Final assignment of "Management and Analysis of Physics Datasets"

Part 2: Data management

University of Padua - Physics of Data

Dr. Andreas-Joachim Peters

UNIVERSITÀ DEGLI STUDI DI PADOVA

Name	ID number	mail@studenti.unipd.it
Chiara Maccani	2027591	chiara.maccani
Samuele Piccinelli	2027650	samuele.piccinelli
Tommaso Stentella	2027586	tommaso.stentella
Cristina Venturini	2022461	cristina.venturini.5

Index

In [1]:
```
# import libraries and useful dependencies

from wget import download
from functools import reduce
import os
import numpy as np
from matplotlib import pyplot as plt
import requests
import time
```

Index

0. Fun Exercise

$+$ is the digit-wise **XOR operation** $\oplus$ of the binary representation of each number. For each operation:

Operation	Binary representation	Result
$2+1$	$10+01=11$	3
$2+5$	$010+101=111$	7
$3+7$	$011+111=100$	4
$4+5$	$100+101=001$	1
$5+9$	$0101+1001=$ $=1100$	12

Index

1. Redundancy

We are programming a file based RAID-4 software algorithm. For this purpose we are converting a single input (raid4.input) file into four data files `raid4.0`, `raid4.1`, `raid4.2`, `raid4.3` and one parity file `raid4.4` - the four data and one parity file we call *stripe files*.

To do this we are reading in a loop sequentially blocks of four bytes from the input file until the whole file is read: in each loop we write one of the four read bytes round-robin to each data file, compute the parity of the four input bytes and write the result into the fifth parity file and we continue until all input data has been read. If the last bytes read from the input file are not filling four bytes, we consider the missing bytes as zero for the parity computation.

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

First we delete the files produced by the script, *if existing*. The program opens the input file, reads it and appends padding at the end if the length of the byte string can not be divided by the number of files the data are going to be striped in. It writes consequently on four different files, based on each identifying index.

The function `parity` computes the parity over a given list. After having opened the destination file, the data is divided in 4-bytes long chunks and the parity of each of them is computed and written on file. The same function can be applied to point **1.2.** to calculate the parity of all bytes within one stripe file - provided the input is now the entire data in each one of the stripped files.

In [2]:
```
%%bash
rm -f -- raid.0
rm -f -- raid.1
rm -f -- raid.2
rm -f -- raid.3

rm -f -- raid.4

rm -f -- reco
rm -f -- reconstructed.input
```

In [3]:
```
ifd = 'raid4.input'

# download file if not present in current directory
if not os.path.isfile(ifd):
    download('https://apeters.web.cern.ch/apeters/pd2021/raid4.input')
```

In [4]:
```
fd = ['raid.{}'.format(num) for num in range(4)]
odf = [open(file, 'wb') for file in fd]

with open(ifd, 'rb') as raid:
    data = raid.read()
    raid.close()
```

```
padding = 0
while len(data) % len(fd) != 0:
    data += bytes([0])
    padding += 1

for i, byte in enumerate(data):
    odf[i % 4].write(bytes([byte]))

for file in odf:
    file.close()
```

In [5]:
```
def parity(series: list):
    '''computes parity over a list'''
    return reduce((lambda x, y: x ^ y), series)
```

In [6]:
```
pdf = open('raid.4', 'wb')

chunks = [data[i:i+4] for i in range(0, len(data), 4)]

for chunk in chunks:
    pdf.write(bytes([parity(chunk)]))

pdf.close()
```

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

In [7]:
```
fd = ['raid.{}'.format(num) for num in range(5)]
odf = [open(file, 'rb') for file in fd]

v_parity = []
for file in odf:
    data = file.read()
    v_parity.append(bytes([parity(data)]))
    file.close()
```

In [8]:
```
size_raids = sum(os.path.getsize(f) for f in fd)
size_input = os.path.getsize(ifd)
overhead = (size_raids-size_input)*100/size_input

print('The size overhead is', round(overhead,2), '%!')
```

```
The size overhead is 25.0 %!
```

The computed column-wise parity acts as a **checksum** for each stripe file. The size overhead is 1/4 of the original file, as expected, since one more file has been written on disk - at most with three added padding bytes which are negligible when computing the file size.

1.3. What is the 5-byte parity value? Write it it in hexadecimal format like `P5 = 0x[q0][q1][q2][q3][q4]`, where the `[qx]` are the hexadecimal parity bytes computed by xor-ing all bytes in each stripe file.

