Final assignment of “Management and Analysis of Physics Datasets”

Group: Lorenzo Borella, Mattia Ceravolo, Marco Giunta, Samuele Pio Lipani

Fun exercise

After some trials, we understood which operator leads to the results reported in the exercise: it is a simple bitwise XOR. The XOR behaviour is shown in the following truth table:

A	B	A^B
0	0	0
0	1	1
1	0	1
1	1	0

The exercise reports integer numbers, therefore it is not straightforward to think about applying a XOR operator bit per bit. In order to do so, one needs to switch to the binary representation of these values. In the tables below we wrote the numbers from 1 to 12 both in integer and binary format.

Decimal Value	Binary Value
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
10	1010
11	1011
12	1100

Here are the explicit calculations for the values of the exercise. As we can see from the tables, **the missing field "?" should be replaced with the value "1"**.

A=1	B=2	A^B=3
1	0	1
0	1	1
0	0	0
0	0	0

A=2	B=5	A^B=7
0	1	1
1	0	1
0	1	1
0	0	0

A=3	B=7	A^B=4
1	1	0
1	1	0
0	1	1
0	0	0

A=4	B=5	A^B=1
0	1	1
0	0	0
1	1	0
0	0	0

A=5	B=9	A^B=12
1	1	0
0	0	0
1	0	1
0	1	1

In [3]:
```python
import numpy as np
import matplotlib.pyplot as plt
import random
import math
```

In [4]:
```python
#data
a=np.arange(1,6)
b=np.array([2,5,7,5,9])

#print  the results
for i in range(len(a)): print(f"",a[i],"XOR",b[i],"=",a[i]^b[i],"\n")
```

```
 1 XOR 2 = 3

 2 XOR 5 = 7

 3 XOR 7 = 4

 4 XOR 5 = 1

 5 XOR 9 = 12
```

Assignment 1: Redundancy

1.1 Write a program which produces four striped data and one parity file using the given input file.

We read the data from the input file "raid4.input" in "rb" mode and we stored it in the array `data`. We printed the dimension of the dataset and we realized that it can be divided into 1117 groups of 4 bytes, leaving out 2 spare elements at the end of it.

In [5]:
```python
#read the input file and store it in "data"
f=open("raid4.input","rb")
data=f.read()
f.close()

#print useful informations about the dataset
print("Data Size: "+str(len(data))+" bytes")
print("Number of groups of 4 bytes:",math.ceil(len(data)/4))
print("Spare elements after grouping by 4:",len(data)%4)
```

```
Data Size: 4470 bytes
Number of groups of 4 bytes: 1118
Spare elements after grouping by 4: 2
```

In order to build and write the four stripe files, we defined the filenames as suggested by the exercise and we opened them in "w" mode. Using a for loop, we selected chunks of 4 bytes from the previously mentioned array `data`, we computed their parity thanks to the user defined function `parity` and we wrote them in the proper files.

Since there were 2 spare elements at the end of the dataset, we inserted an if clause in the "write on file" part of the code in order to add 2 dummy zeros as a tail of our data; in this way we managed to built 4 stripe files, each one containing 1118 elements.

Eventually we inserted the values of the stripe files in a matrix `data_matrix` of dimension (5,1118) just for simplicity. The first 4 columns contain the data as written on the stripe files, while the last column of the matrix is used to store their row-wise parities. Even though this system is not exactly the same as working with files, we thought that it could be still representative of the storage mechanisms and that it could avoid us to read and write the stripe files every time. Moreover, considering the amount of data we are treating, there is no significant difference in the performances of the two methods either.

In [6]:
```python
#file names
stripe0="raid4.0"
stripe1="raid4.1"
stripe2="raid4.2"
stripe3="raid4.3"
stripe4="raid4.4" #parity file

#function to compute parity of a vector
def parity(vec):
    n=len(vec)
    for i in range(1,n):
        if i==1:result=vec[1]^vec[0]
        else:result=result^vec[i]
    return result

#write bytes onto files
with open(stripe0,"w") as f0, open(stripe1,"w") as f1,open(stripe2,"w") as f2,open(stripe3,"w") as f3,o
pen(stripe4,"w") as f4:
    for i in np.arange(0,len(data),4):

        #adding missing zeros at the end of data
        if i==np.arange(0,len(data),4)[-1]:chunk=list([data[i],data[i+1],0,0])
        else: chunk=list(data[i:i+4])

        #compute parity
        par=parity(chunk)

        #write on files
        f0.write(str(chunk[0])+"\n")
        f1.write(str(chunk[1])+"\n")
        f2.write(str(chunk[2])+"\n")
        f3.write(str(chunk[3])+"\n")
        f4.write(str(par)+"\n")


#check dimensions
print("Dimension of each stripe file:",math.ceil(len(data)/4))
print("Total data size:",4*math.ceil(len(data)/4),"(4470 original data + 2 dummy zeros)")
```

```
Dimension of each stripe file: 1118
Total data size: 4472 (4470 original data + 2 dummy zeros)
```

In [7]:
```python
#list of filenames
files=[stripe0,stripe1,stripe2,stripe3,stripe4]

#create matrix with data
data_matrix=np.zeros((5,math.ceil(len(data)/4)),dtype=int)
for i,file in enumerate(files): data_matrix[i,:]=np.loadtxt(file,dtype=int,delimiter="\n")
```

1.2 Extend the program to compute additionally the parity of all bytes within one stripe file.

We computed the column-wise parities of the stripe files considering each column of the `data_matrix` and applying the function `parity` to them.

The computed column-wise parity acts as a checksum for each stripe file. Computing and storing the checksum values is really useful to be able to detect any data loss inside each stripe file. These values are usually computed in the initial state of the storage system and kept stored somewhere. Periodically one can choose to recompute these values in the same way and compare them to the ones computed originally; any detected change in these values is representative of some sort of data loss or corruption inside each stripe file. Of course the checksum information alone is not enough to reconstruct any piece of data; it is impossible to precisely identify the corrupted byte within the stripe file.

In [8]:
```python
#compute column-wise parity for data and parity files
parities=[parity(data_matrix[j,:]) for j in range(5)]

print("Column-wise parities:",parities)
```

```
Column-wise parities: [232, 2, 26, 210, 34]
```

Compute the size overhead by comparing the size of all 5 stripe files with the original file.

We computed the size overhead as the fraction between the actual space used to store the data and their row-wise parities and the space needed to store only the data. In our case, it happens to be the 125%.

This quantity is representative of how much storage space we need to add in order to make a gain in terms of storage reliability: storing the row-wise parity values for our dataset of course gives us the possibility to reconstruct some of our data in case of an unpredicted data loss, nontheless it obliges us to assign some storage space to keep these values saved.

In [9]:
```python
#compute sizes
data_size=4*len(data_matrix[0,:])
total_size=5*len(data_matrix[0,:])

print("Size overhead: "+str(int(total_size*100/data_size))+"%")
```

```
Size overhead: 125%
```

1.3 What is the 5-byte parity value if you write it it in hexadecimal format?

We printed the 5-byte parity value converting the previously computed column-wise parities in hexadecimal format and pasting the results into the format suggested by the exercise:

$$P^5 = 0x[q_0][q_1][q_2][q_3][q_4]$$

where q_i represents the column-wise parity of the i-th stripe file converted in hexadecimal format. Since we were looking for two digits for a byte in hexadecimal format, we added a 0 value where necessary.

In [10]:
```python
hex_par=[]

#compute parities in hexadecimal format
for par in parities:
    val=str(hex(par)[2:])
    if len(val)==1: val="0"+val
    hex_par.append(val)

print("5-Byte Parity Value: 0 x "+hex_par[0]+" "+hex_par[1]+" "+hex_par[2]+" "+hex_par[3]+" "+hex_par[4
])
```

```
5-Byte Parity Value: 0 x e8 02 1a d2 22
```

1.4 If you create a sixth stripe file, which contains the row-wise parities of the five stripe files, what would be the contents of this file? Write down the equation for R.

The row-wise parity of the five stripe files that we constructed before is a file of 1118 identically 0 values. This is of course due to the properties of the XOR operator. It has no meaning to compute the parity of the parity file and the data, since the parity itself already "summarizes" all the informations contained in the data; performing this operation twice is a redundant procedure, which does not provide any additional useful information to our data.

This mechanism can be explained with the following formulae, where we identified with A,B,C and D the data stripe files and with P the parity one.

P=A^B^C^D

R=P^A^B^C^D=(A^B^C^D)^A^B^C^D=(A^A)^(B^B)^(C^C)^(D^D)=0^0^0^0=0

In order to demonstrate it, we computed R with our data and checked that it was an array of 1118 identically 0 values.

In [11]:
```python
#compute R
R=np.zeros(1118)
for i in range(1118):R[i]=parity(data_matrix[:,i])
print("R is identically zero:",np.all(R==0))
```

```
R is identically zero: True
```

1.5 After some time you recompute the 5-byte parity value as in 1.3. Something has been corrupted. Describe how you can recreate the original data file.

We choosed to simulate the corruption of a random byte inside our dataset and its following reconstruction based on the remaining uncorrupted data.

In order to do this, we randomly generated two indices `i` and `j` to identify a specific byte inside our dataset and we corrupted it. The index `i` represents the stripe file in which the byte has been corrupted, therefore it has been choosen randomly as an integer value in the range $[0, 4]$. The `j` value instead represents the row index of the considered byte.

We copied the dataset into another matrix `corrupted_matrix`, we corrupted the byte in the position (i,j) by summing it to a random integer in the range $[0, 255]$ and applying the modulo 255 function afterwards. Eventually we checked that the bytes in the position (i,j) of the corrupted and original data matrices were actually different.

In [15]:
```python
#randomly choose a byte to corrupt
random.seed(1234)
i=random.randint(0,4) #index of striped data files
j=random.randint(0,1118) #column index

#corrupt one byte
corrupted_matrix=data_matrix.copy()
corrupted_matrix[i,j]=(corrupted_matrix[i,j]+random.randint(0,255))%255

print("The byte has been corrupted:",data_matrix[i,j]!=corrupted_matrix[i,j])
```

```
The byte has been corrupted: True
```

We recomputed the column-wise parities of the corrupted data, as one would do in any storage system in order to look for any data corruption. By comparing the newly computed "corrupted parities" with the original parity values, we obtained the proof that something had been actually corrupted.

The checksum values in fact are useful to detect any data loss or corruption inside each stripe file; with the previously mentioned comparison we could infer that the corrupted byte was found in the stripe file "raid4.3", since the non-corresponding parity values were referring to that precise data file. We eventually saved this information in the `derived_i` variable.

We repeated the same comparison even with the parities in hexadecimal format; the result leads of course to the same conclusion.

In [16]:
```python
#compute columnwise parity for all the striped data
parities_corrupted=[parity(corrupted_matrix[j,:]) for j in range(5)]

#compute index i
derived_i=np.arange(0,5,1)[np.asarray(parities_corrupted)!=np.asarray(parities)]

print("Column-wise original parities:",parities)
print("Column-wise corrupted parities:",parities_corrupted)
print("The corrupted byte is in the stripe file: "+str(files[int(derived_i)]))
```

```
Column-wise original parities: [232, 2, 26, 210, 34]
Column-wise corrupted parities: [232, 2, 26, 223, 34]
The corrupted byte is in the stripe file: raid4.3
```

In [17]:
```python
hex_par_corr=[]

#compute parities in hexadecimal format
for i,par in enumerate(parities_corrupted):
    val=str(hex(par)[2:])
    if len(val)==1: val="0"+val
    hex_par_corr.append(val)

print("5-Byte Corrupted Parity Value: 0 x "+hex_par_corr[0]+" "+hex_par_corr[1]+" "+hex_par_corr[2]+" "
+hex_par_corr[3]+" "+hex_par_corr[4])
print("5-Byte Original Parity Value: 0 x "+hex_par[0]+" "+hex_par[1]+" "+hex_par[2]+" "+hex_par[3]+" "+
hex_par[4])
```

```
5-Byte Corrupted Parity Value: 0 x e8 02 1a df 22
5-Byte Original Parity Value: 0 x e8 02 1a d2 22
```

We calculated again the row-wise parity values of the corrupted dataframe and we compared them with the originally stored values. Even in this case, every difference in these values is highlighting the presence of some kind of corruption.

We used this kind of comparison to infer the index value `derived_j`, which is useful to identify in which line of the stripe file "raid4.3" the corrupted byte is. We printed the result and we realized that the corrupted byte was in row 239.

