Final exam - Algorithmics

Note

- **Name:** Naiara Alonso Montes, C46483
- I have completed this exam on my own with no other people helping me.
- Multiple chatbots for helping to create have been used for this exam. That does not mean that the whole code has been generated by them, but they have been used to help me with the most difficult parts of the exam. The code has been adapted to my needs and the requirements of the exam. All resulting images from ex 03 are attached in the zip file together with the code of the rest of the exam. The images belonging to the dataset have a size of half a GB, so for convenience, they have not been included. All the images used are properly referenced in the solution.
- Aprox spent hours: 30

Ex 1

Single line and band (1st and 2nd)

How cosine works

Cosine similarity is a measure of similarity between two non-zero vectors of an inner product space that measures the cosine of the angle between them. The cosine of 0° is 1, and it is less than 1 for any other angle. It is thus a judgment of orientation and not magnitude: two vectors with the same orientation have a cosine similarity of 1, two vectors at 90° have a similarity of 0, and two vectors diametrically opposed have a similarity of -1, independent of their magnitude.

For the images, it is calculated by using the dot product of the normalized vectors (RGB values) of the images.

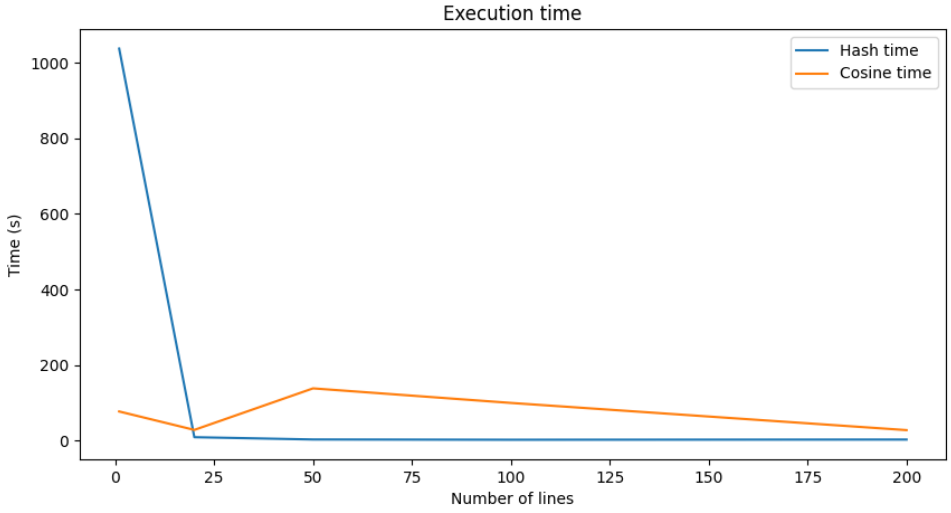
Solution

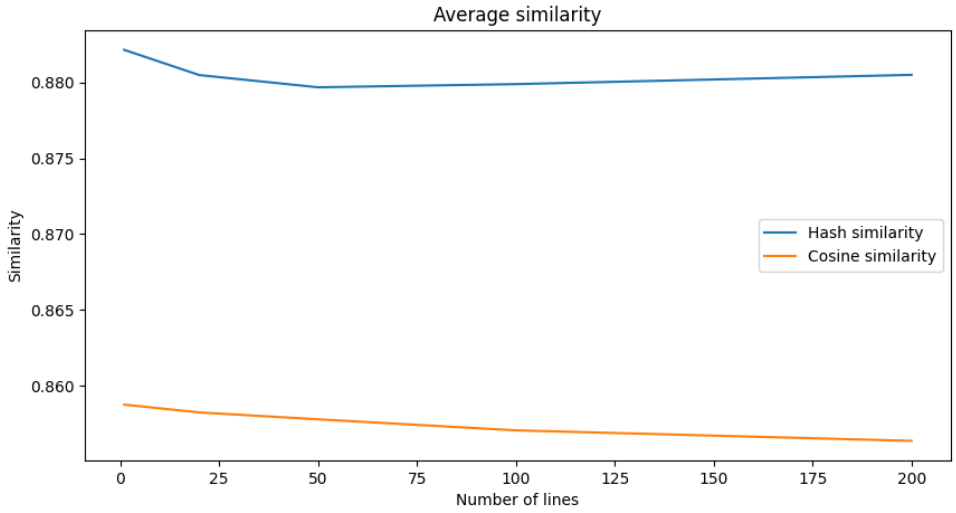
For this exercise I tried two different approaches, using cosine similarity and using hash function. I will explain both of them.