In [9]:
```
parity_value = ' '.join(str(parity.hex()) for parity in v_parity)
print('The 5-byte parity value computed by xor-ing all bytes in each stripe file is P5 = 0x', p
```

```
The 5-byte parity value computed by xor-ing all bytes in each stripe file is P5 = 0x a5 07 a0 9
c 9e
```

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, which is the XOR between all data stripes D0, D1, D2, D3 and the parity P.

The contents of this file would be a list of zeros of the length of the number of bytes contained in the parity file. Indeed:

$$
\begin{aligned}
R = P \oplus D0 \oplus D1 \oplus D2 \oplus D3 = & \quad (1)\\
= (D0 \oplus D1 \oplus D2 \oplus D3) \oplus (D0 \oplus D1 \oplus D2 \oplus D3) = & \quad (2)\\
= (D0 \oplus D0) \oplus (D1 \oplus D1) \oplus (D2 \oplus D2) \oplus (D3 \oplus D3) = & \quad (3)\\
= 0 \oplus 0 \oplus 0 \oplus 0 = 0 & \quad (4)
\end{aligned}
$$

1.5. After some time you recompute the 5-byte parity value as in point 1.3. The result is now `P5 = 0x a5 07 a0 01 9e` . Something has been corrupted. You want to reconstruct the original file `raid4.input` using the 5 stripe files. Describe how you can recreate the original data file. Which stripe files do you use and how do you recreate the original data file with the correct size?

By comparing the parity of each file computed in point **1.4.**, the faulty disk is `raid.3` . The original data file can be reconstructed by computing the XOR of the remaining uncorrupted files (`raid.0` , `raid.1` , `raid.2` and the parity file `raid.4`); below we reconstruct the missing block and compute its parity to verify that it matches the value previously found. The padding can be then removed through the counter `padding` declared in point **1.1.**; at last, we check for the reconstructed input file to have the same size of the initial one. We notice that the original file is still retrieved even if the padding is not removed.

In [10]:
```
corrupted = 'raid.3'
fd.remove(corrupted)
odf = [open(file, 'rb') for file in fd]

with open('reco', 'wb') as reconstructed:
    for raid0, raid1, raid2, raid4 in zip(odf[0].read(), odf[1].read(), odf[2].read(), odf[3].r
        par = parity([raid0, raid1, raid2, raid4])
        reconstructed.write(bytes([par]))
    reconstructed.close()

for file in odf:
    file.close()
```

In [11]:
```
with open('reco', 'rb') as file:
    data = file.read()
    par_reco = parity(data)
    print('The reconstructed', corrupted,'parity is', bytes([par_reco]).hex())
    file.close()
```

```
The reconstructed raid.3 parity is 9c
```

In [12]:
```
fd = ['raid.0', 'raid.1', 'raid.2', 'reco']
odf = [open(file, 'rb') for file in fd]

with open('reconstructed.input', 'wb') as recinput:
    count = 0
    for raid0, raid1, raid2, raid3 in zip(odf[0].read(), odf[1].read(), odf[2].read(), odf[3].r
        par = [raid0, raid1, raid2, raid3]
        count += 1
        if count == len(data):
            par = bytes(par[:len(par) - padding])
            recinput.write(par)
        else:
            par = bytes(par)
            recinput.write(par)
    recinput.close()

for file in odf:
    file.close()
```

In [13]:
```
os.path.getsize('reconstructed.input') == os.path.getsize('raid4.input')
```

Out[13]:
```
True
```

Index

2. Cryptography

A friend has emailed you the following text: `K]amua!trgpy` . She told you that her encryption algorithm works similar to the Caesar cipher:

- To each ASCI value of each letter a secret key value is added (note that ASCII values range from 0 to 255);
- Additionally, to make it more secure, a variable *nonce* is added to each ASCII number. The nonce start value is 5 for the first character of the message and for each following character it is incremented by 1.

2.1. Is this symmetric or asymmetric encryption? Explain why.

The encryption used in this algorithm is symmetric, since the encryption and decryption key are the same.

With symmetric encryption we refer to an encryption technique in which the encryption key is the same as the decryption one. This makes the algorithm easier to implement. It's required that both parties are in possession of the keys, so it's impossible to require a key exchange with this kind of encryption algorithms. In these cases, the exchange is performed via asymmetric key algorithms, which are generally more complex to implement and execute, but that allow this exchange to happen in a safe manner. Afterwards, the communication is encrypted using only symmetric key algorithms, which guarantee a safe and fast communication.