In [20]:
```python
#original parity values
parity_file=data_matrix[4,:]

#re-compute parity with corrupted data
parity_corrupted=np.zeros(1118)
for i in range(1118):parity_corrupted[i]=parity(corrupted_matrix[:4,i])

#compute index j
derived_j=np.arange(0,1118,1)[parity_file!=parity_corrupted]

print("The corrupted byte is in the row:",derived_j)
```

```
The corrupted byte is in the row: [239]
```

After having derived the row and column indices identifying the corrupted byte, we can reconstruct its original value with the remaining uncorrupted informations.

The corrupted byte can be reconstructed using the other data and parity values on its same row. By applying the XOR operator between these values we can reconstruct the original and now missing value for the corrupted byte, since this is the specific property of the XOR operator on which our raid 4 storage reliability is based.

After having reconstructed the byte, we checked that it was the same character as in the original version of the dataframe, in order to verify if our procedure had lead to the correct result.

In [32]:
```python
#corrupted value
value_corrupted=corrupted_matrix[derived_i,derived_j]
print("The corrupted byte is:",chr(value_corrupted))

#reconstructed value
value_reconstructed=parity(np.concatenate((np.reshape(corrupted_matrix[0:3,derived_j],(3,)),corrupted_m
atrix[4,derived_j])))
print("The reconstructed byte is:",chr(value_reconstructed))

#has the reconstrucion lead to the correct result?
print("The reconstructed byte corresponds to the original one:",np.all(data_matrix[derived_i,derived_j]
==value_reconstructed))
```

```
The corrupted byte is: h
The reconstructed byte is: e
The reconstructed byte corresponds to the original one: True
```

Some final comments:

The file based storage system we developed is quite different from and an actual distributed storage system. It allowed us to perform some manipulations in an easier way with respect to what we should have done in a real life system but it gave us the possibility to show the data reconstructing and the distributed file storage mechanisms.

Some comments about what we have done and about the differences with a real life system are mandatory:

- **Corruption vs Data loss:** Our simulation of data corruption is just a simple example to show the data reconstruction mechanism. We have choosed a random byte and we have changed it, but in a real life system it could occur that some arbitrary amount of data gets lost. The procedure to reconstruct any lost data is similar to the one we have shown above. The row and column wise parity values can be used to identify the positions of the lost bytes; these data can be reconstructed afterwards with the same procedure we performed above, just keeping in mind that in the case of a loss of data the dimensions of the stripe files will be inevitably changed. This case could be modeled in the code above by dropping a random byte in the dataframe and reconstructing its original value with the remaining data and parity values.
- **Row-wise an Column-wise parity usage:** In our file based system, we could have choosen to reconstruct the single corrupted byte both in a column-wise and a row-wise fashion. Since the column-wise and row-wise parity values are computed in the same way but on different arrays, they can be both used, together with the remaining uncorrupted data, to reconstruct the wrong byte. Unfortunately this is not always possible in a real life system, since the stripe files are meant to represent different disks on which we store our data. In the case of a disk failure the data can be reconstructed using only the data on the remaining disks.
- **Raid 4:** The data reconstruction can be performed up to some extent in the system we have developed. The raid 4 system gives us the possibility to lose one disk and still be able to reconstruct the original informations, but if we lose more than one disk the data are irrecoverable. In our file based system, we could use also the column-wise parity values, enlarging slightly this data loss limit. In general, having two corrupted bytes along a column or a row in our dataset makes it impossible to reconstruct the original informations with just the column-wise and row-wise respective informations. Even though it is not perfectly reflecting a real life system, with the stripe files as we builted them, we could allow ourselves to lose even two or more bytes on the same row, provided that we still have all their respective column data values. The one-byte data corruption simulation that we reported above is just a simplistic example of data reconstruction: we could even lose more bytes in our stripe files and still be able to reconstruct the whole dataset, provided that the corrupted bytes are an arbitrary small percentage of our dataset and that they are "geometrically" distributed among our files in an optimal way. Again, this methods are hardly applicable in real life systems.

Assignment 2: Cryptography

Assume we receive an email from a friend of us sending the following encrypted message:

K]amua!trgpy

We are told how the encryption algorithm works, meaning that we know we need a key to decrypt the message and that there is a secret variable added for each ASCII character.
It follows that this kind of problem concerns **symmetric encryption**. Indeed:

- there is just one key (symmetric key) used, and it is the same key exploited to encrypt and decrypt the message as we will see with the brute force approach in reconstructing the encryption and decryption procedure
- key lengths are usually 128 or 256 bits, depending on the security criteria, in this case ASCII values range from 0 to 255 and their bits representation is in term of two "nibbles" meaning 8 bits
- even if we have not the key shared from the addresser we can find the correct and only one by a dictionary approach trying all the possibile $[0, 255]$ values

Recall that in symmmetric encryption both sender and receiver know in a deterministic way how the cipher is made, in this case the only missing part is the key that however can be discovered.

Let's try to find the correct key that manages to decrypt the above mentioned unknown text.

By using the unpacking operator in a string and by surrounding everything by square brackets we create a list of single characters that can be translated into their correspondent ASCII values through the `ord()` function

In [1]:
```python
original_mess = [*"K]amua!trgpy"]
original_ascii = [ord(original_mess[i]) for i in range(len(original_mess))]
original_ascii
```

Out[1]:
```
[75, 93, 97, 109, 117, 97, 33, 116, 114, 103, 112, 121]
```

As suggested by the sender, depending on the length of the message, a nonce variable is added to each ASCII value of the message in such a way that the first number is 5 for the firs value, the second is 6 for the second value and so on.

In [2]:
```python
nonce = [i for i in range(5,5 + len(original_ascii))]
```

Without using the numpy library we could generate the resulting list of traslated ASCII numbers by doing another list comprehension. The sender indeed added, to make the message more secure, a nonce variable to each ASCII value of the cypher. Then in order to retrieve the cypher we subtract the nonce list from the original_ascii list

In [3]:
```python
trasl_ascii = [original_ascii[i] - nonce[i] for i in range(len(nonce))]
```

In order to solve the problem we are opting for a dictionary approach. This means we are pairing keys with solutions proposal, trying all of them with `brute force` and choosing the one that leads to a meaningful phrase. First of all let's consider integer keys that range from 0 to 255. The encryption algorithm works by adding the key value to each `ASCII` character of the original message. It follows that we should subtract the same key for the purpose of retrieving the solution.

In [4]:
```python
ascii = [i for i in range(256)]
keym_dictionary = {}

for i in ascii:
    keym_dictionary[i] = [trasl_ascii[j] + i for j in range(len(trasl_ascii))]
```

By printing all the key-message pairs we notice that only one makes sense: as we are going to see it is the one corresponding to $key = \pm 10$ depending on ho we see the encryption algorithm.

