orobanchaceae
oxalidaceae
paeoniaceae
pandaceae
pandanaceae
papaveraceae
paracryphiaceae
passifloraceae

paulowniaceae
pedaliaceae
penaeaceae
pentadiplandraceae
pentaphragmataceae
pentaphylacaceae
penthoraceae
peraceae
peridiscaceae
petermanniaceae
petrosaviaceae
philesiaceae
philydraceae
phrymaceae
phyllanthaceae
phyllonomaceae
physenaceae
phytolaccaceae
picramniaceae
picrodendraceae
piperaceae
pittosporaceae
plantaginaceae
platanaceae
plocospermataceae
plumbaginaceae
poaceae
podostemaceae
polemoniaceae
polygalaceae
polygonaceae
pontederiaceae
portulacaceae
posidoniaceae
potamogetonaceae
primulaceae
proteaceae
putranjivaceae
quillajaceae
rafflesiaceae
ranunculaceae
rapateaceae
resedaceae
restionaceae

rhabdodendraceae
rhamnaceae
rhipogonaceae
rhizophoraceae
roridulaceae
rosaceae
rousseaceae
rubiaceae
ruppiaceae
rutaceae
sabiaceae
salicaceae
salvadoraceae
santalaceae
sapindaceae
sapotaceae
sarcobataceae
sarcolaenaceae
sarraceniaceae
saururaceae
saxifragaceae
scheuchzeriaceae
schisandraceae
schlegeliaceae
schoepfiaceae
scrophulariaceae
setchellanthaceae
simaroubaceae
simmondsiaceae
siparunaceae
sladeniaceae
smilacaceae
solanaceae
sphaerosepalaceae
sphenocleaceae
stachyuraceae
staphyleaceae
stegnospermataceae
stemonaceae
stemonuraceae
stilbaceae
strasburgeriaceae
strelitziaceae
stylidiaceae

styracaceae
surianaceae
symplocaceae
talinaceae
tamaricaceae
tapisciaceae
tecophilaeaceae
tetrachondraceae
tetramelaceae
tetrameristaceae
theaceae
thomandersiaceae
thurniaceae
thymelaeaceae
ticodendraceae
tofieldiaceae
torricelliaceae
tovariaceae
trigoniaceae
triuridaceae
trochodendraceae
tropaeolaceae
typhaceae
ulmaceae
urticaceae
vahliaceae
velloziaceae
verbenaceae
violaceae
vitaceae
vivianiaceae
vochysiaceae
winteraceae
xanthorrhoeaceae
xeronemataceae
xyridaceae
zingiberaceae
zosteraceae
zygophyllaceae

Code:

caesar.py

```python
# This is for testing, not for actual code
# ---------------
import logging
# ---------------
from util import chi_squared, command_line_process, file_decode, convert
import string


# guess the cipher key using chi squared values
def caesar_chi(text: str, get_rot_3: bool=False) -> str:
    chi_values = []
    key = string.ascii_lowercase
    for i in range(26):
        new_text = convert(key, text)
        chi_val = chi_squared(new_text)
        chi_values.append((new_text, chi_val))
        logging.debug(f"\tkey: {i:2}\tchi: {chi_val:.2f}")
        key = key[1:] + key[:1]

    min_index = chi_values.index(min(chi_values, key=lambda x: x[1]))
    if get_rot_3:
        return chr(ord('a') + min_index)
    return key[min_index:] + key[:min_index]


if __name__ == "__main__":
    # get the files to decode
    files = command_line_process("caesar.log")

    # decode them
    outputs = [file_decode(file, caesar_chi) for file in files]

    # print the outputs
    print("\n---------\n|Results|\n---------\n")
    with open("caesar.txt", "w") as f:
        for file, key, text in outputs:
            output = f"{file}\tkey: {key}\n{text}\n"
            print(output)
            f.write(output)
```

data.py

*This is like 400 pages long in a word document, but it contains arrays and dictionaries for English letter frequencies and letter triplets.