2.2. Write a small brute force program which tests keys in $[0, 255]$ and use a dictionary approach to figure out the original message. What is the decryption algorithm/formula to be used?

The algorithm to be used is implemented in the function `caesar_decrypt` : the nonce (initially set to 5) and the input-given key are subtracted to each character in the encrypted message. The modulus 255 is then taken to avoid values out of the ASCII range.

In `caesar_break` every key in the $[0, 255]$ interval is tried: the so obtained pseudo-words are compared to a document (`wordlist.txt`) containing all english words (refer to here for reference). If present in the dictionary, the final message is printed out along with the working key.

In [14]:
```
if not os.path.isfile('wordlist.txt'):
    download('https://www.ics.uci.edu/~kay/wordlist.txt')
```

In [15]:
```
def caesar_decrypt(message: str, key: int) -> str:
    decrypted = ''
    nonce = 5
    for char in message:
        value = ord(char) - nonce - key
        decrypted += chr(value % 255)
        nonce += 1
    return decrypted

def caesar_break(code: str):
    with open('wordlist.txt') as words:
        dictionary = {word.strip() for word in words}
        words.close()
    for x in range(255):
        word = caesar_decrypt(code, x)
        if set(word.split()).issubset(dictionary):
            print('The used key is {}, the original message text is {}!'.format(x, word))
```

In [16]:
```
text = 'K]amua!trgpy'

caesar_break(text)
```

```
The used key is 245, the original message text is Padova rocks!
```

Index

3. Object Storage

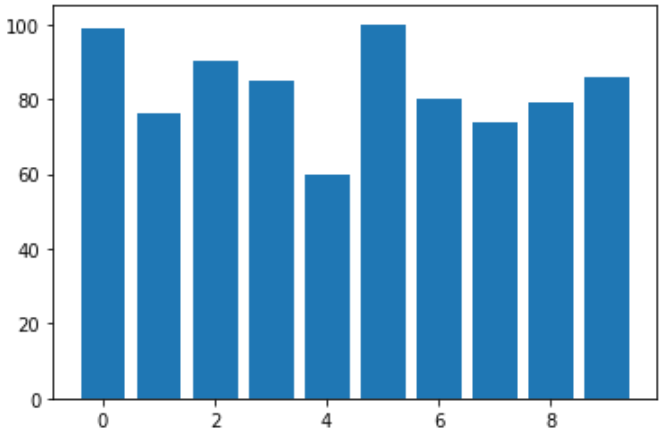
Imagine we have a system with ten hard disks (10 locations). We enumerate the location of a file using an index of the hard disk $[0, \ldots, 9]$. Our hash algorithm for placement produces hashes which are distributed uniform over the value space for a flat input key distribution.
We want now to simulate the behaviour of our hash algorithm without the need to actually compute any hash value. Instead of using real filenames, which we would hash and map using a hash table to a location (as we did in the exercise), we are "computing" a location for "any" file by generating a random number for the location in the range $[0, \ldots, 9]$ to assign a file location. To place a file in the storage system we use this random location where the file will be stored and consumes space.

Assume each disk has 1 TB of space, we have 10 TB in total. Place as many files of 10 GB size as possible to hard disks choosing random locations until one hard disk is full.

3.1. Write a program which simulates the placement of 10 GB files to random locations and account the used space on each hard disk. Once the first hard disk is full, you stop to place files. How many files did you manage to place? What is the percentage of total used space on all hard disks in the moment the first disk is full?

In [2]:

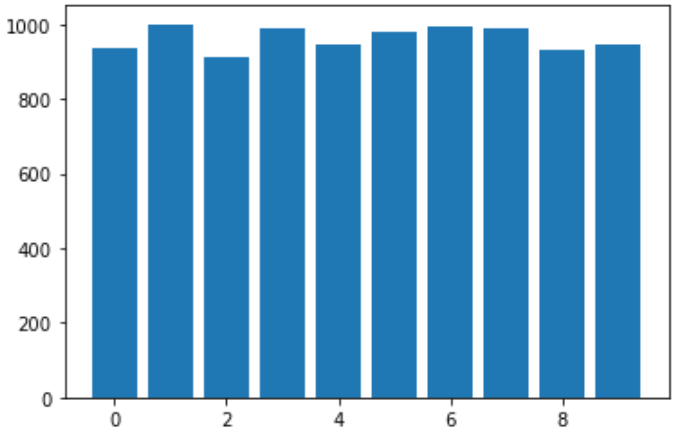
```
# set seed
np.random.seed(12345)

storage = np.zeros((10))
files_count = 0
block_size = 10
nblock = int(1000/block_size)

i = np.random.randint(0,10,nblock*10)

while np.all(storage < nblock):
    storage[i[files_count]] += 1
    files_count += 1

plt.bar([i for i in range(10)], storage)
plt.show()
print('Total number of stored files:',files_count)
print('Percentage of total used space: %.2f'%(files_count/nblock/10*100))
print('Average used space (GB): ',storage.mean()*block_size,'\tstd:',storage.std()*block_size)
print('Average used space (nfiles): ',storage.mean(),'\tstd:',storage.std(),'\tsqrt(avg):',np.s
```




```
Total number of stored files: 829
Percentage of total used space: 82.90
Average used space (GB):  829.0         std: 113.61778029868388
Average used space (nfiles):  82.9      std: 11.361778029868388         sqrt(avg): 9.1049437120
7203
```

3.2. Repeat the same task placing 1 GB files until the first hard disk is full. How many files did you manage to place? What is the percentage of total used space on all hard disks in the moment the first disk is full?

In [3]:
```
np.random.seed(12345)

storage = np.zeros((10))
files_count = 0
block_size = 1
nblock = int(1000/block_size)

i = np.random.randint(0,10,nblock*10)

while np.all(storage < nblock):
    storage[i[files_count]] += 1
    files_count += 1

plt.bar([i for i in range(10)], storage)
plt.show()
print('Total number of stored files:',files_count)
print('Percentage of total used space: %.2f'%(files_count/nblock/10*100))
print('Average used space (GB): ',storage.mean()*block_size,'\tstd:',storage.std()*block_size)
print('Average used space (nfiles): ',storage.mean(),'\tstd:',storage.std(),'\tsqrt(avg):',np.s
```



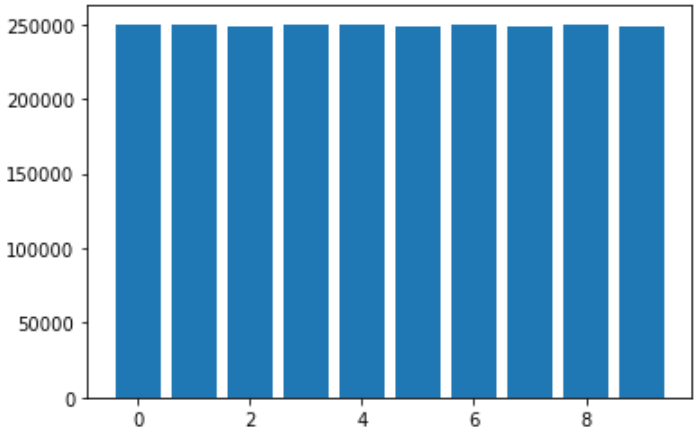

```
Total number of stored files: 9630
Percentage of total used space: 96.30
Average used space (GB):  963.0         std: 29.281393409467384
Average used space (nfiles):  963.0     std: 29.281393409467384         sqrt(avg): 31.032241298
3658
```

3.3. Based on this observation: why do you think object storage typically stores fixed size blocks of 4 M and not files of GBs size as a whole (so called block storage approach)? Run the same program for 4 M block sizes and demonstrate the benefits.

Using smaller blocks allows to have a greater sample and therefore a better approximation of the distribution (i.e. uniform in this case). However, this also increases the number of blocks to be read given a certain amount of data. The size of 4 M is a tradeoff between these two situations. Moreover the usage of fixed size blocks makes the storage system suitable for applying useful techniques such as striping, which can increase the performance and reliability of the system.