In [5]:
```python
for j in ascii:
    print([chr(keym_dictionary[j][i]) for i in range(len(keym_dictionary[j]))], j)
```

```
['F', 'W', 'Z', 'e', 'l', 'W', '\x16', 'h', 'e', 'Y', 'a', 'i'] 0
['G', 'X', '[', 'f', 'm', 'X', '\x17', 'i', 'f', 'Z', 'b', 'j'] 1
['H', 'Y', '\\', 'g', 'n', 'Y', '\x18', 'j', 'g', '[', 'c', 'k'] 2
['I', 'Z', ']', 'h', 'o', 'Z', '\x19', 'k', 'h', '\\', 'd', 'l'] 3
['J', '[', '^', 'i', 'p', '[', '\x1a', 'l', 'i', ']', 'e', 'm'] 4
['K', '\\', '_', 'j', 'q', '\\', '\x1b', 'm', 'j', '^', 'f', 'n'] 5
['L', ']', '`', 'k', 'r', ']', '\x1c', 'n', 'k', '_', 'g', 'o'] 6
['M', '^', 'a', 'l', 's', '^', '\x1d', 'o', 'l', '`', 'h', 'p'] 7
['N', '_', 'b', 'm', 't', '_', '\x1e', 'p', 'm', 'a', 'i', 'q'] 8
['O', '`', 'c', 'n', 'u', '`', '\x1f', 'q', 'n', 'b', 'j', 'r'] 9
['P', 'a', 'd', 'o', 'v', 'a', ' ', 'r', 'o', 'c', 'k', 's'] 10
['Q', 'b', 'e', 'p', 'w', 'b', '!', 's', 'p', 'd', 'l', 't'] 11
['R', 'c', 'f', 'q', 'x', 'c', '"', 't', 'q', 'e', 'm', 'u'] 12
['S', 'd', 'g', 'r', 'y', 'd', '#', 'u', 'r', 'f', 'n', 'v'] 13
['T', 'e', 'h', 's', 'z', 'e', '$', 'v', 's', 'g', 'o', 'w'] 14
['U', 'f', 'i', 't', '{', 'f', '%', 'w', 't', 'h', 'p', 'x'] 15
['V', 'g', 'j', 'u', '|', 'g', '&', 'x', 'u', 'i', 'q', 'y'] 16
['W', 'h', 'k', 'v', '}', 'h', "'", 'y', 'v', 'j', 'r', 'z'] 17
['X', 'i', 'l', 'w', '~', 'i', '(', 'z', 'w', 'k', 's', '{'] 18
['Y', 'j', 'm', 'x', '\x7f', 'j', ')', '{', 'x', 'l', 't', '|'] 19
['Z', 'k', 'n', 'y', '\x80', 'k', '*', '|', 'y', 'm', 'u', '}'] 20
['[', 'l', 'o', 'z', '\x81', 'l', '+', '}', 'z', 'n', 'v', '~'] 21
['\\', 'm', 'p', '{', '\x82', 'm', ',', '~', '{', 'o', 'w', '\x7f'] 22
[']', 'n', 'q', '|', '\x83', 'n', '-', '\x7f', '|', 'p', 'x', '\x80'] 23
['^', 'o', 'r', '}', '\x84', 'o', '.', '\x80', '}', 'q', 'y', '\x81'] 24
['_', 'p', 's', '~', '\x85', 'p', '/', '\x81', '~', 'r', 'z', '\x82'] 25
['`', 'q', 't', '\x7f', '\x86', 'q', '0', '\x82', '\x7f', 's', '{', '\x83'] 26
['a', 'r', 'u', '\x80', '\x87', 'r', '1', '\x83', '\x80', 't', '|', '\x84'] 27
['b', 's', 'v', '\x81', '\x88', 's', '2', '\x84', '\x81', 'u', '}', '\x85'] 28
['c', 't', 'w', '\x82', '\x89', 't', '3', '\x85', '\x82', 'v', '~', '\x86'] 29
['d', 'u', 'x', '\x83', '\x8a', 'u', '4', '\x86', '\x83', 'w', '\x7f', '\x87'] 30
['e', 'v', 'y', '\x84', '\x8b', 'v', '5', '\x87', '\x84', 'x', '\x80', '\x88'] 31
['f', 'w', 'z', '\x85', '\x8c', 'w', '6', '\x88', '\x85', 'y', '\x81', '\x89'] 32
['g', 'x', '{', '\x86', '\x8d', 'x', '7', '\x89', '\x86', 'z', '\x82', '\x8a'] 33
['h', 'y', '|', '\x87', '\x8e', 'y', '8', '\x8a', '\x87', '{', '\x83', '\x8b'] 34
['i', 'z', '}', '\x88', '\x8f', 'z', '9', '\x8b', '\x88', '|', '\x84', '\x8c'] 35
['j', '{', '~', '\x89', '\x90', '{', ':', '\x8c', '\x89', '}', '\x85', '\x8d'] 36
['k', '|', '\x7f', '\x8a', '\x91', '|', ';', '\x8d', '\x8a', '~', '\x86', '\x8e'] 37
['l', '}', '\x80', '\x8b', '\x92', '}', '<', '\x8e', '\x8b', '\x7f', '\x87', '\x8f'] 38
['m', '~', '\x81', '\x8c', '\x93', '~', '=', '\x8f', '\x8c', '\x80', '\x88', '\x90'] 39
['n', '\x7f', '\x82', '\x8d', '\x94', '\x7f', '>', '\x90', '\x8d', '\x81', '\x89', '\x91'] 40
['o', '\x80', '\x83', '\x8e', '\x95', '\x80', '?', '\x91', '\x8e', '\x82', '\x8a', '\x92'] 41
['p', '\x81', '\x84', '\x8f', '\x96', '\x81', '@', '\x92', '\x8f', '\x83', '\x8b', '\x93'] 42
['q', '\x82', '\x85', '\x90', '\x97', '\x82', 'A', '\x93', '\x90', '\x84', '\x8c', '\x94'] 43
['r', '\x83', '\x86', '\x91', '\x98', '\x83', 'B', '\x94', '\x91', '\x85', '\x8d', '\x95'] 44
['s', '\x84', '\x87', '\x92', '\x99', '\x84', 'C', '\x95', '\x92', '\x86', '\x8e', '\x96'] 45
['t', '\x85', '\x88', '\x93', '\x9a', '\x85', 'D', '\x96', '\x93', '\x87', '\x8f', '\x97'] 46
['u', '\x86', '\x89', '\x94', '\x9b', '\x86', 'E', '\x97', '\x94', '\x88', '\x90', '\x98'] 47
['v', '\x87', '\x8a', '\x95', '\x9c', '\x87', 'F', '\x98', '\x95', '\x89', '\x91', '\x99'] 48
['w', '\x88', '\x8b', '\x96', '\x9d', '\x88', 'G', '\x99', '\x96', '\x8a', '\x92', '\x9a'] 49
['x', '\x89', '\x8c', '\x97', '\x9e', '\x89', 'H', '\x9a', '\x97', '\x8b', '\x93', '\x9b'] 50
['y', '\x8a', '\x8d', '\x98', '\x9f', '\x8a', 'I', '\x9b', '\x98', '\x8c', '\x94', '\x9c'] 51
['z', '\x8b', '\x8e', '\x99', '\xa0', '\x8b', 'J', '\x9c', '\x99', '\x8d', '\x95', '\x9d'] 52
['{', '\x8c', '\x8f', '\x9a', '¡', '\x8c', 'K', '\x9d', '\x9a', '\x8e', '\x96', '\x9e'] 53
['|', '\x8d', '\x90', '\x9b', '¢', '\x8d', 'L', '\x9e', '\x9b', '\x8f', '\x97', '\x9f'] 54
['}', '\x8e', '\x91', '\x9c', '£', '\x8e', 'M', '\x9f', '\x9c', '\x90', '\x98', '\xa0'] 55
['~', '\x8f', '\x92', '\x9d', '¤', '\x8f', 'N', '\xa0', '\x9d', '\x91', '\x99', '¡'] 56
['\x7f', '\x90', '\x93', '\x9e', '¥', '\x90', 'O', '¡', '\x9e', '\x92', '\x9a', '¢'] 57
['\x80', '\x91', '\x94', '\x9f', '¦', '\x91', 'P', '¢', '\x9f', '\x93', '\x9b', '£'] 58
['\x81', '\x92', '\x95', '\xa0', '§', '\x92', 'Q', '£', '\xa0', '\x94', '\x9c', '¤'] 59
['\x82', '\x93', '\x96', '¡', '¨', '\x93', 'R', '¤', '¡', '\x95', '\x9d', '¥'] 60
['\x83', '\x94', '\x97', '¢', '©', '\x94', 'S', '¥', '¢', '\x96', '\x9e', '¦'] 61
['\x84', '\x95', '\x98', '£', 'ª', '\x95', 'T', '¦', '£', '\x97', '\x9f', '§'] 62
['\x85', '\x96', '\x99', '¤', '«', '\x96', 'U', '§', '¤', '\x98', '\xa0', '¨'] 63
['\x86', '\x97', '\x9a', '¥', '¬', '\x97', 'V', '¨', '¥', '\x99', '¡', '©'] 64
['\x87', '\x98', '\x9b', '¦', '\xad', '\x98', 'W', '©', '¦', '\x9a', '¢', 'ª'] 65
['\x88', '\x99', '\x9c', '§', '®', '\x99', 'X', 'ª', '§', '\x9b', '£', '«'] 66
['\x89', '\x9a', '\x9d', '¨', '¯', '\x9a', 'Y', '«', '¨', '\x9c', '¤', '¬'] 67
['\x8a', '\x9b', '\x9e', '©', '°', '\x9b', 'Z', '¬', '©', '\x9d', '¥', '\xad'] 68
['\x8b', '\x9c', '\x9f', 'ª', '±', '\x9c', '[', '\xad', 'ª', '\x9e', '¦', '®'] 69
['\x8c', '\x9d', '\xa0', '«', '²', '\x9d', '\\', '®', '«', '\x9f', '§', '¯'] 70
['\x8d', '\x9e', '¡', '¬', '³', '\x9e', ']', '¯', '¬', '\xa0', '¨', '°'] 71
['\x8e', '\x9f', '¢', '\xad', '´', '\x9f', '^', '°', '\xad', '¡', '©', '±'] 72
['\x8f', '\xa0', '£', '®', 'µ', '\xa0', '_', '±', '®', '¢', 'ª', '²'] 73