mono-alphabetic.py

```
import logging
from multiprocessing import Pool
from os import cpu_count

from util import command_line_process, file_decode, convert
import data
import string
import random

CIPHER_TEXT: str = "boof"
KEYS: int = 50
KEY_STEPS: int = 5000

def eval(text: str) -> float:
    score = 0

    # i is the right side of the trigram
    for i in range(3, len(text)):
        # isolate the trigram
        trigram = text[i-3:i]
        # if the trigram is in the dictionary, increase score
        if trigram in data.trigrams:
            score += data.trigrams[trigram]
    return score


def genetic_key(key_attempt: int) -> tuple[float, str]:
    random.seed(key_attempt)

    # start with a random key
    key = list(string.ascii_lowercase)
    random.shuffle(key)

    # evaluate the key
    cur_score = eval(convert(key, CIPHER_TEXT))

    # go until we get 5000 swaps that don't improve score
    i = 0
    while i < KEY_STEPS:
        index1, index2 = random.sample(range(26), 2)
        new_key = key.copy()
        new_key[index1], new_key[index2] = key[index2], key[index1]

        new_score = eval(convert(new_key, CIPHER_TEXT))
```

```
        if new_score > cur_score:
            cur_score = new_score
            key = new_key
            i = 0
        i += 1

    logging.debug(f"Attempt {key_attempt:2}: {key} with score
{cur_score:.2f}")
    print(f"[PROGRESS] {key_attempt}/{KEYS}")

    return cur_score, "".join(key)


def initializer(text: str):
    global CIPHER_TEXT
    CIPHER_TEXT = text


def mono_solver(text: str) -> str:
    # run the function `genetic_key` in parallel
    logging.debug(f"Running `genetic_key` in parallel with {cpu_count()}
threads.")
    pool = Pool(cpu_count(), initializer, (text,))
    results = pool.map(genetic_key, range(1, KEYS))

    best_score, best_key = max(results, key=lambda x: x[0])

    logging.debug(f"Best key: {best_key} with score {best_score:.2f}")
    return best_key


if __name__ == "__main__":
    # get the files from the command line
    files = command_line_process("mono.log")

    # decode them
    outputs = [file_decode(file, mono_solver) for file in files]

    # print the outputs
    print("\n---------\n|Results|\n---------\n")
    with open("mono.txt", "w") as f:
        for file, key, text in outputs:
            output = f"{file}\tkey: {key}\n{text}\n"
            print(output)
            f.write(output)
```

util.py

```
# For dealing with the command line
# -------------------------
import sys
import logging
import os.path as osp
# -------------------------
# For making the code look nicer
# -------------------------
from typing import Union, Any, Callable
# -------------------------

from data import english_expected
import string
import math

ALPHABET = string.ascii_lowercase

def text_process(nonplaintext: str) -> str:
    return "".join([c.lower() for c in nonplaintext if c.isalpha()])


def vigenere_encrypt(key: str, text: str) -> str:
    # preprocess the key and the text
    plaintext = text_process(text)
    key = key.lower()

    # make the key as long as the text by tiling it
    long_key = key * (len(plaintext) // len(key)) + key[:len(plaintext) %
len(key)]

    # generate the cipher text
    ciphertext = []
    for i in range(len(plaintext)):
        plainchar = plaintext[i]
        keychar = long_key[i]
        print(plainchar, keychar)
        ciphernum = (ALPHABET.index(plainchar) + ALPHABET.index(keychar)) %
26
        ciphertext.append(ALPHABET[ciphernum])

    return "".join(ciphertext)


def vigenere_decrypt(key: str, ciphertext: str):
```

```
    # tile the key to be as long as the cipher text
    key_str = key * (len(ciphertext) // len(key)) + key[:len(ciphertext) %
len(key)]
    decrypted = []

    j = 0
    for i in range(len(ciphertext)):
        if ciphertext[i].isalpha():
            # get the 0-25 representation the letter
            chr_num = ord(ciphertext[i].lower()) - ord('a')
            chr_num -= ord(key_str[j]) - ord('a')

            # make sure it isn't negative
            if chr_num < 0:
                chr_num += 26

            if ciphertext[i].isupper():
                decrypted.append(chr(chr_num + ord('A')))
            else:
                decrypted.append(chr(chr_num + ord('a')))
            j += 1
        else:
            decrypted.append(ciphertext[i])

    return "".join(decrypted)

def rot_n_str(n: int, text: str) -> str:
    key = string.ascii_lowercase
    key = key[n:] + key[:n]
    return convert(key, text)


def convert(key: str, text: str) -> str:
    if isinstance(key, list):
        key = "".join(key)
    key_2 = key + key.upper()
    mapping = dict(zip(key_2, string.ascii_letters))
    return "".join([mapping.get(c, c) for c in text])


def chi_squared(input_text: str, difference: bool=True) -> Union[float,
list[float]]:
    counts = [0.0] * 26

    plaintext = "".join([c.lower() for c in input_text if c.isalpha()])
```

```
    length = len(plaintext)

    for c in plaintext:
        counts[ord(c) - ord('a')] += 1

    if difference:
        total = 0.0
        for i in range(26):
            total = total + math.pow((counts[i] -
length*english_expected[i]),2)/(length*english_expected[i])
        return total
    else:
        return [x / sum(counts) for x in counts]


def command_line_process(logname: str) -> list[str]:
    # grab all the parts of the commandline
    opts = [opt for opt in sys.argv[1:] if opt.startswith("-")]
    args = [arg for arg in sys.argv[1:] if not arg.startswith("-")]

    # set the logging level according to the cmdline arguments
    if log_opt := [x for x in opts if "log" in x]:
        loglevel = log_opt[0][6:]
        numeric_level = getattr(logging, loglevel.upper(), None)
        if not isinstance(numeric_level, int):
            raise ValueError('Invalid log level: %s' % loglevel)
        logging.basicConfig(level=numeric_level, filename=f"{logname}",
filemode="w")
        print(f"Log level: {loglevel.upper()}\n")
    else:
        print("Log level: WARNING\n")

    # print out cmdline
    logging.debug(args)
    logging.debug(opts)

    existing_files = []
    for arg in args:
        existing_files.append(arg) if osp.exists(arg) else
logging.warning(f"Cannot find {arg}")

    # confirm there is at least one file to decode
    if len(existing_files) == 0:
        print("You have to provide at least one file to decrypt.")
        print(f"\tusage: {logname}.py [--log=...] [file1] [file2] ...")
```

```
        sys.exit(1)

    return existing_files


def file_decode(file: str, decoder: Callable[[str], str], mono=True) ->
tuple[str, str]:
    # process each of the provided files
    with open(file, "r") as f:
        logging.debug(f"{file}:")
        orig_text = f.read()
    key = decoder(orig_text)
    if not key:
        return "", "", ""
    if mono:
        key = correction(orig_text, key)
        return file, key, convert(key, orig_text)
    else:
        return file, key, vigenere_decrypt(key, orig_text)


def string_swap(string: str, index1: int, index2: int) -> str:
    string_list = list(string)
    string_list[index1], string_list[index2] = string_list[index2],
string_list[index1]
    return "".join(string_list)


def correction(text: str, key: str) -> str:
    print(f"key: {key}\n{convert(key, text)}")
    user_in = input("Does the input need changes? [Y/n]: ").lower().strip()
    if user_in == "y":
        while True:
            user_in = input("Enter two letters to swap (ex. `a b`) or `exit`:
").lower().strip()

            if user_in == "exit":
                break
            if len(user_in) != 3 or user_in[1] != " ":
                print("Invalid input.")
                continue

            # get the two letters and their indicies within the key
            letter1, letter2 = user_in.split()
            index1, index2 = ord(letter1) - ord('a'), ord(letter2) - ord('a')
```

```
            # swap the indicies
            key = string_swap(key, index1, index2)

            # reprint the text
            print(f"key: {key}\n{convert(key, text)}")
    return key
```

vigenere.py

```
from caesar import caesar_chi
from util import command_line_process, file_decode, ALPHABET, text_process

MAX_KEY_LEN: int = 13


def twist_alg(ciphertext: str) -> int:
    # get the columns for each of the possible key lengths
    all_key_lengths = {}
    for key_len in range(1, MAX_KEY_LEN+1):
        cols = {}
        for i in range(key_len):
            cols[i] = ciphertext[i::key_len]
        all_key_lengths[key_len] = cols

    # get the letter frequencies
    all_key_frequencies = {}
    for k, cols in all_key_lengths.items():
        # get the frequencies of all the letters in each column
        letter_frequencies = {}
        for i, col in cols.items():
            # get the counts of each letter
            letter_counts = {}
            for letter in col:
                if letter not in letter_counts:
                    letter_counts[letter] = 0
                letter_counts[letter] += 1
            letter_frequencies[i] = letter_counts
        all_key_frequencies[k] = letter_frequencies

    # get all possible letters for each column
    all_key_letters = {}
    for key_len in all_key_frequencies:
        sub_dict = {}
        for num in all_key_frequencies[key_len]:
            letters_in = []
            for letter in all_key_frequencies[key_len][num]:
                letters_in.append(letter[0])
            sub_dict[num] = letters_in
        all_key_letters[key_len] = sub_dict

    # fill in all the frequencies that didn't show up in the column
    all_key_frequencies_complete = all_key_frequencies.copy()
    for c in ALPHABET:
```

```
        for index in all_key_letters:
            for i in all_key_letters[index]:
                if c not in all_key_letters[index][i]:
                    all_key_frequencies_complete[index][i][c] = 0

    # sort the key frequencies in descending order
    all_key_frequencies_complete_sorted = {}
    for index in all_key_frequencies_complete:
        sub_dict = {}
        for i in all_key_frequencies_complete[index]:
            sub_dict[i] =
(sorted(all_key_frequencies_complete[index][i].items(), key=lambda x: x[1],
reverse=True))
        all_key_frequencies_complete_sorted[index] = sub_dict

    # convert all the numbers to percentages
    all_key_percentages = {}
    for i in all_key_frequencies_complete_sorted:
        sub_dict = {}
        for j in all_key_frequencies_complete_sorted[i]:
            percentage_list = []
            if (j - 1) <= (len(ciphertext) % i):
                divisor = (len(ciphertext) // i) + 1
            else:
                divisor = len(ciphertext) // i

            for k in all_key_frequencies_complete_sorted[i][j]:
                tuple_new = (k[0], k[1] / divisor)
                percentage_list.append(tuple_new)
            sub_dict[j] = percentage_list
        all_key_percentages[i] = sub_dict

    # collect all the letter percentages without their letters
    cj = {}
    for i in all_key_percentages:
        final = [0] * 26
        for j in all_key_percentages[i]:
            cj_list = [ k[1] for k in all_key_percentages[i][j] ]
            final = [ final[n] + cj_list[n] for n in range(len(cj_list))]
        cj[i] = final

    # using the twist algorithm
    twists = {}
    for i in cj:
        twist = 0
```

```
        for j in enumerate(cj[i]):
            if j[0] <= 12:
                twist += j[1]
            else:
                twist -= j[1]
        twist *= 100 / i
        twists[i] = twist

    # using the twist+ algorithm
    twistplus = {}
    twistlist = list(twists.values())
    for i in twistlist:
        subtact = 0
        for j in range(twistlist.index(i)):
            subtact += (twistlist[j] / twistlist.index(i))
        number = i - subtact
        if twistlist.index(i) != 0:
            twistplus[twistlist.index(i) + 1] = number

    def twistplus_key(d: dict) -> int:
        mode_val = 0
        for i, j in d.items():
            if j > mode_val:
                mode = i
                mode_val = j
        return mode

    return twistplus_key(twistplus)


def vigenere_solver(ciphertext: str) -> str:
    plain_ciphertext = text_process(ciphertext)
    key_len = twist_alg(plain_ciphertext)

    print(f"Key Length: {key_len}")

    key_letters = []
    for i in range(key_len):
        # get the text to analyze
        col = plain_ciphertext[i::key_len]
        key_letters.append(caesar_chi(col, True))

    return "".join(key_letters)
```

```python
if __name__ == "__main__":
    # get the files from the command line
    files = command_line_process("vigenere.log")

    # decode them
    outputs = [file_decode(file, vigenere_solver, mono=False) for file in
files]

    # print the outputs
    print("\n---------\n|Results|\n---------\n")
    with open("vigenere.txt", "w") as f:
        for file, key, text in outputs:
            output = f"{file}\tkey: {key}\n{text}\n"
            print(output)
            f.write(output)
```