In [4]:
```
np.random.seed(12345)

storage = np.zeros((10))
files_count = 0
block_size = 0.004
nblock = int(1000/block_size)

i = np.random.randint(0,10,nblock*10)

while np.all(storage < nblock):
    storage[i[files_count]] += 1
    files_count += 1

plt.bar([i for i in range(10)], storage)
plt.show()
print('Total number of stored files:',files_count)
print('Percentage of total used space: %.2f'%(files_count/nblock/10*100))
```

```
print('Average used space (GB): ',storage.mean()*block_size,'\tstd:',storage.std()*block_size)
print('Average used space (nfiles): ',storage.mean(),'\tstd:',storage.std(),'\tsqrt(avg):',np.s
```




```
Total number of stored files: 2491278
Percentage of total used space: 99.65
Average used space (GB):  996.5112      std: 2.1825928067323965
Average used space (nfiles):  249127.8  std: 545.6482016830992  sqrt(avg): 499.127037937237
```

3.4. Compute the average used space on all hard disks and the standard deviation for the average used space for 10 GB and 1 GB and 4 M files. How is the standard deviation correlated to the block size and why? If we now repeat such an experiment for many more (thousands) of hard disks, which kind of distribution do you get when you do a histogram of the used space of all hard disks?

The standard deviation of the number of files in a hard disk goes with the square root of the average number of files, so it increases by reducing the block size. The standard deviation of the used storage however is derived by the previous one by multiplying it with the blocksize. With respect to the number of blocks the block size decreases linearly, while the standard deviation increases with a $1/2$ power-law, resulting in a reduction of the standard deviation of the used storage per disk.

The shape of the distribution, i.e. the way the sample reproduces the uniform distribution, depends both on the total number of samples (blocks) and on the ratio between the number of blocks in a single disk and the total number of disks. This is because we have an upper limit in the number of "extractions", i.e. we have to stop when a disk is full.
In fact, the distribution must be sampled extensively over all the disks before one of them reaches completion and this could be achieved either by decreasing even more the size of the blocks or by increasing the capacity of a single disk. Therefore the expected distribution is still uniform, but how well it is sampled depends on these choices.

Index

4. REST APIs & Block Chain Technology

Under https://pansophy.app:8443 you find a Crypto Coin Server exporting a simple Block Chain. The task is to implement a client and use a simple REST API to submit transactions to the Block Chain: your goal is to book coins from other people's accounts to your own account.

The server implements a *Proof Of Time algorithm* (minimum of 10 seconds). To add a transaction to move coins to your account, you have to submit a merit request and you have to let time pass before you can send a claim request to execute your transaction on the Block Chain. If you claim your transaction too fast after a merit request, your request is discarded.

4.1.1. Use Python program, doing the HTTPS requests respecting Proof of Time. You will have to add at least one successful transaction to the Block Chain.

The following code iterates over a predefined number of transactions and, for each of them, *steals* 1 coin from the account with the highest balance. This is achieved by getting the updated version of the blockchain

at every iteration and looping over the accounts ledger.

In [20]:

```
url = 'https://pansophy.app:8443'

selected_account = 'genesis'
merit = {
        'operation': 'merit',
        'team': 'CCTS',
        'coin': 1,
        'stealfrom': selected_account
        }

claim = {
        'operation':'claim',
        'team':'CCTS'
        }

n_transaction = 1
for i in range(n_transaction):
    state = requests.get(url, verify=False)

    if state.status_code != 200:
        print('Error occurred during GET')
        break

    json_state = state.json()
    top = 0
    for account, coin in json_state['accounts'].items():
        if top < coin:
            top = coin
            selected_account = account
    if json_state['accounts'][selected_account] == 1:
        print('Blockchain saturated')
        break

    merit['stealfrom'] = selected_account

    requests.post(url, json=merit, verify=False)
    time.sleep(10)
    requests.post(url, json=claim, verify=False)

print(requests.get(url, verify=False).json()['accounts'])
```

```
/Users/tommaso/anaconda3/lib/python3.8/site-packages/urllib3/connectionpool.py:981: InsecureReq
uestWarning: Unverified HTTPS request is being made to host 'pansophy.app'. Adding certificate
verification is strongly advised. See: https://urllib3.readthedocs.io/en/latest/advanced-usage.
html#ssl-warnings
  warnings.warn(
/Users/tommaso/anaconda3/lib/python3.8/site-packages/urllib3/connectionpool.py:981: InsecureReq
uestWarning: Unverified HTTPS request is being made to host 'pansophy.app'. Adding certificate
verification is strongly advised. See: https://urllib3.readthedocs.io/en/latest/advanced-usage.
html#ssl-warnings
  warnings.warn(
/Users/tommaso/anaconda3/lib/python3.8/site-packages/urllib3/connectionpool.py:981: InsecureReq
uestWarning: Unverified HTTPS request is being made to host 'pansophy.app'. Adding certificate
verification is strongly advised. See: https://urllib3.readthedocs.io/en/latest/advanced-usage.
html#ssl-warnings
  warnings.warn(
/Users/tommaso/anaconda3/lib/python3.8/site-packages/urllib3/connectionpool.py:981: InsecureReq
uestWarning: Unverified HTTPS request is being made to host 'pansophy.app'. Adding certificate
verification is strongly advised. See: https://urllib3.readthedocs.io/en/latest/advanced-usage.
html#ssl-warnings
  warnings.warn(
{'21centurysboys': 1, 'AC Milan': 1, 'Andreas J. Peters': 1, 'BellaFra': 1, 'BellaZio': 1, 'CAN
E': 999922, 'CCTS': 3, 'FaoTom': 1, 'Fogliame': 1, 'GioC': 1, 'Giovanni': 1, 'GitPush': 1, 'Lor
enzoDomenichetti': 8, 'LuPi AndAle': 1, 'Oesais': 1, 'PM': 1, 'PaoloMuschio': 1, 'Team1': 1, 'b
ellaZio': 1, 'genesis': 1, 'whitenoisematters': 50}
```