['\x90', '¡', '¤', '¯', '¶', '¡', '`', '²', '¯', '£', '«', '³'] 74
['\x91', '¢', '¥', '°', '·', '¢', 'a', '³', '°', '¤', '¬', '´'] 75
['\x92', '£', '¦', '±', '¸', '£', 'b', '´', '±', '¥', '\xad', 'µ'] 76
['\x93', '¤', '§', '²', '¹', '¤', 'c', 'µ', '²', '¦', '®', '¶'] 77
['\x94', '¥', '¨', '³', 'º', '¥', 'd', '¶', '³', '§', '¯', '·'] 78
['\x95', '¦', '©', '´', '»', '¦', 'e', '·', '´', '¨', '°', '¸'] 79
['\x96', '§', 'ª', 'µ', '¼', '§', 'f', '¸', 'µ', '©', '±', '¹'] 80
['\x97', '¨', '«', '¶', '½', '¨', 'g', '¹', '¶', 'ª', '²', 'º'] 81
['\x98', '©', '¬', '·', '¾', '©', 'h', 'º', '·', '«', '³', '»'] 82
['\x99', 'ª', '\xad', '¸', '¿', 'ª', 'i', '»', '¸', '¬', '´', '¼'] 83
['\x9a', '«', '®', '¹', 'À', '«', 'j', '¼', '¹', '\xad', 'µ', '½'] 84
['\x9b', '¬', '¯', 'º', 'Á', '¬', 'k', '½', 'º', '®', '¶', '¾'] 85
['\x9c', '\xad', '°', '»', 'Â', '\xad', 'l', '¾', '»', '¯', '·', '¿'] 86
['\x9d', '®', '±', '¼', 'Ã', '®', 'm', '¿', '¼', '°', '¸', 'À'] 87
['\x9e', '¯', '²', '½', 'Ä', '¯', 'n', 'À', '½', '±', '¹', 'Á'] 88
['\x9f', '°', '³', '¾', 'Å', '°', 'o', 'Á', '¾', '²', 'º', 'Â'] 89
['\xa0', '±', '´', '¿', 'Æ', '±', 'p', 'Â', '¿', '³', '»', 'Ã'] 90
['¡', '²', 'µ', 'À', 'Ç', '²', 'q', 'Ã', 'À', '´', '¼', 'Ä'] 91
['¢', '³', '¶', 'Á', 'È', '³', 'r', 'Ä', 'Á', 'µ', '½', 'Å'] 92
['£', '´', '·', 'Â', 'É', '´', 's', 'Å', 'Â', '¶', '¾', 'Æ'] 93
['¤', 'µ', '¸', 'Ã', 'Ê', 'µ', 't', 'Æ', 'Ã', '·', '¿', 'Ç'] 94
['¥', '¶', '¹', 'Ä', 'Ë', '¶', 'u', 'Ç', 'Ä', '¸', 'À', 'È'] 95
['¦', '·', 'º', 'Å', 'Ì', '·', 'v', 'È', 'Å', '¹', 'Á', 'É'] 96
['§', '¸', '»', 'Æ', 'Í', '¸', 'w', 'É', 'Æ', 'º', 'Â', 'Ê'] 97
['¨', '¹', '¼', 'Ç', 'Î', '¹', 'x', 'Ê', 'Ç', '»', 'Ã', 'Ë'] 98
['©', 'º', '½', 'È', 'Ï', 'º', 'y', 'Ë', 'È', '¼', 'Ä', 'Ì'] 99
['ª', '»', '¾', 'É', 'Ð', '»', 'z', 'Ì', 'É', '½', 'Å', 'Í'] 100
['«', '¼', '¿', 'Ê', 'Ñ', '¼', '{', 'Í', 'Ê', '¾', 'Æ', 'Î'] 101
['¬', '½', 'À', 'Ë', 'Ò', '½', '|', 'Î', 'Ë', '¿', 'Ç', 'Ï'] 102
['\xad', '¾', 'Á', 'Ì', 'Ó', '¾', '}', 'Ï', 'Ì', 'À', 'È', 'Ð'] 103
['®', '¿', 'Â', 'Í', 'Ô', '¿', '~', 'Ð', 'Í', 'Á', 'É', 'Ñ'] 104
['¯', 'À', 'Ã', 'Î', 'Õ', 'À', '\x7f', 'Ñ', 'Î', 'Â', 'Ê', 'Ò'] 105
['°', 'Á', 'Ä', 'Ï', 'Ö', 'Á', '\x80', 'Ò', 'Ï', 'Ã', 'Ë', 'Ó'] 106
['±', 'Â', 'Å', 'Ð', '×', 'Â', '\x81', 'Ó', 'Ð', 'Ä', 'Ì', 'Ô'] 107
['²', 'Ã', 'Æ', 'Ñ', 'Ø', 'Ã', '\x82', 'Ô', 'Ñ', 'Å', 'Í', 'Õ'] 108
['³', 'Ä', 'Ç', 'Ò', 'Ù', 'Ä', '\x83', 'Õ', 'Ò', 'Æ', 'Î', 'Ö'] 109
['´', 'Å', 'È', 'Ó', 'Ú', 'Å', '\x84', 'Ö', 'Ó', 'Ç', 'Ï', '×'] 110
['µ', 'Æ', 'É', 'Ô', 'Û', 'Æ', '\x85', '×', 'Ô', 'È', 'Ð', 'Ø'] 111
['¶', 'Ç', 'Ê', 'Õ', 'Ü', 'Ç', '\x86', 'Ø', 'Õ', 'É', 'Ñ', 'Ù'] 112
['·', 'È', 'Ë', 'Ö', 'Ý', 'È', '\x87', 'Ù', 'Ö', 'Ê', 'Ò', 'Ú'] 113
['¸', 'É', 'Ì', '×', 'Þ', 'É', '\x88', 'Ú', '×', 'Ë', 'Ó', 'Û'] 114
['¹', 'Ê', 'Í', 'Ø', 'ß', 'Ê', '\x89', 'Û', 'Ø', 'Ì', 'Ô', 'Ü'] 115
['º', 'Ë', 'Î', 'Ù', 'à', 'Ë', '\x8a', 'Ü', 'Ù', 'Í', 'Õ', 'Ý'] 116
['»', 'Ì', 'Ï', 'Ú', 'á', 'Ì', '\x8b', 'Ý', 'Ú', 'Î', 'Ö', 'Þ'] 117
['¼', 'Í', 'Ð', 'Û', 'â', 'Í', '\x8c', 'Þ', 'Û', 'Ï', '×', 'ß'] 118
['½', 'Î', 'Ñ', 'Ü', 'ã', 'Î', '\x8d', 'ß', 'Ü', 'Ð', 'Ø', 'à'] 119
['¾', 'Ï', 'Ò', 'Ý', 'ä', 'Ï', '\x8e', 'à', 'Ý', 'Ñ', 'Ù', 'á'] 120
['¿', 'Ð', 'Ó', 'Þ', 'å', 'Ð', '\x8f', 'á', 'Þ', 'Ò', 'Ú', 'â'] 121
['À', 'Ñ', 'Ô', 'ß', 'æ', 'Ñ', '\x90', 'â', 'ß', 'Ó', 'Û', 'ã'] 122
['Á', 'Ò', 'Õ', 'à', 'ç', 'Ò', '\x91', 'ã', 'à', 'Ô', 'Ü', 'ä'] 123
['Â', 'Ó', 'Ö', 'á', 'è', 'Ó', '\x92', 'ä', 'á', 'Õ', 'Ý', 'å'] 124
['Ã', 'Ô', '×', 'â', 'é', 'Ô', '\x93', 'å', 'â', 'Ö', 'Þ', 'æ'] 125
['Ä', 'Õ', 'Ø', 'ã', 'ê', 'Õ', '\x94', 'æ', 'ã', '×', 'ß', 'ç'] 126
['Å', 'Ö', 'Ù', 'ä', 'ë', 'Ö', '\x95', 'ç', 'ä', 'Ø', 'à', 'è'] 127
['Æ', '×', 'Ú', 'å', 'ì', '×', '\x96', 'è', 'å', 'Ù', 'á', 'é'] 128
['Ç', 'Ø', 'Û', 'æ', 'í', 'Ø', '\x97', 'é', 'æ', 'Ú', 'â', 'ê'] 129
['È', 'Ù', 'Ü', 'ç', 'î', 'Ù', '\x98', 'ê', 'ç', 'Û', 'ã', 'ë'] 130
['É', 'Ú', 'Ý', 'è', 'ï', 'Ú', '\x99', 'ë', 'è', 'Ü', 'ä', 'ì'] 131
['Ê', 'Û', 'Þ', 'é', 'ð', 'Û', '\x9a', 'ì', 'é', 'Ý', 'å', 'í'] 132
['Ë', 'Ü', 'ß', 'ê', 'ñ', 'Ü', '\x9b', 'í', 'ê', 'Þ', 'æ', 'î'] 133
['Ì', 'Ý', 'à', 'ë', 'ò', 'Ý', '\x9c', 'î', 'ë', 'ß', 'ç', 'ï'] 134
['Í', 'Þ', 'á', 'ì', 'ó', 'Þ', '\x9d', 'ï', 'ì', 'à', 'è', 'ð'] 135
['Î', 'ß', 'â', 'í', 'ô', 'ß', '\x9e', 'ð', 'í', 'á', 'é', 'ñ'] 136
['Ï', 'à', 'ã', 'î', 'õ', 'à', '\x9f', 'ñ', 'î', 'â', 'ê', 'ò'] 137
['Ð', 'á', 'ä', 'ï', 'ö', 'á', '\xa0', 'ò', 'ï', 'ã', 'ë', 'ó'] 138
['Ñ', 'â', 'å', 'ð', '÷', 'â', '¡', 'ó', 'ð', 'ä', 'ì', 'ô'] 139
['Ò', 'ã', 'æ', 'ñ', 'ø', 'ã', '¢', 'ô', 'ñ', 'å', 'í', 'õ'] 140
['Ó', 'ä', 'ç', 'ò', 'ù', 'ä', '£', 'õ', 'ò', 'æ', 'î', 'ö'] 141
['Ô', 'å', 'è', 'ó', 'ú', 'å', '¤', 'ö', 'ó', 'ç', 'ï', '÷'] 142
['Õ', 'æ', 'é', 'ô', 'û', 'æ', '¥', '÷', 'ô', 'è', 'ð', 'ø'] 143
['Ö', 'ç', 'ê', 'õ', 'ü', 'ç', '¦', 'ø', 'õ', 'é', 'ñ', 'ù'] 144
['×', 'è', 'ë', 'ö', 'ý', 'è', '§', 'ù', 'ö', 'ê', 'ò', 'ú'] 145
['Ø', 'é', 'ì', '÷', 'þ', 'é', '¨', 'ú', '÷', 'ë', 'ó', 'û'] 146
['Ù', 'ê', 'í', 'ø', 'ÿ', 'ê', '©', 'û', 'ø', 'ì', 'ô', 'ü'] 147
['Ú', 'ë', 'î', 'ù', 'Ā', 'ë', 'ª', 'ü', 'ù', 'í', 'õ', 'ý'] 148
['Û', 'ì', 'ï', 'ú', 'ā', 'ì', '«', 'ý', 'ú', 'î', 'ö', 'þ'] 149
['Ü', 'í', 'ð', 'û', 'Ă', 'í', '¬', 'þ', 'û', 'ï', '÷', 'ÿ'] 150
['Ý', 'î', 'ñ', 'ü', 'ă', 'î', '\xad', 'ÿ', 'ü', 'ð', 'ø', 'Ā'] 151
['Þ', 'ï', 'ò', 'ý', 'Ą', 'ï', '®', 'Ā', 'ý', 'ñ', 'ù', 'ā'] 152
['ß', 'ð', 'ó', 'þ', 'ą', 'ð', '¯', 'ā', 'þ', 'ò', 'ú', 'Ă'] 153
['à', 'ñ', 'ô', 'ÿ', 'Ć', 'ñ', '°', 'Ă', 'ÿ', 'ó', 'û', 'ă'] 154
['á', 'ò', 'õ', 'Ā', 'ć', 'ò', '±', 'ă', 'Ā', 'ô', 'ü', 'Ą'] 155
['â', 'ó', 'ö', 'ā', 'Ĉ', 'ó', '²', 'Ą', 'ā', 'õ', 'ý', 'ą'] 156
['ã', 'ô', '÷', 'Ă', 'ĉ', 'ô', '³', 'ą', 'Ă', 'ö', 'þ', 'Ć'] 157
['ä', 'õ', 'ø', 'ă', 'Ċ', 'õ', '´', 'Ć', 'ă', '÷', 'ÿ', 'ć'] 158
['å', 'ö', 'ù', 'Ą', 'ċ', 'ö', 'µ', 'ć', 'Ą', 'ø', 'Ā', 'Ĉ'] 159
['æ', '÷', 'ú', 'ą', 'Č', '÷', '¶', 'Ĉ', 'ą', 'ù', 'ā', 'ĉ'] 160
['ç', 'ø', 'û', 'Ć', 'č', 'ø', '·', 'ĉ', 'Ć', 'ú', 'Ă', 'Ċ'] 161
['è', 'ù', 'ü', 'ć', 'Ď', 'ù', '¸', 'Ċ', 'ć', 'û', 'ă', 'ċ'] 162
['é', 'ú', 'ý', 'Ĉ', 'ď', 'ú', '¹', 'ċ', 'Ĉ', 'ü', 'Ą', 'Č'] 163
