Q1) The key is SUNTZU.

First, I used the kasiski method to figure out the size of the key. Then I use all the possible combinations of key to figure out the plaintext.

```
# Here
#                              ciphertext                              =
'OBRGXIMYAZZAWCATBNMUYYHAZNVGFCXPVVSIJSVLKIFAVGBIECAZSBWGRGRQWUCHMMOCYE\
# FLGQQNKFSHQMGYALNKCIJQVEKVWXNFOYFYQBESGOYTXMAYTXSISNBPMSGOJBKFWRUTTMLS\
# BNQMLLRGFNZUAWHZLBRVZGHUVZMCKJEHSLSWGXCNZYEXRIMLPXRIXNUXRNSNRPHFDHBMAY\
# WKHTKNGNUXRNJUVGMYNYEYNLYYGPGYFSBNQQWUCHMMSLRWTFDYQRNOJUEWNLVUZIDHWXLH\
# TLKNEXMALBRQGUMMGXCUFXLHTLLLRTROJYFIDHLIGADLUBVXENSCALVCDFFIQCFAHISILU\
# XXZXNUAMZAWISRNOJYKMQYECGRSBWHAHLUFBBPDPWLJBRYOCYEAYSVYXSISPRKSNZYPHMM\
# WKHXMWWMGAZNEOFMDHKORMGOKNUHTAZQRAZPWBRTQXGZFMTJAXUTRNWCAPZLUFRODLFYFL\
# GUKHRODLTYRGRYWHNLRIUCNMDXOCGAKIFAQXKUQMVGZFDBVLSIJSGADLWCFGNCFMGTMWWI\
# STBIMHGKXBSPVGFVWHRYHNWXSKNGHLBENHYYQPZLXUEXNHDSBGDQZIXGNQKNUXCCKUFMQI\
# MMRYEYUNFHEUDIAZVUJWNGQYSFVSDNZYFNOLWGRBLJGLGTMWWISKZJAXVMXCFVEBMAAHTB\
# SNGUPENMWCGBRIFFLHMYOBBBRNZIEHTAZFLTBKMUVGSYVQVMGNZYROHFKISPZLOBBVZHLB\
# BKNOYBYRTHVYELSUFXGADJJISBSUTFRPZSGZPTQLQCAZHNGHGADMCCYEEODARGDLSFQHDM\
# FIGKZCKYNLDWGHQEDPQHRBSBWLNKDBAMFNOJDSJTFIFMYHZXWXZHQYLBNGSQAWRHMWWQNK\
# HMVYPEZLWXUXVCDFAHSQSMGXOLWWVHTMLCZXHHOUVMHHYZBKQYAHSHQWWGRGSMFIEPHFDB\
# RMTLFBVLZLESOTBEXIEYQYKBFNOJDCRLAOLWEHRMWMGADYFYZRRZJIAMHYJQVMGIMNQXKU\
# QNUXUUDORHENAGRMGULCFUDCFANEHNLFRTGYSXBYXIMLBIOIFYAMGUKWBNMNWXSHQGGLRM\
# GUFYVMGYJHHFDLAWNEROHYEBNLANLHQNZYABBYKNPTKWMFNMHIFMJBSBJYTTQXLIPHLGAM\
# FTQCSN'

# #calculating the variance of the string using the formula
# def var(string):
#     count=[0]*26
#     for i in range(0,len(string)):
#         count[ord(string[i].capitalize())-ord('A')]+=1
#     assert sum(count)==len(string)
#     s=0
#     for i in range(0,26):
#         s+=(float(count[i])/len(string) - float(1)/26)**2
#     return s/26
# # just the kasiski to figure out the best key size.
# def kasiski(cipher, length):
#     l=[]
#     for j in range(0,length):
#         l.append("")
#     for j in range(0,len(cipher)):
#         l[j%length]+=cipher[j]
#     s=0
#     for k in l:
#         s+=var(k)
#     return s/length
```

```
for i in range(1,10):
#                                            print("key_length=="+str(i)+",
variance="+str(kasiski(ciphertext,i))+"\n")
# print("When key is 6 using the kakiski method I get the highest population
variance\n")
#key_length==1, variance=0.00012949433537570132

#key_length==2, variance=0.00045688699059466726

#key_length==3, variance=0.00043077436088673985

#key_length==4, variance=0.0005205586239355436

#key_length==5, variance=0.00028329702312441217

#key_length==6, variance=0.0011559381015222505

#key_length==7, variance=0.00034640900714718974

#key_length==8, variance=0.000619305544783434

#key_length==9, variance=0.0005952735732189888

#When key is 6 using the kakiski method I get the highest population
#variance
# vigenere = [""]*6
# for i in range(0, len(ciphertext)):
#     vigenere[i % 6] += ciphertext[i]

# def decrypt(cipher,key,capital=True):
#     decipher=""
#     if capital:
#         for i in range(0,len(cipher)):
#                                    decipher=decipher+chr((ord(cipher[i])-
ord(key[i%len(key)]))%26+ord('A'))
#     else:
#         for i in range(0,len(cipher)):
#                                    decipher=decipher+chr((ord(cipher[i])-
ord(key[i%len(key)]))%26+ord('a'))
#     return decipher
# def freqmax(string,n):
#     count=[0]*26
#     for i in range(0,len(string)):
#         count[ord(string[i].capitalize())-ord('A')]+=1
#     count=sorted(range(len(count)), key=lambda x: count[x],reverse=True)
#     candidate=[]
#     for i in range(0,n):
#         candidate.append(chr((count[i]-4)%26+ord('A')))
#     return candidate
# generate all the possible keys from the max frequency
# def generator(n):
#     keypool = []
#     for i in range(0, len(vigenere)):
#         keypool.append(freqmax(vigenere[i], n))
#     keys = []
#     for a in keypool[0]:
#         for b in keypool[1]:
#             for c in keypool[2]:
#                 for d in keypool[3]:
#                     for e in keypool[4]:
#                         for f in keypool[5]:
#                                 keys.append(a + b + c + d + e + f)
#     return keys
#def selector(n,threshold):
#     keys=generator(n)
#     keylist=[]
#     varlist=[]
#     for key in keys:
#         s=var(decrypt(ciphertext,key))
#         if s>threshold:
#             keylist.append(key)
#             varlist.append(s)
#     return [y for x,y in sorted(zip(varlist,keylist),reverse=True)]

# finallist=selector(3,0.001)
# print("Printing the key with decrypted text")
# for key in finallist:
#     plain=decrypt(ciphertext, key, capital=False)
#     print(key+"\n"+plain+"\n")
```

Using the above code my key is SUNTZU and when decrypted it does make sense
Printing the key with decrypted text
SUNTZU

```
whenyouengageinactualfightingifvictoryislongincomingthemensweaponswillgrow
dullandtheirardorwillbedampenedifyoulaysiegetoatownyouwillexhaustyourstren
gthandifthecampaignisprotractedtheresourcesofthestatewillnotbeequaltothest
rainneverforgetwhenyourweaponsaredulledyourardordampenedyourstrengthexhaus
tedandyourtreasurespentotherchieftainswillspringuptotakeadvantageofyourext
remitythennomanhoweverwisewillbeabletoavertheconsequencesthatmustensuethu
sthoughwehaveheardofstupidhasteinwarclevernesshasneverbeenseenassociatedwi
thlongdelaysinallhistorythereisnoinstanceofacountryhavingbenefitedfromprol
ongedwarfareonlyonewhoknowsthedisastrouseffectsofalongwarcanrealizethesupr
emeimportanceofrapidityinbringingittoacloseitisonlyonewhoisthoroughlyacqua
intedwiththeevilsofwarwhocanthoroughlyunderstandtheprofitablewayofcarrying
itontheskillfulgeneraldoesnotraiseasecondlevyneitherarehissupplywagonsload
edmorethantwiceoncewarisdeclaredhewillnotwastepreciousttimeinwaitingforrein
forcementsnorwillheturnhisarmybackforfreshsuppliesbutcrossestheenemysfront
```

```
ierwithoutdelaythevalueoftimethatisbeingalittleaheadofyouropponenthascount
edformorethaneithernumericalsuperiorityorthenicestcalculationswithregardto
commissariat
```

Q2)

a) 0.0010405667735207099

b) 0.0010069737954353337

c) when key is yz
0.0005425572595902267
when key is xyz
0.00034741510016235287
when key is wxyz
0.0002459193736662644
when key is vwxyz
0.00018148975566557982
when key is uvwxyz
0.0001679715278616378
The significance of population variance is that it gives us an idea about how out of balance the distribution of characters is In English language, all the keys are not evenly distributed as it is also the case in above dictionary that is provided to us. So it becomes very easy to break simple substitution cipher using frequency analysis. Using the vigenere cipher, the original distribution is broken making the distribution to be spread more evenly. Due to this I think the population varience would decrease as well. It might be close to zero.

d) 0.0011098867554911514
0.0011034546954876623
0.0012505199318386133
0.0011889247809577482
0.0013157126756027854
Caesar cipher is the collection of k independent cipher so each character in Caesar cipher is shifted by the same amount, thus causing the population variance to be unchanged. Therefor the population variance remains close to the English characters which is why the result is pretty close to result in b. It is much higher than the values in c.

e) 0.00046995129412711825
0.0005082166002495672
0.0005558574514618472
0.0003314984771028727
I think that this method is a variant of Kasiski attack. In kasiski method we break the ciphertext into certain block lengths and try to find the common occurring patterns. In this we are doing just mathematical stuff instead of manipulating the characters. If we guess the key length wrong, then the average population will be much lesser than normal English levels. But if we guess the key right it will be closer to the population variance of normal English levels around 0.001.

Q3) In DES, we have a 64 bit key which we pass through Permutation Choice PC-1 box to get the key of 56 bits. The 56 bits are then divided into two 28-bit halves; each half is thereafter treated separately. In

successive rounds, both halves are rotated left by one or two bits (specified for each round), and then 48 subkey bits are selected by *Permuted Choice 2* (*PC-2*). So not all the 56 bit key are used an equal number of time.
The bits that are discarded from the key in each round are as follows:
K1: 9 18 22 25 35 38 43 54
K2:10 19 23 26 36 39 44 55
K3: 0 12 21 25 29 38 41 46
K4: 2 14 23 27 31 40 43 48
K5: 1 4 16 25 33 42 45 50
K6: 3 6 18 27 35 44 47 52
K7: 1 5 8 20 37 46 49 54
K8: 3 7 10 22 28 39 48 51
K9: 4 8 11 23 29 40 49 52
K10: 6 10 13 25 31 42 51 54
K11: 8 12 15 27 28 33 44 53
K12: 1 10 14 17 30 35 46 55
K13: 3 12 16 19 29 32 37 48
K14: 5 14 18 21 31 34 39 50
K15: 7 16 20 23 33 36 41 52
K16: 8 17 21 24 34 37 42 53