4.1.2. What is the maximum number of transactions one given team can add to the Block Chain in one day?

Assuming the time for POST to be negligible (and hash computation time on the server side to be less than the *Proof of time* of $10\ s$) then the number of transactions per day is constrained by the *Proof of Time* interval of $10\ s$, resulting in 8640 transactions per day.

4.2. The server has a function to compute a hash of a block in the Block Chain:

```
def calculate_hash(self):
    block_of_string = '{}{}{}{}{}'.format(self.index, self.team, self.prev_hash,
self.coins, self.timestamp)
    self.my_hash = hashlib.sha256(block_of_string.encode()).hexdigest()
    return self.my_hash
```

4.2.1. Explain what this function does. Why is this "the key" for Block Chain technology?

This function takes a generic input file and returns a 256 bit word (visualized in hexadecimal representation). The function is designed in such a way that:

- It is very difficult to invert;
- A small change in the input produces a large change in the output;
- The cases of two element of the domain having the same image are very rare.

These features make it suitable for file validation as the file can remain open to the public while the hash guarantees protection and is used to check the validity of the data. In the blockchain it plays a key role in enabling the cross-control of the content of a node by the others.

4.2.2. If you have the knowledge of the hash function, how can you validate the contents of the Block Chain you received using a GET request to make sure that nobody has tampered with it? Explain the algorithm to validate a Block Chain.

One can go through the nodes of the blockchain and check the following:

1. Check if the block indexes are in the right order;
2. Verify if every block points to the right previous block, through comparing the value of their hashes;
3. Verify the proof of work: one example of a function for this scope would be

```
@staticmethod
def verifying_proof(last_proof, proof):
    # verifying the proof: does hash(last_proof, proof) contain 4 leading 0s?
    guess = f'{last_proof}{proof}'.encode()
    guess_hash = hashlib.sha256(guess).hexdigest()
    return guess_hash[:4] == '0000'

@staticmethod
def proof_of_work(last_proof):
    ''' this algorithm identifies a number f' such that hash(ff')
    contains 4 leading 0s: f is the previous f', f' is the new proof '''
    proof_no = 0
    while BlockChain.verifying_proof(proof_no, last_proof) is False:
    proof_no += 1
    return proof_no
```