['ê', 'û', 'þ', 'ĉ', 'Đ', 'û', 'º', 'Č', 'ĉ', 'ý', 'ą', 'č'] 164
['ë', 'ü', 'ÿ', 'Ċ', 'đ', 'ü', '»', 'č', 'Ċ', 'þ', 'Ć', 'Ď'] 165
['ì', 'ý', 'Ā', 'ċ', 'Ē', 'ý', '¼', 'Ď', 'ċ', 'ÿ', 'ć', 'ď'] 166
['í', 'þ', 'ā', 'Č', 'ē', 'þ', '½', 'ď', 'Č', 'Ā', 'Ĉ', 'Đ'] 167
['î', 'ÿ', 'Ă', 'č', 'Ĕ', 'ÿ', '¾', 'Đ', 'č', 'ā', 'ĉ', 'đ'] 168
['ï', 'Ā', 'ă', 'Ď', 'ĕ', 'Ā', '¿', 'đ', 'Ď', 'Ă', 'Ċ', 'Ē'] 169
['ð', 'ā', 'Ą', 'ď', 'Ė', 'ā', 'À', 'Ē', 'ď', 'ă', 'ċ', 'ē'] 170
['ñ', 'Ă', 'ą', 'Đ', 'ė', 'Ă', 'Á', 'ē', 'Đ', 'Ą', 'Č', 'Ĕ'] 171
['ò', 'ă', 'Ć', 'đ', 'Ę', 'ă', 'Â', 'Ĕ', 'đ', 'ą', 'č', 'ĕ'] 172
['ó', 'Ą', 'ć', 'Ē', 'ę', 'Ą', 'Ã', 'ĕ', 'Ē', 'Ć', 'Ď', 'Ė'] 173
['ô', 'ą', 'Ĉ', 'ē', 'Ě', 'ą', 'Ä', 'Ė', 'ē', 'ć', 'ď', 'ė'] 174
['õ', 'Ć', 'ĉ', 'Ĕ', 'ě', 'Ć', 'Å', 'ė', 'Ĕ', 'Ĉ', 'Đ', 'Ę'] 175
['ö', 'ć', 'Ċ', 'ĕ', 'Ĝ', 'ć', 'Æ', 'Ę', 'ĕ', 'ĉ', 'đ', 'ę'] 176
['÷', 'Ĉ', 'ċ', 'Ė', 'ĝ', 'Ĉ', 'Ç', 'ę', 'Ė', 'Ċ', 'Ē', 'Ě'] 177
['ø', 'ĉ', 'Č', 'ė', 'Ğ', 'ĉ', 'È', 'Ě', 'ė', 'ċ', 'ē', 'ě'] 178
['ù', 'Ċ', 'č', 'Ę', 'ğ', 'Ċ', 'É', 'ě', 'Ę', 'Č', 'Ĕ', 'Ĝ'] 179
['ú', 'ċ', 'Ď', 'ę', 'Ġ', 'ċ', 'Ê', 'Ĝ', 'ę', 'č', 'ĕ', 'ĝ'] 180
['û', 'Č', 'ď', 'Ě', 'ġ', 'Č', 'Ë', 'ĝ', 'Ě', 'Ď', 'Ė', 'Ğ'] 181
['ü', 'č', 'Đ', 'ě', 'Ģ', 'č', 'Ì', 'Ğ', 'ě', 'ď', 'ė', 'ğ'] 182
['ý', 'Ď', 'đ', 'Ĝ', 'ģ', 'Ď', 'Í', 'ğ', 'Ĝ', 'Đ', 'Ę', 'Ġ'] 183
['þ', 'ď', 'Ē', 'ĝ', 'Ĥ', 'ď', 'Î', 'Ġ', 'ĝ', 'đ', 'ę', 'ġ'] 184
['ÿ', 'Đ', 'ē', 'Ğ', 'ĥ', 'Đ', 'Ï', 'ġ', 'Ğ', 'Ē', 'Ě', 'Ģ'] 185
['Ā', 'đ', 'Ĕ', 'ğ', 'Ħ', 'đ', 'Ð', 'Ģ', 'ğ', 'ē', 'ě', 'ģ'] 186
['ā', 'Ē', 'ĕ', 'Ġ', 'ħ', 'Ē', 'Ñ', 'ģ', 'Ġ', 'Ĕ', 'Ĝ', 'Ĥ'] 187
['Ă', 'ē', 'Ė', 'ġ', 'Ĩ', 'ē', 'Ò', 'Ĥ', 'ġ', 'ĕ', 'ĝ', 'ĥ'] 188
['ă', 'Ĕ', 'ė', 'Ģ', 'ĩ', 'Ĕ', 'Ó', 'ĥ', 'Ģ', 'Ė', 'Ğ', 'Ħ'] 189
['Ą', 'ĕ', 'Ę', 'ģ', 'Ī', 'ĕ', 'Ô', 'Ħ', 'ģ', 'ė', 'ğ', 'ħ'] 190
['ą', 'Ė', 'ę', 'Ĥ', 'ī', 'Ė', 'Õ', 'ħ', 'Ĥ', 'Ę', 'Ġ', 'Ĩ'] 191
['Ć', 'ė', 'Ě', 'ĥ', 'Ĭ', 'ė', 'Ö', 'Ĩ', 'ĥ', 'ę', 'ġ', 'ĩ'] 192
['ć', 'Ę', 'ě', 'Ħ', 'ĭ', 'Ę', '×', 'ĩ', 'Ħ', 'Ě', 'Ģ', 'Ī'] 193
['Ĉ', 'ę', 'Ĝ', 'ħ', 'Į', 'ę', 'Ø', 'Ī', 'ħ', 'ě', 'ģ', 'ī'] 194
['ĉ', 'Ě', 'ĝ', 'Ĩ', 'į', 'Ě', 'Ù', 'ī', 'Ĩ', 'Ĝ', 'Ĥ', 'Ĭ'] 195
['Ċ', 'ě', 'Ğ', 'ĩ', 'İ', 'ě', 'Ú', 'Ĭ', 'ĩ', 'ĝ', 'ĥ', 'ĭ'] 196
['ċ', 'Ĝ', 'ğ', 'Ī', 'ı', 'Ĝ', 'Û', 'ĭ', 'Ī', 'Ğ', 'Ħ', 'Į'] 197
['Č', 'ĝ', 'Ġ', 'ī', 'Ĳ', 'ĝ', 'Ü', 'Į', 'ī', 'ğ', 'ħ', 'į'] 198
['č', 'Ğ', 'ġ', 'Ĭ', 'ĳ', 'Ğ', 'Ý', 'į', 'Ĭ', 'Ġ', 'Ĩ', 'İ'] 199
['Ď', 'ğ', 'Ģ', 'ĭ', 'Ĵ', 'ğ', 'Þ', 'İ', 'ĭ', 'ġ', 'ĩ', 'ı'] 200
['ď', 'Ġ', 'ģ', 'Į', 'ĵ', 'Ġ', 'ß', 'ı', 'Į', 'Ģ', 'Ī', 'Ĳ'] 201
['Đ', 'ġ', 'Ĥ', 'į', 'Ķ', 'ġ', 'à', 'Ĳ', 'į', 'ģ', 'ī', 'ĳ'] 202
['đ', 'Ģ', 'ĥ', 'İ', 'ķ', 'Ģ', 'á', 'ĳ', 'İ', 'Ĥ', 'Ĭ', 'Ĵ'] 203
['Ē', 'ģ', 'Ħ', 'ı', 'ĸ', 'ģ', 'â', 'Ĵ', 'ı', 'ĥ', 'ĭ', 'ĵ'] 204
['ē', 'Ĥ', 'ħ', 'Ĳ', 'Ĺ', 'Ĥ', 'ã', 'ĵ', 'Ĳ', 'Ħ', 'Į', 'Ķ'] 205
['Ĕ', 'ĥ', 'Ĩ', 'ĳ', 'ĺ', 'ĥ', 'ä', 'Ķ', 'ĳ', 'ħ', 'į', 'ķ'] 206
['ĕ', 'Ħ', 'ĩ', 'Ĵ', 'Ļ', 'Ħ', 'å', 'ķ', 'Ĵ', 'Ĩ', 'İ', 'ĸ'] 207
['Ė', 'ħ', 'Ī', 'ĵ', 'ļ', 'ħ', 'æ', 'ĸ', 'ĵ', 'ĩ', 'ı', 'Ĺ'] 208
['ė', 'Ĩ', 'ī', 'Ķ', 'Ľ', 'Ĩ', 'ç', 'Ĺ', 'Ķ', 'Ī', 'Ĳ', 'ĺ'] 209
['Ę', 'ĩ', 'Ĭ', 'ķ', 'ľ', 'ĩ', 'è', 'ĺ', 'ķ', 'ī', 'ĳ', 'Ļ'] 210
['ę', 'Ī', 'ĭ', 'ĸ', 'Ŀ', 'Ī', 'é', 'Ļ', 'ĸ', 'Ĭ', 'Ĵ', 'ļ'] 211
['Ě', 'ī', 'Į', 'Ĺ', 'ŀ', 'ī', 'ê', 'ļ', 'Ĺ', 'ĭ', 'ĵ', 'Ľ'] 212
['ě', 'Ĭ', 'į', 'ĺ', 'Ł', 'Ĭ', 'ë', 'Ľ', 'ĺ', 'Į', 'Ķ', 'ľ'] 213
['Ĝ', 'ĭ', 'İ', 'Ļ', 'ł', 'ĭ', 'ì', 'ľ', 'Ļ', 'į', 'ķ', 'Ŀ'] 214
['ĝ', 'Į', 'ı', 'ļ', 'Ń', 'Į', 'í', 'Ŀ', 'ļ', 'İ', 'ĸ', 'ŀ'] 215
['Ğ', 'į', 'Ĳ', 'Ľ', 'ń', 'į', 'î', 'ŀ', 'Ľ', 'ı', 'Ĺ', 'Ł'] 216
['ğ', 'İ', 'ĳ', 'ľ', 'Ņ', 'İ', 'ï', 'Ł', 'ľ', 'Ĳ', 'ĺ', 'ł'] 217
['Ġ', 'ı', 'Ĵ', 'Ŀ', 'ņ', 'ı', 'ð', 'ł', 'Ŀ', 'ĳ', 'Ļ', 'Ń'] 218
['ġ', 'Ĳ', 'ĵ', 'ŀ', 'Ň', 'Ĳ', 'ñ', 'Ń', 'ŀ', 'Ĵ', 'ļ', 'ń'] 219
['Ģ', 'ĳ', 'Ķ', 'Ł', 'ň', 'ĳ', 'ò', 'ń', 'Ł', 'ĵ', 'Ľ', 'Ņ'] 220
['ģ', 'Ĵ', 'ķ', 'ł', 'ŉ', 'Ĵ', 'ó', 'Ņ', 'ł', 'Ķ', 'ľ', 'ņ'] 221
['Ĥ', 'ĵ', 'ĸ', 'Ń', 'Ŋ', 'ĵ', 'ô', 'ņ', 'Ń', 'ķ', 'Ŀ', 'Ň'] 222
['ĥ', 'Ķ', 'Ĺ', 'ń', 'ŋ', 'Ķ', 'õ', 'Ň', 'ń', 'ĸ', 'ŀ', 'ň'] 223
['Ħ', 'ķ', 'ĺ', 'Ņ', 'Ō', 'ķ', 'ö', 'ň', 'Ņ', 'Ĺ', 'Ł', 'ŉ'] 224
['ħ', 'ĸ', 'Ļ', 'ņ', 'ō', 'ĸ', '÷', 'ŉ', 'ņ', 'ĺ', 'ł', 'Ŋ'] 225
['Ĩ', 'Ĺ', 'ļ', 'Ň', 'Ŏ', 'Ĺ', 'ø', 'Ŋ', 'Ň', 'Ļ', 'Ń', 'ŋ'] 226
['ĩ', 'ĺ', 'Ľ', 'ň', 'ŏ', 'ĺ', 'ù', 'ŋ', 'ň', 'ļ', 'ń', 'Ō'] 227
['Ī', 'Ļ', 'ľ', 'ŉ', 'Ő', 'Ļ', 'ú', 'Ō', 'ŉ', 'Ľ', 'Ņ', 'ō'] 228
['ī', 'ļ', 'Ŀ', 'Ŋ', 'ő', 'ļ', 'û', 'ō', 'Ŋ', 'ľ', 'ņ', 'Ŏ'] 229
['Ĭ', 'Ľ', 'ŀ', 'ŋ', 'Œ', 'Ľ', 'ü', 'Ŏ', 'ŋ', 'Ŀ', 'Ň', 'ŏ'] 230
['ĭ', 'ľ', 'Ł', 'Ō', 'œ', 'ľ', 'ý', 'ŏ', 'Ō', 'ŀ', 'ň', 'Ő'] 231
['Į', 'Ŀ', 'ł', 'ō', 'Ŕ', 'Ŀ', 'þ', 'Ő', 'ō', 'Ł', 'ŉ', 'ő'] 232
['į', 'ŀ', 'Ń', 'Ŏ', 'ŕ', 'ŀ', 'ÿ', 'ő', 'Ŏ', 'ł', 'Ŋ', 'Œ'] 233
['İ', 'Ł', 'ń', 'ŏ', 'Ŗ', 'Ł', 'Ā', 'Œ', 'ŏ', 'Ń', 'ŋ', 'œ'] 234
['ı', 'ł', 'Ņ', 'Ő', 'ŗ', 'ł', 'ā', 'œ', 'Ő', 'ń', 'Ō', 'Ŕ'] 235
['Ĳ', 'Ń', 'ņ', 'ő', 'Ř', 'Ń', 'Ă', 'Ŕ', 'ő', 'Ņ', 'ō', 'ŕ'] 236
['ĳ', 'ń', 'Ň', 'Œ', 'ř', 'ń', 'ă', 'ŕ', 'Œ', 'ņ', 'Ŏ', 'Ŗ'] 237
['Ĵ', 'Ņ', 'ň', 'œ', 'Ś', 'Ņ', 'Ą', 'Ŗ', 'œ', 'Ň', 'ŏ', 'ŗ'] 238
['ĵ', 'ņ', 'ŉ', 'Ŕ', 'ś', 'ņ', 'ą', 'ŗ', 'Ŕ', 'ň', 'Ő', 'Ř'] 239
['Ķ', 'Ň', 'Ŋ', 'ŕ', 'Ŝ', 'Ň', 'Ć', 'Ř', 'ŕ', 'ŉ', 'ő', 'ř'] 240
['ķ', 'ň', 'ŋ', 'Ŗ', 'ŝ', 'ň', 'ć', 'ř', 'Ŗ', 'Ŋ', 'Œ', 'Ś'] 241
['ĸ', 'ŉ', 'Ō', 'ŗ', 'Ş', 'ŉ', 'Ĉ', 'Ś', 'ŗ', 'ŋ', 'œ', 'ś'] 242
['Ĺ', 'Ŋ', 'ō', 'Ř', 'ş', 'Ŋ', 'ĉ', 'ś', 'Ř', 'Ō', 'Ŕ', 'Ŝ'] 243
['ĺ', 'ŋ', 'Ŏ', 'ř', 'Š', 'ŋ', 'Ċ', 'Ŝ', 'ř', 'ō', 'ŕ', 'ŝ'] 244
['Ļ', 'Ō', 'ŏ', 'Ś', 'š', 'Ō', 'ċ', 'ŝ', 'Ś', 'Ŏ', 'Ŗ', 'Ş'] 245
['ļ', 'ō', 'Ő', 'ś', 'Ţ', 'ō', 'Č', 'Ş', 'ś', 'ŏ', 'ŗ', 'ş'] 246
['Ľ', 'Ŏ', 'ő', 'Ŝ', 'ţ', 'Ŏ', 'č', 'ş', 'Ŝ', 'Ő', 'Ř', 'Š'] 247
['ľ', 'ŏ', 'Œ', 'ŝ', 'Ť', 'ŏ', 'Ď', 'Š', 'ŝ', 'ő', 'ř', 'š'] 248
['Ŀ', 'Ő', 'œ', 'Ş', 'ť', 'Ő', 'ď', 'š', 'Ş', 'Œ', 'Ś', 'Ţ'] 249
['ŀ', 'ő', 'Ŕ', 'ş', 'Ŧ', 'ő', 'Đ', 'Ţ', 'ş', 'œ', 'ś', 'ţ'] 250
['Ł', 'Œ', 'ŕ', 'Š', 'ŧ', 'Œ', 'đ', 'ţ', 'Š', 'Ŕ', 'Ŝ', 'Ť'] 251
['ł', 'œ', 'Ŗ', 'š', 'Ũ', 'œ', 'Ē', 'Ť', 'š', 'ŕ', 'ŝ', 'ť'] 252
['Ń', 'Ŕ', 'ŗ', 'Ţ', 'ũ', 'Ŕ', 'ē', 'ť', 'Ţ', 'Ŗ', 'Ş', 'Ŧ'] 253
['ń', 'ŕ', 'Ř', 'ţ', 'Ū', 'ŕ', 'Ĕ', 'Ŧ', 'ţ', 'ŗ', 'ş', 'ŧ'] 254
['Ņ', 'Ŗ', 'ř', 'Ť', 'ū', 'Ŗ', 'ĕ', 'ŧ', 'Ť', 'Ř', 'Š', 'Ũ'] 255
```