Cosine similarity consumes too much time, so I used a "pre-processing" step to make the code run faster. What it does at first is, block by block of `n` vertical lines, it hashes the block, and then, after hashing all the blocks, it calculates the cosine similarity on the RGB block with the highest coincidence among hashes.

To reduce complexity, I decided to work with consecutive blocks of `n`, which it is not ideal, but with small sizes of `n` it performs well.

I compared both methods, raw cosine and hashing + cosine, here are the results:





As seen in the images, the only moment where hashing + cosine is worst is when searching for a single line. In all other cases, it is better in time. In terms of similarity, it is always better. The results suggest that the hashing techniques is better for a wider band search, faster and accurate, in other case, if the search is for a single line, the raw cosine is better.

Here is also one example of the similarity between the images:

The result shows two consecutive bands of the same image, despite not having the same pattern, the color matches and makes the difference between the comparison among other bands.

One of the questions was *which size of the block is ideal in terms of similarity*. It is obvious that as long as the block size increases, the similarity decreases. This is because it is more difficult to find common patterns in larger blocks. However, the time complexity is reduced. The election of `n` depends mainly on the size of the image in which we want to perform the search.

This is an example of a wider band search, but in this case the coincidence is clear to the human eye despite the low similarity:

Effects on resized/reshaped images:

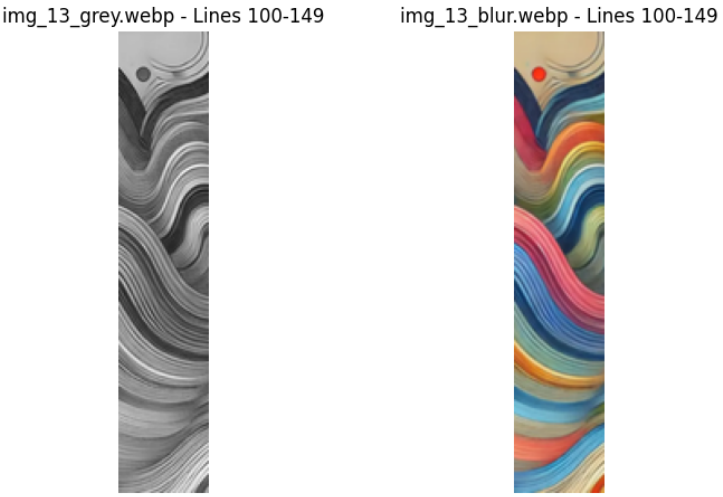
For this part, I will work with 4 images, with 3 of them being transformations of the same one and other one similar to the original one. I will use the hashed pre-processing method to compare the images. I used transformations of `imag_13.webp` and the original image `imag_14.webp`.

Based on the results I can state the following:

- The solution is able to produce a high value of similarity between the image `img_14.webp` and the transformations of `img_13.webp`.
- Those matches found including parts of the colorized image and the black and white image are the same section of the image whereas the comparison between grey-scale chooses a different section of the image with a high similarity. None of the color same color image are show as a high similarity. Only among them.
- The negative color transformation is not a good choice as it does not show a high similarity with the original image, it suggests that cosine focuses on the color and not the shape.

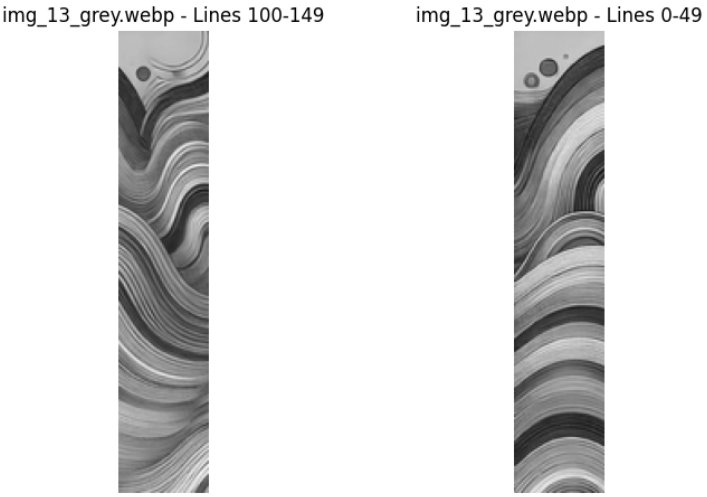
Some examples:

Similarity: 0.9110

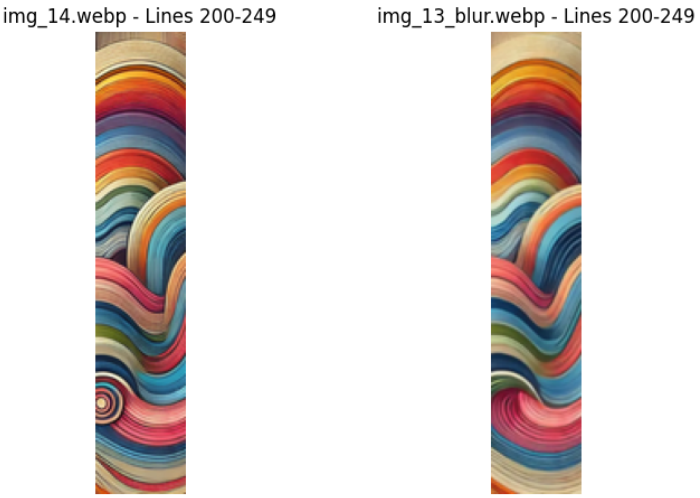


This is the highest similarity found, the first one is the grey-scale and the blurred image. This gives the hint that the color is not really the most influential factor in the similarity.

Similarity: 0.8971



Similarity: 0.8955



This last example is among one of the blurred bands and the "similar" image. Above all the transformations, the cosine similarity detect the "same" colors and the fewer deviations in the color, despite the shape of the image.

Ex 2

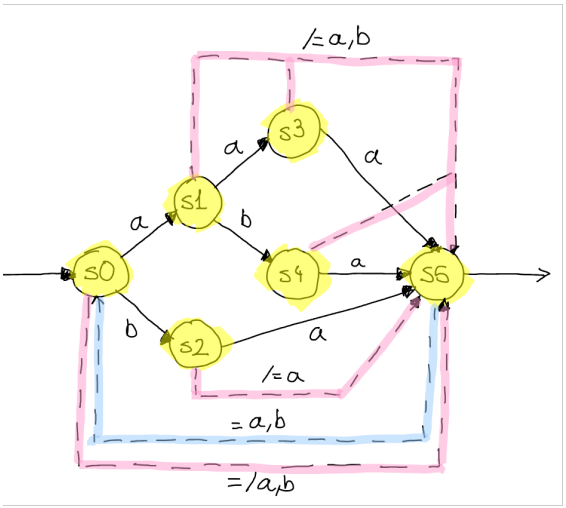
The statement of the exercise is difficult to interpreter as the regular expressions are not clear and well separates. I found it hard to distinguish where is the end and the beginning of the next one. **Under my interpretation, there are 4 regular expressions**

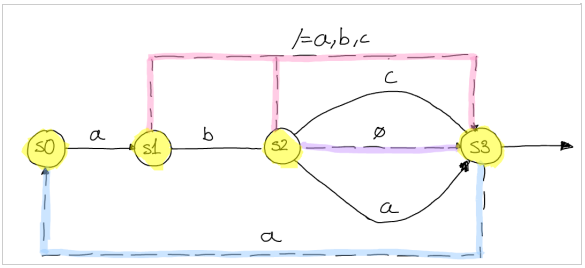
- `(aba|ba|aaa)+`
- `(ab|abc|aba)+`
- `(ba(c+))`
- `(cbb|bba|a)+`

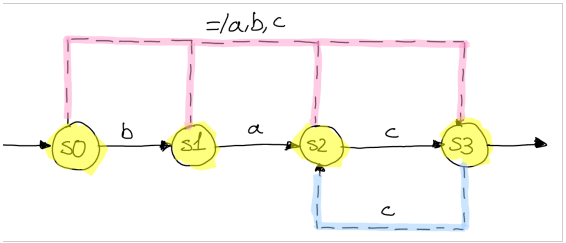
From now on, I will refer to them as regex 1, regex 2, regex 3 and regex 4. In the images, the doted pink line is equivalent to leading the automata to an error state where any of the sequences break. The blue one is for the repetition of the sequence again. For simplicity, it has been represented like this.