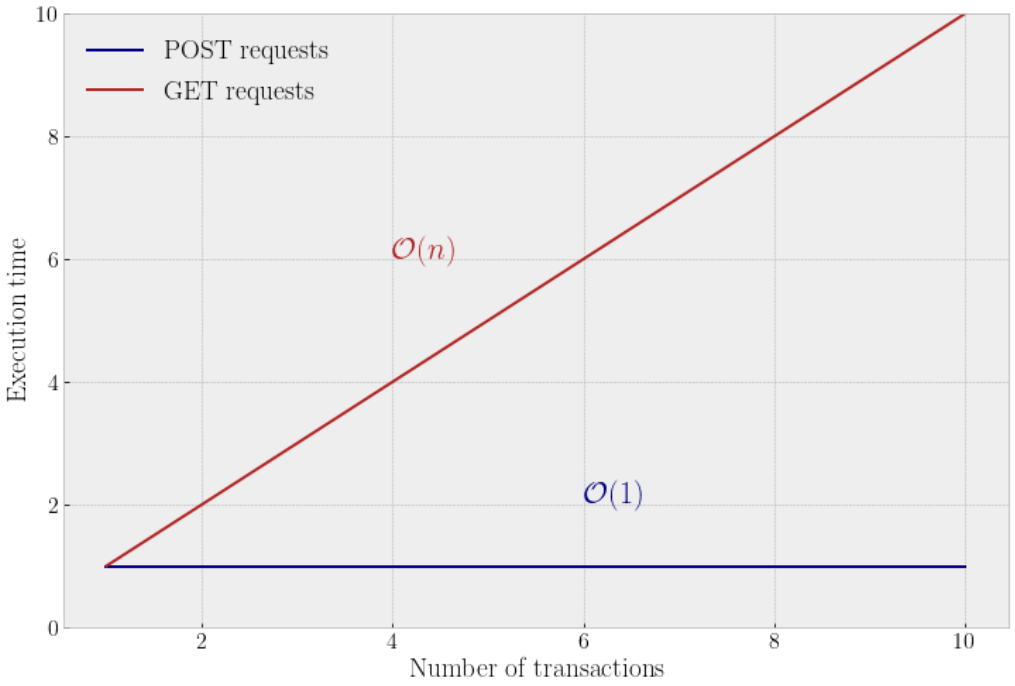
1. Check the blocks' timestamps: are the blocks created in a sequential time order?

4.2.3. Why might the GET REST API run into scalability problems? Express the scalability behaviour of execution times of GET and POST requests in Big O notation in relation to the number of transactions recorded in the Block Chain. Draw execution time vs transactions for GET and POST requests.

The `GET REST API` involves an exchange of a JSON file that has a length approximately proportional to the number of transactions n, so the execution time increases with the number of transaction as $\mathcal{O}(n)$. `POST` on the other hand involves, both for **merit** and **claim**, the exchange of a JSON file of fixed length. As a consequence the execution time is constant with respect to the number of transaction: $\mathcal{O}(1)$.

4.2.4. If the Crypto server goes down, the way it is implemented it loses the current account balances. How can the server recompute the account balances after a restart from the saved Block Chain?

Each and every block holds the information on a transaction up to the very first block which contains the initial state of the chain, i.e. the total amount of coins and their belonging to the *genesis* account. The account balances can be then reconstructed going through all the history of the transactions saved in the chain.



4.2.5. What are the advantages of using a REST API and JSON in a client-server architecture? What are possible disadvantages?

REST is an architectural style which was created to address the problems of SOAP, which is a standardized protocol that sends messages using other protocols such as HTTP and SMTP.
In order to understand the advantages and disadvantages of REST APIs it is useful to understand how REST differs from SOAP, and in which context it constitutes an actual improvement.

First of all, as stated above, REST is an architectural style, which means it's an architecture that lays down a set of guidelines to follow if you want to provide a RESTful web service.
This implies that REST has a more flexible architecture than SOAP: it consists of only loose guidelines and lets developers implement the recommendations in their own way. It allows different messaging formats, such as HTML, JSON, XML, and plain text, while SOAP only allows XML.
REST is also a more lightweight architecture, so RESTful web services have a better performance.
REST is almost always better for web-based APIs, as it makes data available as resources (e.g. user) as opposed to services (e.g. getUser) which is how SOAP operates.
Furthermore, REST inherits HTTP operations, meaning you can make simple API calls using the well-known HTTP verbs like GET, POST, PUT, and DELETE.

On the other hand, SOAP is an official protocol, so it comes with strict rules and advanced security features. This implies a higher complexity, and so it requires more bandwidth and resources which can lead to slower page load times.
Nevertheless, SOAP will likely continue to be used for enterprise-level web services that require high security and complex transactions, which REST is not able to guarantee. APIs for financial services, payment gateways, CRM software, identity management, and telecommunication services are commonly used examples of SOAP.

As for what concerns the JSON format, it is more lightweight and less verbose, and also easier to read and write as well.
However, XML still has some advantages that JSON can't provide. For example, XML lets you place metadata within tags and also handles mixed content better - especially when mixed node arrays require detailed expressions.

In summary:
Advantages: a REST API is easy to use and there is no need for complex infrastructure (HTTPS protocol is widespread and just a web client is needed), JSON format is easy to read.

Disadvantages: HTTPS protocol is not so secure when used without any certification from both the client and the server side. JSON is not efficient in storing informations.