The original message was therefore:

In [6]:
```python
message = [chr(keym_dictionary[10][i]) for i in range(len(keym_dictionary[10]))]
"".join(message)
```

Out[6]:
```
'Padova rocks'
```

Encryption procedure

We could try, in order to show one of the property of symmetric encryption, that given the key and the original message we can rebuild the cyphred one by reversing the decryption procedure:
what changes are the signs of the key and the `nonce` variable!

In [7]:
```python
backwords = [ord(message[i]) for i in range(len(message))]
backwords = [backwords[i] - 10 for i in range(len(backwords))]
nonce_back = [i for i in range(5,5 + len(backwords))]
backwords = [backwords[i] + nonce_back[i] for i in range(len(backwords))]
init = [chr(backwords[i]) for i in range(len(backwords))]
"".join(init)
```

Out[7]:
```
'K]amua!trgpy'
```

Assignment 3: Object Storage Technology

In [1]:
```python
import numpy as np
import matplotlib.pyplot as plt
from numba import njit
import seaborn as sns
import pandas as pd
from IPython.display import display
import os
```

In [2]:
```python
np.random.seed(12345)
```

3.1 & 3.2

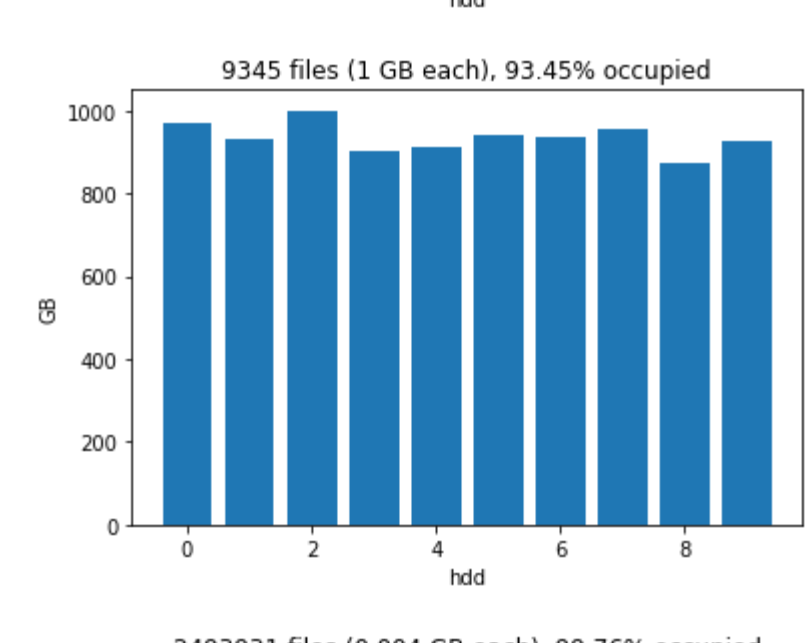
First of all let's start by simulating the process of filling our hard drives. As said in the exercise text in a realistic setting we would use a hash function, but here it suffices to assign files randomly with uniform probability to any hard drive; this is true because an "asymmetric" hash (i.e. one that preferentially fills e.g. HD 0) would never be actually utilized, hence asymptotically the uniform distribution will behave similarly to a realistic hash.
Let's say we're working with 10 1TB HDDs (as said in the exercise text) and with files of `filesize` GB; then our "Montecarlo" algorithm to simulate the storage process works as follows:

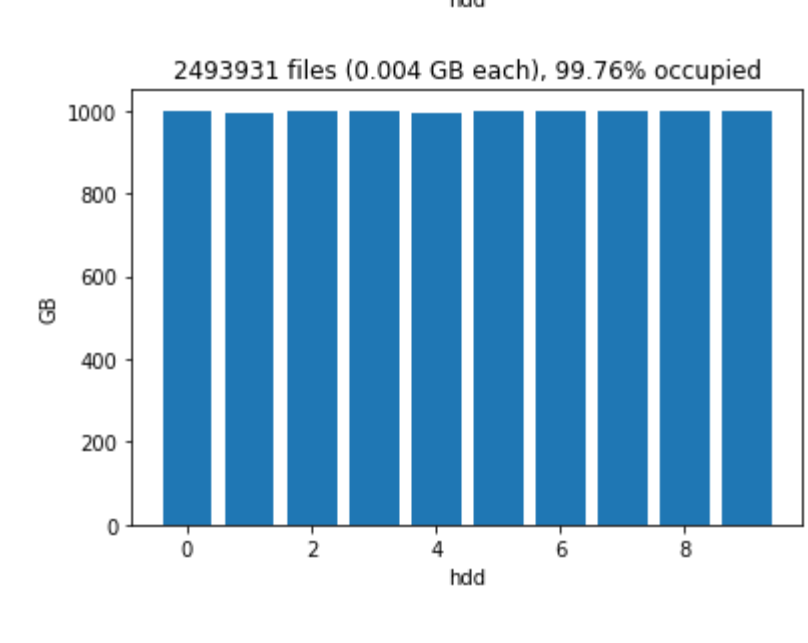
- we create an empty length-10 vector `x`, so that its element represent the amount of stored data in the drives at any given time;
- we initiate a loop where as long as no hard drive has reached 1 TB (1000 GB) we sample uniformly an integer `i` in [0, 9], then increment `x[i]` by `filesize`;
- when one drive i.e. one `x` component reaches 1000 GB we stop.

Once we have this vector we can use a barplot to represent the insides of our simulated drives.

In [6]:
```
@njit
def populate_drives(filesize) :
    x = np.zeros(10)

    while (x < 1000).all() : # 1 TB = 1000 GB
        x[np.random.randint(10)] += filesize
    return x

def populate_and_plot_drives(filesize, verbose = True) :
    x = populate_drives(filesize)
    if verbose :
        print(f"x in GB (filesize = {filesize} GB):", x)
    fig, ax = plt.subplots()
    ax.bar(np.arange(10), x)
    # we add to the plot 2 informations: how many files were successfully written + how much storage ou
t of the 10 TBs we were able to use
    ax.set_title(f"{int(x.sum()/filesize)} files ({filesize} GB each), {x.sum()/1e4:.2%} occupied") #
 x.sum() = tot. n. of GB saved, x.sum()/1e4 = occupied fraction of the 10 available TBs
    ax.set_xlabel("hdd")
    ax.set_ylabel("GB");

    return x

list_of_file_sizes = [10, 1, 4e-3] # the assignment asks to make plots for filesize = 10 GB, 1 GB, 4 MB
list_of_drives = []

for i in list_of_file_sizes :
    list_of_drives.append(populate_and_plot_drives(i))
```

```
x in GB (filesize = 10 GB): [ 790.   920.   800.   860.   770. 1000.   890.   860.   670.   740.]
x in GB (filesize = 1 GB): [ 968.   931. 1000.   901.   912.   943.   935.   957.   874.   924.]
x in GB (filesize = 0.004 GB): [ 996.804   994.82    999.46    998.64    995.552 1000.      997.932   997.0
16
  997.636   997.864]
```





3.1a & 3.1b

With this seed the first execution of the above cells yields the following results.

In [7]:
```
def summary(x, f) :
    return f"{f} GB files: we wrote {int(x.sum()/f)} files i.e. {int(x.sum()/f)}*{f} = {x.sum():.2f} GB
s of data" + "\n" + f"we were able to exploit {x.sum():.0f}/10000 = {x.sum()/1e4:.4f} = {x.sum()/1e4:.
2%} of the available space"

print(summary(list_of_drives[0], list_of_file_sizes[0]))
```

```
10 GB files: we wrote 830 files i.e. 830*10 = 8300.00 GBs of data
we were able to exploit 8300/10000 = 0.8300 = 83.00% of the available space
```

If the simulation cell is executed again multiple times (without resetting the seed) one usually finds values between 80% and 92% (mostly centered around ~85%); this relatively large spread is analyzed below.

3.2a & 3.2b

With this seed the first execution of the simulation cell yields the following results.

In [8]:
```
print(summary(list_of_drives[1], list_of_file_sizes[1]))
```

```
1 GB files: we wrote 9345 files i.e. 9345*1 = 9345.00 GBs of data
we were able to exploit 9345/10000 = 0.9345 = 93.45% of the available space
```

This time the results are "tighter", in the sense that if the cell is re-executed with a seed reset one typically finds values between 92% and 97% (mostly centered around ~95-96%), and gives us the first hint that smaller file size equals more efficient exploitation of space. *Why does this happen?*

3.3

With this seed the first execution of the simulation cell yields the following results.

In [10]:
```
print(summary(list_of_drives[2], list_of_file_sizes[2]))
```

```
0.004 GB files: we wrote 2493931 files i.e. 2493931*0.004 = 9975.72 GBs of data
we were able to exploit 9976/10000 = 0.9976 = 99.76% of the available space
```

With the a 4 MB file size it's basically impossible to find numbers outside the range 99.5-99.8% by manually re-executing the code cell. *Why is this the case?*
We also notice that the smaller the file size the flatter the obtained distribution. Once again we ask: *but why?*

Increasing efficiency explanation

We can explain the above results using a simple semi-quantitative argument based on the law of large numbers as follows.
Let's temporarily imagine that n, the number of files we write/the number of iterations the algorithm runs, is not a random variable, but actually a predetermined constant (for example fixed in advance by us). If this is the case then the number of files we are able to store inside drive i obeys a *binomial distribution* with $p = 0.1$; this follows from the fact that at each algorithm iteration every hard drive has a 10% chance of being "chosen" by our uniform generator, therefore every iteration is equivalent to one Bernoulli trial per hard disk (hence the binomial, by "accumulation" of Bernoulli trials). This fact is useful because it lets us give an order of magnitude estimate of how large n needs to be to actually fill almost all of the available space.
Indeed let's say that we chose n such that on average each hard drive is almost full (in order to reproduce the 80-90%+ occupied storage values we observed empirically); this implies that if n_i is the number of files in drive i we then have

$$E[X_i] = E[n_i f] \lesssim 1000 \text{ GB}$$

where we defined X_i as the number of occupied GBs in drive i and f as the constant size of a single file.
Since

$$n_i \sim \text{Binom}(n, p = 0.1) \quad \forall i = 1, \dots, 10$$

we obtain by linearity:

$$E[X_i] = E(n_i f) = npf = \frac{nf}{10} \approx 1000 \text{ GB} \quad \forall i = 1, \dots, 10$$

Hence:

$$f\frac{n}{10} \approx 1000 \text{ GB} \implies n \approx \frac{10^4 \text{ GB}}{f}$$

If we want to be more precise we can write:

$$n \lesssim \frac{10^4 \text{ GB}}{f}$$

since all X_i but one are strictly $< 10^3$ GB when the algorithm stops the algorithm; this means that we expect n to be slightly less than $10^4/f$.
The result that $n \propto 1/f$ approximately is quite intuitive and reasonable, since to cover the same fixed distance with smaller steps more time is needed.
For completeness let us remark that in the unluckiest case possible (at least in principle) every file will end up in the same HD, which means that the algorithm will only last $10^3/f$ iterations (since a single hard drive can only fit 1000 GBs of data); hence

$$\frac{10^3 \text{ GB}}{f} \le n \le \frac{10^4 \text{ GB}}{f}$$

but this result is purely of academic interest; for all practical purposes it suffices to say n will always be slightly less than $10^4/f$.
Even though this estimate for the upper bound on n was obtained using a somewhat simplistic model it actually yields a pretty close number to the "experimental" results (as shown in the next cell); it also correctly reproduces the scaling of the problem (notice that when we go from $f = 10$ GB from $f = 1$ GB we save ~1 o.o.m. more files, and similarly when f drops to 4 MB we gain another 3 extra o.o.m.'s).

In [12]:
```
def estimate_n(filesize) :
    return 1e4/filesize

for x, f in zip(list_of_drives, list_of_file_sizes) :
    observed_n = x.sum()/f
    predicted_n = estimate_n(f)
    print(f"relative difference between observed {observed_n} and predicted {predicted_n} ({f} GB): {n
p.abs(predicted_n - observed_n)/predicted_n:.2%}")
```

```
relative difference between observed 830.0 and predicted 1000.0 (10 GB): 17.00%
relative difference between observed 9345.0 and predicted 10000.0 (1 GB): 6.55%
relative difference between observed 2493931.000000268 and predicted 2500000.0 (0.004 GB): 0.24%
```

Now that we can somewhat satisfyingly explain/predict the observed values of n we can explain why the efficiency increases, why the histogram gets flatter with decreasing f and why the above approximation gets better with smaller f's.
The above equations say that the smaller f the longer the algorithm will last; indeed as we will always have the files of size f n is almost guaranteed to be close to its theoretical maximum. We also notice that even in the worst case scenario we can be sure that, if $f_1, f_2, f_3 = 10$ GB, 1 GB, 0.004 GB respectively then we will have $n_1 \le 10 n_2$ and $n_2 \le 1000 n_3$ - so that 4 MB files make n inevitably huge.
Using the above we obtain:

$$100 \le n_1 \le 1000$$
$$1000 \le n_2 \le 10^4$$
$$2.5 \cdot 10^5 \le n_3 \le 2.5 \cdot 10^6$$

This means that the smaller f the more random numbers we sample from the uniform distribution, and therefore *the closer we get to having a flat distribution* (by the law of large numbers); this already explains the observed shapes! But if a small f makes the empirical distribution closer to the theoretical uniform one then it means that *the smaller f the closer the bin heights*; this in turn guarantees that all bins have a height very close to the one of the full drive, which stopped the algorithm. This finally gives two important results:

- With a smaller f all of the X_i's (i.e. GBs stored in drive i) become almost equal to each other, and in particular to the one which reached 1 TB; this means that smaller $f \implies$ higher efficiency/percentage of overall occupied storage.
- Due to the above the smaller f the more "constant" the X_i's become, which means their std gets much smaller since they all get closer to their ideal unknown mean. Notice that as they all get closer to 1 TB the assumption $E(X_i) \approx 1$ TB becomes more and more correct, which makes our $n \approx 10^4/f$ result less of a simplistic approximation and more of an actual prediction; this explains why as f gets smaller this approximation gets better and better.