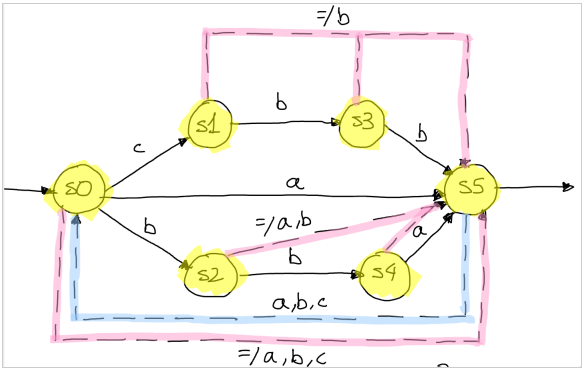
There is a problem with the seconds regex, with `ab` and `aba`. Basically the order of this expressions in this regex matter. If we want to match for example `abab`. There are two options: two matches of `ab` or a single match of `aba`. In the order they appear, `aba` with be ignored as the it is a "first match". It can be checked on this webpage. From now on, I will consider just `(ab|abc)+`

Here are the corresponding Thomson's NFA for each of the regular expressions:









For Regex 1, the NFA tabulation is the following:

Current State	Input Symbol	Next State(s)

Current State	Input Symbol	Next State(s)
s0	a	s1
s0	b	s2
s0	=/a, b	s5
s1	a	s3
s1	b	s4
s1	=/a, b	s5
s2	a	s5
s2	=/a	s5
s3	a	s5
s3	=/a	s5
s4	a	s5
s4	=/a	s5
s5	a,b	s0

For Regex 2, the NFA tabulation is the following:

Current State	Input Symbol	Next State(s)
s0	a	s1
s0	=/a	s3
s1	b	s2
s1	=/b	s3
s2	c	s3
s2	a	s0
s3	a	s0

For Regex 3, the NFA tabulation is the following:

Current State	Input Symbol	Next State(s)
s0	b	s1
s0	=/b	s3
s1	a	s2
s1	=/a	s3
s2	c	s3
s2	=/c	s3
s3	c	s2

For Regex 4, the NFA tabulation is the following:

Current State	Input Symbol	Next State(s)
s0	a	s5
s0	b	s2

Current State	Input Symbol	Next State(s)
s0	c	s1
s0	=/a, b, c	s5
s1	b	s3
s1	=/b	s5
s2	b	s4
s2	=/b	s5
s3	b	s5
s3	=/b	s5
s4	a	s5
s4	=/a	s5
s5	a,b,c	s0

The transitions sx -> =/y -> xw, can be also considered to back to the initial state.

Code and results

For the code I needed to add extra states. My idea was to use the states described in the table to the code implementation, and change the states according to the input char (next element in the text string). Even though it is not perfect, it achieves decent results, compared to other Internet tools.

Here are the results for all the regex and for all the generated files:

	injected_output_1.t	injected_output_2.t	injected_output_3.t	injected_output_4.txt
regex 1	2462	2427	2040	2407
regex 2	955	944	938	934
regex 3	265	266	267	265
regex 4	3836	3857	3794	3798

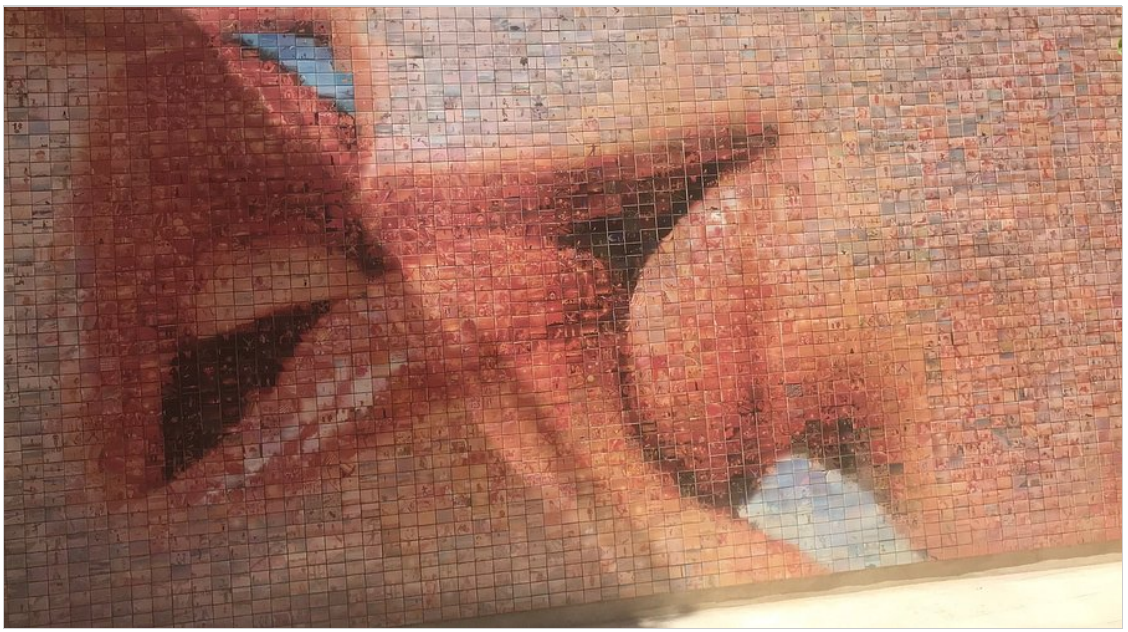
The explanation for the results is that among the random generated text, there are patterns more common than others. For example the `(ba(c)+)` is the most difficult to find as it is a very specific pattern. The other ones are more common and easier to find. The regex 4 is the most common one. If we consider that we only have input 10 strings that match the regex. The percentage of random matching for its specific file is the following:

Regex	Random matching percentage
1	99.59%
2	98.94%
3	96.25%
4	99.73%

This percentage is **really** high considering that it is random. Augmenting the complexity of the regex and the alphabet size will decrease it.

Ex 3

To solve this exercise I decided to choose the *multi photography collage*. I opted to do this one as in Barcelona, Spain there is a similar mural.



It has been created by collecting multiple tiles with a photo printed on it. There is a trick here as the pictures are "solarized" to make the image more similar to the required color and tonality.

My solution is a bit simplistic but effective. To create the image I did the following steps:

Pre-processing dataset

- Obtain the images from the collage. The source of the images are open-source images from the Kaggle website. I used 2 different datasets as I needed huge variety of images. The first one is taken from Wonders of the World (here) and from Landscape Color and Greyscale images (here). From the last dataset I only took the color images.
- Resize the images to have squared shape.
- Obtain the most predominant color of each image (most common color in the image).
- In a separate file, save the images with the most predominant color.

Create the collage

- Took the image and *blur* it. Apply a filter of `n` size to reduce dimensions. It takes the average of the pixels in the filter and assigns it to the center pixel. It reduces the dimensions of the image.
- For each pixel, compare the color of the pixel with the color of the images in the dataset. The most similar image is assign to that specific pixel.
- Render the new image.

Computational complexity and compute time

I will separate the complexity by the pieces of the code:

- `square_dataset_img` : $O(N + M \cdot (W \cdot H + W' \cdot H'))$, where N is the number of entries in the CSV, M is the number of images in the dataset, W and H are the width and height of the images in the dataset and W' and H' are the width and height of the images in the dataset.
- `convert_images_to_pixel_art` : $O(W \cdot H)$, where W and H are the width and height of the image.
- `convert_to_collage` , $O(W \cdot H + (N \cdot f^2))$, where W and H are the width and height of the image, N is the number of images in the dataset and f is the filter size.

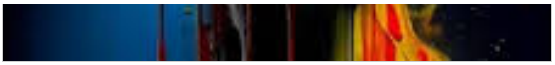
For the compute time it is the most time-consuming as the image quality increases and the filter decreases. The main limitation was to find a balance for creating a good quality examples, with the most time spend for any of the following reaching up to 1 hour and a half.

Results

The best obtained result is the following, with a filter of 4x4:

This result is obtained for a 894x898 image. The time processing of the time increases as the size of the image increases and with the reduction of the filter size.

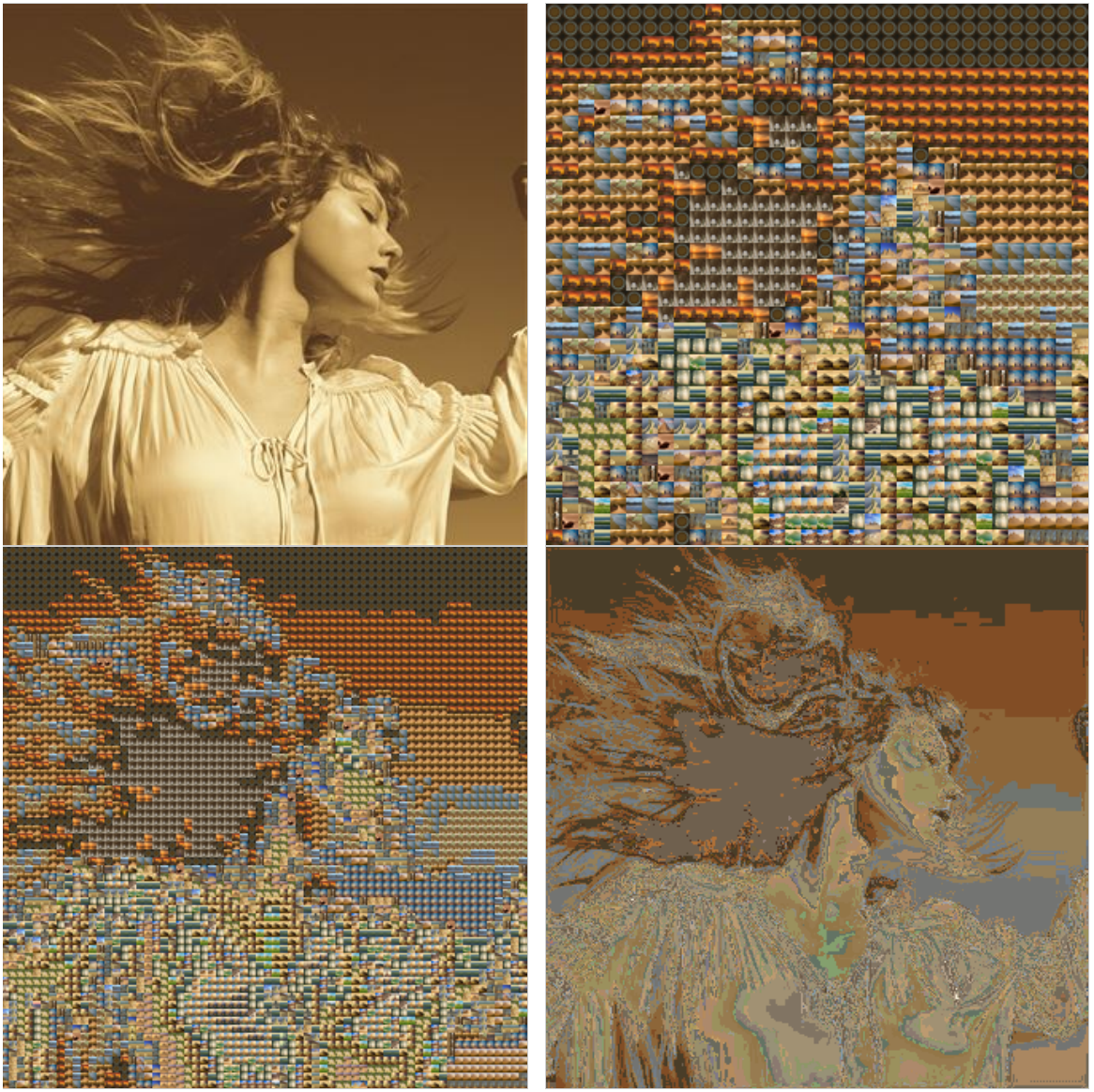
I also tried to do not blur the image and the results are slightly better, considering the image is 225x225, with no filter:


In both images it is obvious that the collage is not perfect and struggles with the background color. It is specially remarkable on the second image, where the background is black but doesn't seem uniform in the collage. On the other hand, the color details are well represented.

Reducing the filter size

I tried with the same image with different sizes of the filter to appreciate the differences, with `n = [10, 5, 1]`



As seen, when the filter reach the value 1, the image tranform into a single pixel, loosing all the details of the original image. The perfect result would be achieved in between the filter sizes 5 and 1, for this image size. Maybe if tried with a high quality image (1080p, 4k) the results would be better.

Conformity with results

In all the examples there is a common pattern, the smaller the filter is, the less detail of the original images forming the collage, in the case of `n=1` the image is just a single pixel.

Above all, I am happy with the obtained results, as the images are quite recognizable. If oly I have had more time and resources, I would have tried to "play" with parameters and augmenting the original images dataset to have better results. But as mentioned, I am happy with the results.

In [1]:
```
!jupyter nbconvert --to html --embed-images final_report.ipynb
```

```
[NbConvertApp] Converting notebook final_report.ipynb to html
[NbConvertApp] WARNING | Alternative text is missing on 2 image(s).
[NbConvertApp] Writing 9476163 bytes to final_report.html
```