To recap: with smaller values of f more files will need to be written before any hard drive is full, which means we're sampling a lot more uniform random numbers and therefore obtaining a distribution very close to uniform. By "flatness" this makes all bins approximately as tall as the one that reached 1 TB, hence we get closer to filling 100% of the available space (while also making our formula more correct).

Notice that this increase in efficiency increases monotonically with a decreasing f, which seems to suggest that going even further than 4 MB is desirable; this is actually not true due to other reasons. In particular notice that a typical computer is able to process 4 MB much more efficiently than single bytes, hence 4 MBs offers a pretty good compromise between storage efficiency and IOPS - while at the same time making individual blocks much lighter to e.g. move around the internet (which is relevant if the hard drives are far away) compared to, say, 1 GB (which would offer comparable efficiency).

3.4

Let's organize our results in a small dataframe for visual clarity's sake.

In [13]:
```
df = pd.DataFrame(np.vstack((list_of_file_sizes, list(map(np.mean, list_of_drives)), list(map(np.std, l
ist_of_drives)))).T, columns = ["filesize", "mean", "std"])
display(df) # obviously the mean is just tot GB written / 10, i.e. we obtain x.sum() once again but div
ided by 10
```

	filesize	mean	std
0	10.000	830.0000	90.664216
1	1.000	934.5000	33.767588
2	0.004	997.5724	1.528796

We already explained why the mean scales as $1/f$: smaller files $\implies$ more files are written $\implies$ more numbers are sampled from the uniform distribution $\implies$ the resulting histogram is closer to flat $\implies$ the rectangles' heights get closer $\implies$ each drive's content becomes closer to the full one $\implies$ the overall storage efficiency increases.
This long chain can also explain why the standard deviation of the sample decreases with f: indeed as we know σ measures the amount of variability in the observed dataset, which is why

$$\text{constant data points} \implies \sigma = 0$$
$$\text{wildly different data points} \implies \sigma \gg 1$$

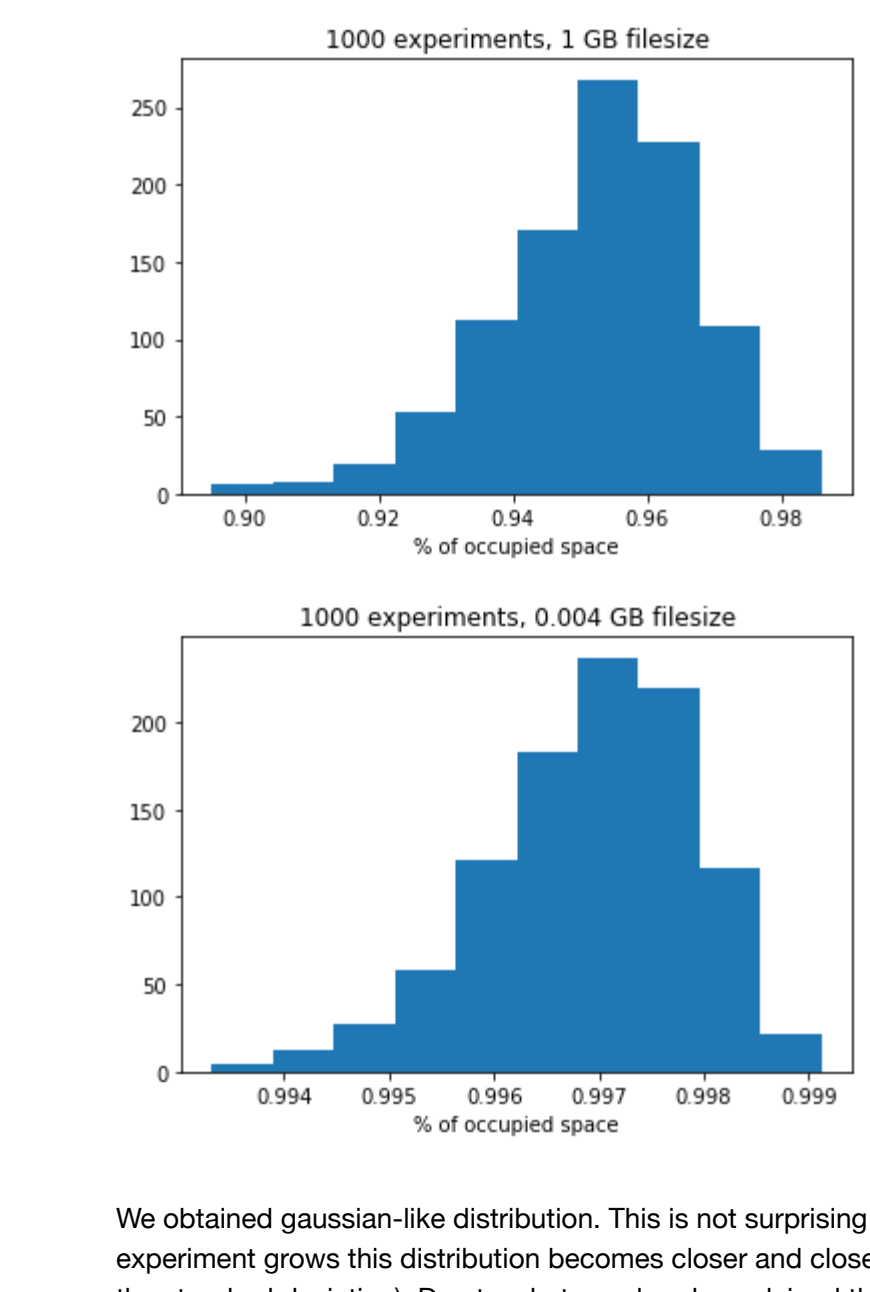
This suffices to explain the decreasing σ behaviour: since a smaller and smaller f makes the drives closer and closer to full we obtain that

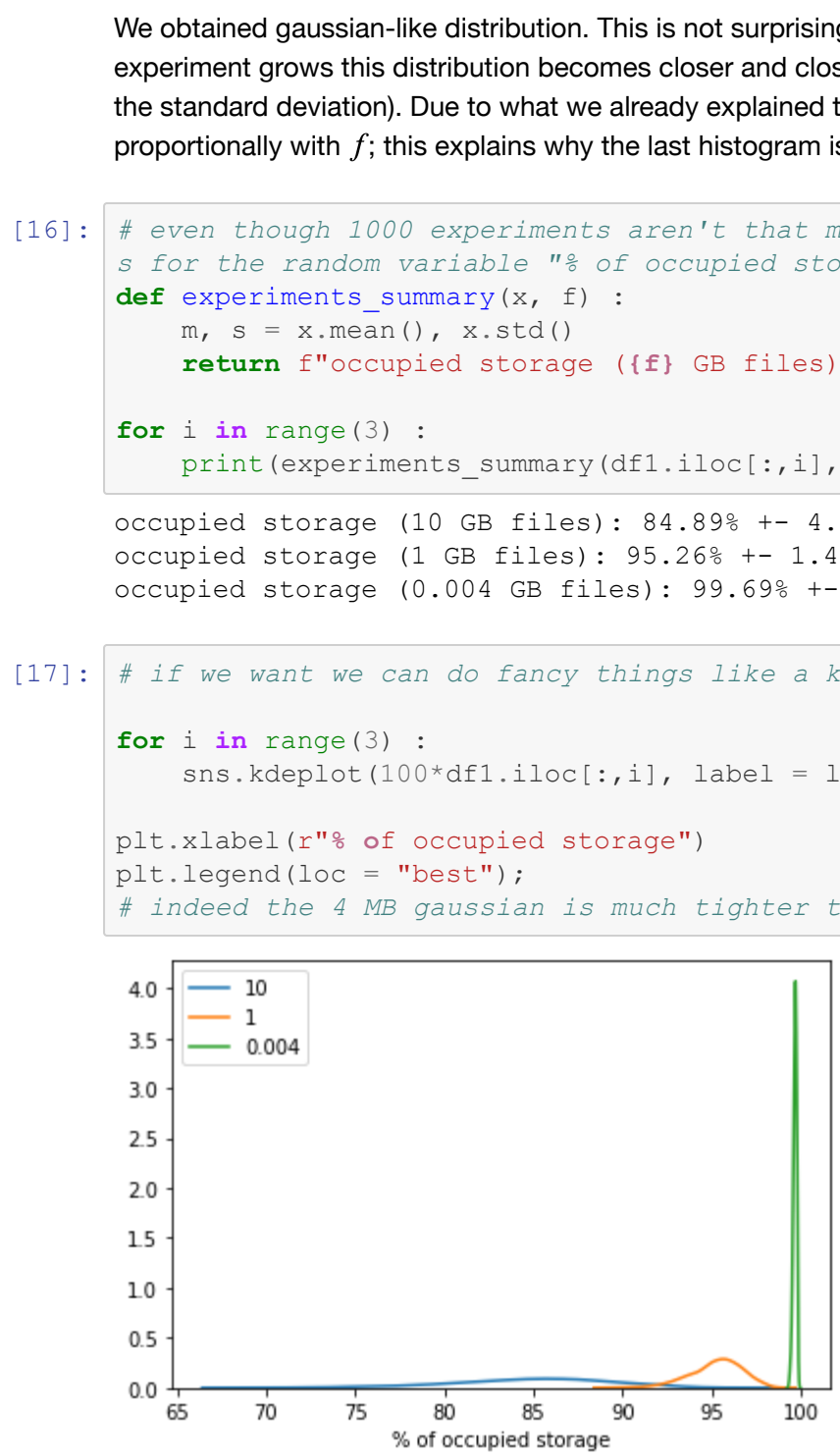
$$\text{smaller } f \implies \text{almost constant } \vec{x} \text{ components} \implies \text{smaller } \sigma$$

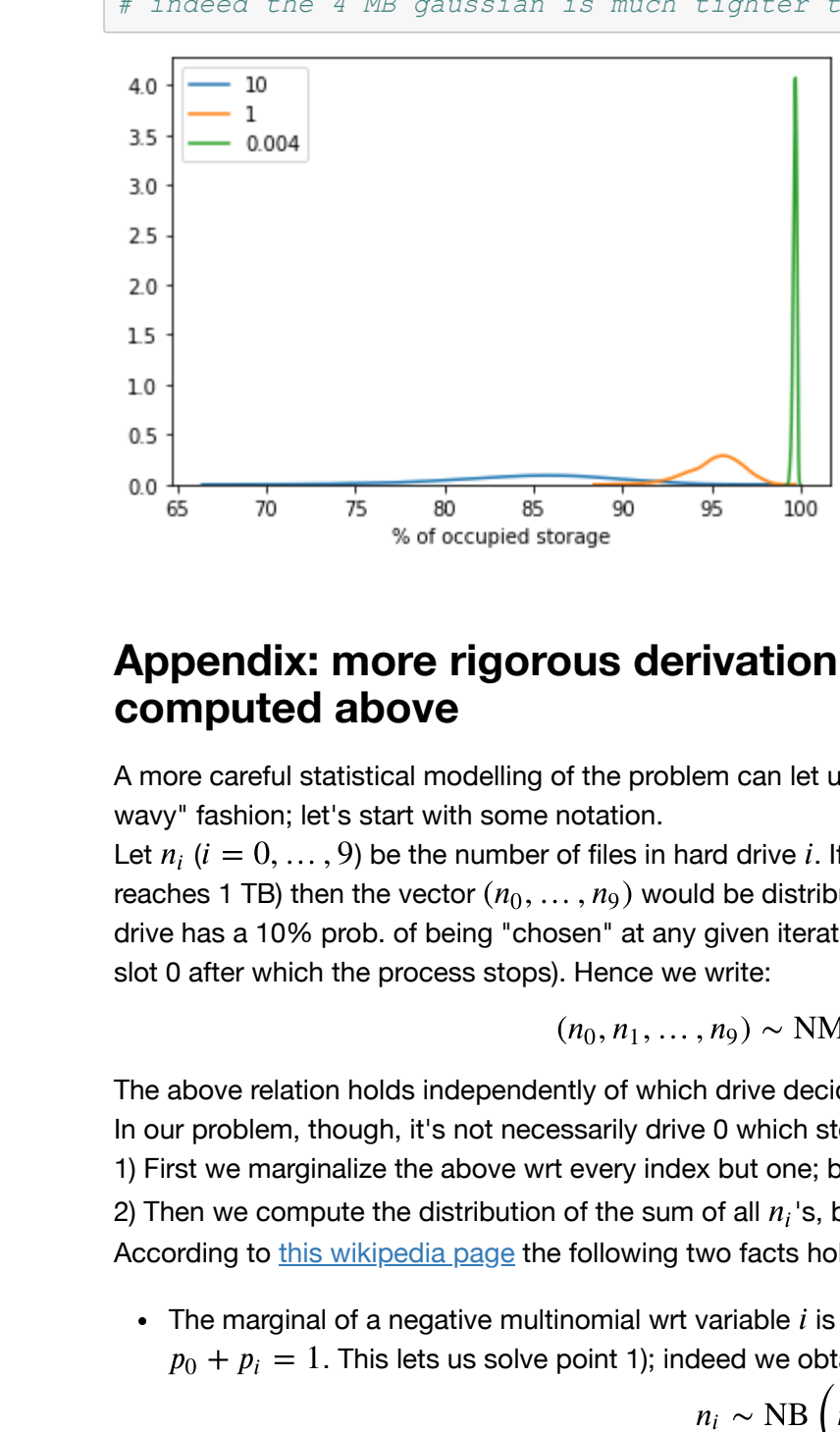
Let's now repeat the experiment for many times and plot how the random variable "% of occupied space" is distributed.

In [14]:
```
@njit
def obtain_drive_statistic(filesize, n_experiments = 1000, percent = True) :
    y = np.zeros(n_experiments)
    for i in range(n_experiments) :
        y[i] = populate_drives(filesize).sum()
    if percent :
        return y/1e4
    else :
        return y

def plot_drive_statistic(filesize, n_experiments = 1000) :
    y = obtain_drive_statistic(filesize, n_experiments)
    fig, ax = plt.subplots()
    ax.hist(y)
    ax.set_title(f"{n_experiments} experiments, {filesize} GB filesize")
    ax.set_xlabel("% of occupied space");

    return y
```

In [15]:
```
# since the histogram for 4 MB takes a bit of time to speed up debugging we save the results
save = True

if not os.path.exists("./df1.csv") :
    l = [plot_drive_statistic(f) for f in list_of_file_sizes]
    df1 = pd.DataFrame(np.array(l).T, columns = [str(f) for f in list_of_file_sizes])
    if save :
        df1.to_csv("df1.csv", index = False)
    #display(df1)
else :
    df1 = pd.read_csv("df1.csv")
    for i in range(3) :
        fig, ax = plt.subplots()
        ax.hist(df1.iloc[:,i])
        ax.set_title(f"1000 experiments, {df1.columns[i]} GB filesize")
        ax.set_xlabel(r"% of occupied space");
```







We obtained gaussian-like distribution. This is not surprising due to the central limit theorem; indeed we expect that as the number of experiment grows this distribution becomes closer and closer to a gaussian with mean equal to the "true" mean (with the same holding for the standard deviation). Due to what we already explained the "true" mean scales inversely with f, whereas the "true" std scales proportionally with f; this explains why the last histogram is so much "tighter" horizontally and more on the right wrt the first, etc.

In [16]:
```
# even though 1000 experiments aren't that many they nonetheless allow us to obtain more solid estimate
s for the random variable "% of occupied storage"
def experiments_summary(x, f) :
    m, s = x.mean(), x.std()
    return f"occupied storage ({f} GB files): {m:.2%} +- {s:.2%}"

for i in range(3) :
    print(experiments_summary(df1.iloc[:,i], list_of_file_sizes[i]))
```

```
occupied storage (10 GB files): 84.89% +- 4.37%
occupied storage (1 GB files): 95.26% +- 1.48%
occupied storage (0.004 GB files): 99.69% +- 0.10%
```

In [17]:
```
# if we want we can do fancy things like a kde

for i in range(3) :
    sns.kdeplot(100*df1.iloc[:,i], label = list_of_file_sizes[i])

plt.xlabel(r"% of occupied storage")
plt.legend(loc = "best");
# indeed the 4 MB gaussian is much tighter than the others, and almost centered around 100%
```

Appendix: more rigorous derivation of the n average upper bound estimate computed above

A more careful statistical modelling of the problem can let us obtain results analogous to what we already derived, but in a less "hand-wavy" fashion; let's start with some notation.
Let n_i ($i = 0, \dots, 9$) be the number of files in hard drive i. If the whole process stopped when e.g. drive 0 is full (instead of the first which reaches 1 TB) then the vector $(n_0, \dots, n_9)$ would be distributed according to a *negative multinomial* with $\vec{p} = (0.1, \dots, 0.1)$ (since each drive has a 10% prob. of being "chosen" at any given iteration) and $r = 1000 \text{ GB}/f$ (since by definition r is the number of successes in slot 0 after which the process stops). Hence we write:

$$(n_0, n_1, \dots, n_9) \sim \text{NM}\left(r = \frac{1000 \text{ GB}}{f}, \vec{p} = (0.1, 0.1, \dots, 0.1)\right)$$

The above relation holds independently of which drive decides when the process stops.
In our problem, though, it's not necessarily drive 0 which stops the algorithm; this means we need to do the following.
1) First we marginalize the above wrt every index but one; by symmetry this will give us the distribution of any n_i.
2) Then we compute the distribution of the sum of all n_i's, because this will give us the distribution of $n = \sum_{i=0}^{9} n_i$.
According to this wikipedia page the following two facts hold:

- The marginal of a negative multinomial wrt variable i is a *negative binomial* with the same r and p_i rescaled in such a way that $p_0 + p_i = 1$. This lets us solve point 1); indeed we obtain
$$n_i \sim \text{NB}\left(r' = r = \frac{1000 \text{ GB}}{f}, p' = 5p = \frac{1}{2}\right)$$
- The random variable "sum of two positive/negative binomial/multinomial distributions with the same $p/\vec{p}$" is a distribution of the same kind, with the same $p/\vec{p}$ and the other parameter equal to the sum of the two n/r's; this lets us solve point 2), since if combined with the previous bullet point it implies that
$$n \sim \text{NB}\left(r = \frac{10^4 \text{ GB}}{f}, p = \frac{1}{2}\right)$$

Notice that n's distribution cannot actually be a NB, since the probability of n surpassing the max value is 0 (whereas the NB has no such constraint); this means that if we want to compute e.g. $\langle n \rangle$ we need to "cut" the rightmost part of the distribution, which means that $\langle n \rangle$ is slightly less than $\langle \text{NB}(r = 10^4 \text{ GB}/f, \ p = 1/2) \rangle$. Since

$$\left\langle \text{NB}\left(r = \frac{10^4 \text{ GB}}{f}, p = \frac{1}{2}\right)\right\rangle = \frac{pr}{1-p} = \frac{10^4 \text{ GB}}{f}$$

we obtain

$$\langle n \rangle \lesssim \frac{10^4 \text{ GB}}{f}$$

which is *the same result we already obtained*, but in a more "professional" way. As a bonus we can also produce approximations for other interesting quantities, such as the std.

Assignement 4: Rest APIs & Block Chain Technology

Below there is a simple client implementation with its usage.

In [25]:
```
import requests
import json
import time

class blockchain:

    def __init__(self, team, url):
        self.accounts = []
        self.chain = []
        self.current_hash = ''
        self.requests = []
        self.team = team
        self.url = url

    def get(self):
        response = requests.get(self.url, verify = False)
        return json.loads(response.text)

    def request(self, coin, stealfrom):
        merit = json.loads(json.dumps({"operation": "merit", "team": self.team, "coin": coin, "stealfro
m": stealfrom}, indent = 3))
        claim = json.loads(json.dumps({"operation": "claim", "team": self.team}, indent = 3))
        requests.post(self.url, json=merit, verify = False)
        time.sleep(10)
        requests.post(self.url, json=claim, verify = False)
        print('Done')
```

4.1.1

In [26]:
```
client = blockchain('21centuryboys','https://pansophy.app:8443')
client.request(1000, 'genesis')
```

```
/opt/anaconda3/lib/python3.7/site-packages/urllib3/connectionpool.py:1004: InsecureRequestWarning: Un
verified HTTPS request is being made to host 'pansophy.app'. Adding certificate verification is stron
gly advised. See: https://urllib3.readthedocs.io/en/latest/advanced-usage.html#ssl-warnings
  InsecureRequestWarning,
Done
/opt/anaconda3/lib/python3.7/site-packages/urllib3/connectionpool.py:1004: InsecureRequestWarning: Un
verified HTTPS request is being made to host 'pansophy.app'. Adding certificate verification is stron
gly advised. See: https://urllib3.readthedocs.io/en/latest/advanced-usage.html#ssl-warnings
  InsecureRequestWarning,
```

4.1.2

In [27]:
```
int(24*60*60/10)
```

Out[27]: 8640

4.2.1

This function computes the block hash that will be in the data of the next block in the chain. If one block is modified and we calculate the new hash we see that it is different with respect to the hash (of the same block) computed at the creation of the next block in the chain. So if the 2 hash are identical we are sure that the block is unchanged and repeating the operation block by block like a 'chain' we can verify that nobody compromised the all blockchain.

4.2.2

It's simple: we get all the block's data using a GET request; we start from the first block and we compute its hash; if the hash is the same reported in the second block we repeat the same operation on the second block, on the third and so on... at the last block we have validated all the chain.

4.2.3

With POST requests the chain becomes bigger and bigger, so the GET respose of the server might take much time to trasmit the JSON file of all the chain to the client.

So we have that a POST request will take tha same time indipendetly from the chain length. Istead the GET request will take a time proportional to the chain length N: t = cost. + O(N). Where the cost. term is due for example for the criptography of the data. How we can see below this cost. term is negligible w.r.t the trasmission time and it is about 100 ms.

In [28]:
```
%time client.request(1000, 'genesis');
```

```
/opt/anaconda3/lib/python3.7/site-packages/urllib3/connectionpool.py:1004: InsecureRequestWarning: Un
verified HTTPS request is being made to host 'pansophy.app'. Adding certificate verification is stron
gly advised. See: https://urllib3.readthedocs.io/en/latest/advanced-usage.html#ssl-warnings
  InsecureRequestWarning,
Done
CPU times: user 20.4 ms, sys: 6.51 ms, total: 26.9 ms
Wall time: 10.3 s
/opt/anaconda3/lib/python3.7/site-packages/urllib3/connectionpool.py:1004: InsecureRequestWarning: Un
verified HTTPS request is being made to host 'pansophy.app'. Adding certificate verification is stron
gly advised. See: https://urllib3.readthedocs.io/en/latest/advanced-usage.html#ssl-warnings
  InsecureRequestWarning,
```

In [30]:
```
# %time client.get();
```

We report a small percentage of the data printed by the command above just for illustration purposes:

```
CPU times: user 74.1 ms, sys: 26.5 ms, total: 101 ms Wall time: 712 ms
Out[29]:{'accounts': {'21centuryboys': 1,'AC Milan': 1,'Andreas J. Peters': 1,'AstonVilladiVilla': 401016,[...]}}
```

In [23]:
```
import matplotlib.pyplot as plt
import numpy as np

plt.plot(range(1000), 0.1 + 0.1*np.arange(1000), label = 'GET')
plt.plot(range(1000), (0.1 + 10)*np.array([1]*1000), label = 'POST')
plt.legend(loc = 'best')
plt.xlabel('chain length')
plt.ylabel('time request')
plt.tick_params(
    #axis='x',          # changes apply to the x-axis
    which='both',      # both major and minor ticks are affected
    bottom=False,      # ticks along the bottom edge are off
    left=False,        # ticks along the top edge are off
    labelbottom=False,
    labelleft=False) # labels along the bottom edge are off
```


4.2.4

The server can recompute the current account balances just starting from the first block and recomputing all the transations having care of checking the hash.

4.2.5

One advantage of using the REST API is the facility with we can create a client object that interacts with the server. We only need two functions (in this case) for our purpose: GET and POST. Furthermore the HTTPS protocol allows for secure (with the SSL criptography) and authenticated (not in this case) comunication between client and server. The JSON data formata's advantages are the facility with the server can handle the coming data (for example in a data streaming process) and the client can create the data to send. One disadvantage can be the scalability problem of the GET request discussed in 4.2.3.

In [ ]: