Practicum: Kleurpaletten

Multimedia - Multimediatechnieken

23 februari 2022

Belangrijk: De verbetering is grotendeels geautomatiseerd, dus lees aandachtig de instructies (omtrent bestandsnamen en andere conventies) zodat je een correct practicum maakt. Respecteer de lengte limiet voor de antwoorden. Het niet respecteren van deze limiet zal leiden tot afgetrokken punten.
Jullie zullen de finale versie van de Python code indienen, samen met de antwoorden op de afzonderlijke tekstuele vragen. Elke tekstueel antwoord komt in een apart bestand met vooraf bepaalde bestandsnaam én structuur. Deze bestanden zijn platte tekst (plain text), en kunnen met eender welke platte tekst verwerker gemaakt worden. Kies zelf uit bijvoorbeeld: vim, Sublime Text, Notepad++, Atom, emacs, ... Maak geen tabellen of lijsten, maar antwoord heel kort zoals aangegeven bij de vragen.

Hoewel de verbetering deels geautomatiseerd is, zal de ingediende code naast functionele correctheid ook **gequoteerd** worden **op stijl en snelheid**. Streef dus naar robuuste, kwalitatieve, en snelle code.

1 Introductie

Kleurpaletten of Color Lookup Tables (LUTs) worden gebruikt om kleurrijke afbeeldingen weer te geven met een kleiner aantal kleuren. Dit was vroeger zeer belangrijk toen beeldschermen of browsers maar een bepaald aantal kleuren konden weergeven. Tegenwoordig is dit minder een probleem, maar het is nog steeds gebruikelijk in het coderen van bvb. GIF bestanden en hardwarematig coderen van texturen in computer graphics (bvb. DXT1).

In dit practicum zullen we een GIF encoder implementeren. Bij afbeeldings- en videoformaten is het gebruikelijk dat enkel het formaat van de bitstroom gestandaardiseerd is. De exacte implementatie van de encoder wordt dus open gelaten, zolang deze resulteert in een bitstroom die voldoet aan de specificaties en dus gedecodeerd kan worden. Hierdoor is er een bepaalde vrijheid tijdens het encoderen waardoor de ene encoder beter kan presteren dan een andere encoder. Met beter presteren bedoelen we dat we een afbeelding met minder bits kunnen opslaan met dezelfde afbeeldingskwaliteit. In dit practicum zullen we tonen hoe we stelselmatig de GIF encoder kunnen verbeteren, startende van een minimale basisversie.

De GIF-encoder bevat drie stappen:

1. het opstellen van een kleurpalet.
2. het kiezen van een LUT-index voor iedere pixel.
3. het coderen van de gekozen LUT-indices.

De derde stap wordt gedaan aan de hand van Lempel–Ziv–Welch (LZW) compressie. LZW is een verliesloze compressiemethode die meerdere symbolen (hier dus LUT-indices) samen codeert als een bitsequentie. De bedoeling is dat vaak voorkomende symboolsequenties gerepresenteerd wordt door een kleiner aantal bits.

2 Opgave

Bibliotheken

Gelieve de volgende Python 3 bibliotheken te installeren indien dit nog niet het geval is (`conda install` als je conda gebruikt, of `pip3 install` indien je niet met conda werkt):

- `numpy` : Proceduraal programmeren van met arrays.
- `scikit-learn` : Machine learning algoritmen, waaruit we k-means clustering zullen gebruiken.
- `Pillow` : Bibliotheek voor het inladen, manipuleren en opslaan van afbeelding. We zullen enkel gebruik maken van de inlaadfunctionaliteit.
- `bitarray` : Handig manipuleren van bit arrays. Wordt gebruikt in de LZW compressie. Let op: aangezien dit package een native (niet-Python) component bevat moet er een stukje gecompileerd worden op Linux en macOS. Voor Linux heb je dan ook `python3-dev` nodig (te installeren via `apt`); alternatief kan je eventueel `sudo apt install python3-bitarray` proberen. Miniconda zal bij gebruik van `conda install` geprecompileerde binaries installeren.

Belangrijk: Gebruik maken van `tqdm` mag. Maak verder geen gebruik van extra bibliotheken.

2.1 De GIF encoder

Voor dit practicum hebben we een eenvoudige GIF encoder geïmplementeerd in Python 3. Deze encoder kan enkel één afbeelding met één kleurenpalet encoderen. De volledige standaard bevat echter meer functionaliteit, zoals meerdere sequentiële afbeeldingen en lokale kleurenpaletten.

De encoder kan je oproepen aan de hand van het volgende commando:

`python3 gifencoder.py -i` *in.png* `-o` *out.gif* `-b` *bits* `[-m` *lut-method* `] [-d` *dither*`]`

Waarbij argumenten in *cursief* door jou dienen ingevuld te worden, en onderdelen tussen [vierkante haken] optioneel zijn:

`-i in.png`	Het pad naar de input afbeelding die we wensen te encoderen met de GIF codec.
`-o out.gif`	Het pad en bestandsnaam van het output GIF-bestand. Zal overschreven worden indien het reeds bestaat.
`-b bits`	Het aantal bits dat kan gebruikt worden per pixel. Met andere woorden, de kleurtabel zal 2^{bits} groot zijn. We noemen dit verder de b-waarde.
`-m method`	De methode om de kleurtabel te construeren.
`-d dither`	Het dithering algoritme. Keuze uit `0`, `1` of `2`.

De laatste twee argumenten zijn dus optioneel, en komen later in het practicum aan bod. Voor meer info, bekijk volgend commando: `python3 gifencoder.py -h`

Vraag 0

Wanneer je bovenstaand commando succesvol uitvoert (met zelf ingevulde argumenten), dan krijg je terug:

```
PSNR: nan dB
```

NaN staat voor *not a number*. PSNR is een (te) simpele wiskundige maat die uitdrukt hoe sterk twee afbeeldingen op elkaar lijken. Hoe hoger de PSNR, des te meer dat de afbeeldingen op elkaar lijken. Wat we dus willen is een hoge PSNR na compressie. Stel twee grijswaarden afbeeldingen A en B met R rijen en K kolommen, dan worden de Mean Squared Error (MSE) en PSNR als volgt berekend:

$$\text{MSE} = \frac{1}{RK}\sum_{k=1}^{K}\sum_{r=1}^{R}(A_{r,k} - B_{r,k})^2 \tag{1}$$

$$\text{PSNR} = 10 \cdot \log_{10}\big(\text{MAX}^2 / \text{ MSE}\big) \tag{2}$$

Hierbij is MAX de maximale waarde die de waarden kunnen bereiken; aangezien de afbeelding in dit practicum 8-bit per kleurkanaal hebben, en we de waarden niet herschalen, is dit dus 255. Wanneer in de limiet twee afbeeldingen identiek zijn, neem je het logaritme van $+\infty$, wat $+\infty$ blijft.

Informatie: Wanneer we echter met kleurafbeeldingen werken, wordt formule 1 of 2 toegepast op elk kleurkanaal en wordt een (gewogen) gemiddelde van de MSE of PSNR genomen. Aangezien het logaritme een niet-lineaire operator is, geeft het gemiddelde berekenen op de MSE en dan overgaan naar PSNR uiteraard niet hetzelfde resultaat wanneer je de PSNR-waarden gebruikt om daarmee een gemiddelde te berekenen:

$$\underbrace{\sum_c w_c \overbrace{10\log_{10}(\text{MAX}^2/\text{MSE}_c)}^{\text{PSNR}_c}}_{???} \neq \overbrace{10\log_{10}(\text{MAX}^2/\underbrace{\sum_c w_c\,\text{MSE}_c}_{\text{MSE}})}^{\text{PSNR}},$$

waarbij w_c het gewicht van kleurkanaal c voorstelt, met $\sum_c w_c = 1$. Meestal krijgt blauw heel wat minder gewicht in het gemiddelde aangezien we er als mens minder gevoelig voor zijn (zie hoofdstuk over kleuren en volgend practicum). Merk op dat dit louter een wiskundige bepaling is dat niet volledig overeenkomt met het menselijk visueel systeem. Het is echter wel snel en gemakkelijk te berekenen.

De NaN die je terug krijgt komt van een nog-niet-geïmplementeerde PSNR-functie. Implementeer PSNR (met gelijke gewichten $w_R = w_G = w_B$, door uitmiddeling van MSE per kanaal) in `gifencoder.py` in volgende functie:

```
def calculate_psnr(img_A, img_B):
```

Hierbij zijn `img_A` en `img_B` al numpy arrays. Maak hierbij gebruik van `numpy`! Python loops (`for`, `while`) zijn niet toegestaan, aangezien Python onwaarschijnlijk traag is (zoals gezien in

het introductiepracticum). Numpy is relatief gezien snel, aangezien bijna alle functionaliteit geïmplementeerd is in C.

Hint: Bekijk https://scipy-lectures.org/intro/numpy/operations.html#basic-reductions en raadpleeg de laatste API documentatie van numpy voor informatie. Bekijk ook zeker de resulterende GIF bestanden.

Hint: Denk goed na over de datatypes (byte, unsigned byte, int, unsigned int, float, double, ...) in Numpy en de bijhorende berekeningen die je uitvoert. Zorg dat wat je programmeert ook weldegelijk het juiste antwoord produceert. Denk aan precisie en ranges van getaltypes. Test bijvoorbeeld of jouw implementatie symmetrisch is: `psnr(A,B) == psnr(B,A)`. Dit kan je bijvoorbeeld toevoegen als controle in de `main()`-functie.

Hint: Bereken met de hand de PSNR voor het worst-case scenario. Denk dus ook na wat het worst case scenario is voor het geval van afbeeldingen. Schrijf eventueel zelf een Python-functie die uw PSNR implementatie test met verschillende inputs.

Hint: De verwachtingswaarde van de PSNR tussen twee willekeurige afbeeldingen is ongeveer 7.747dB.

Belangrijk: Python-loops zijn niet toegestaan! Vermijd dat je de logica herhaalt voor het R-, G-, en B-kanaal (denk wiskundig!).

Vraag 1

Encodeer een `kodim`-testafbeelding (kies zelf welke!) met b-waardes: 2, 5, en 8, zonder optionele argumenten. Herhaal de encodering driemaal per b-waarde. Wat kan je zeggen over de kwaliteit wanneer b groter wordt? Wat valt er je op als je de encodering meerdere malen uitvoert met éénzelfde b-waarde? Hoe komt dit?

Belangrijk: Gegeven volgende structuur:

Hoe groter b, hoe *(A)*.
Uitvoeren van meerdere encoderingen met dezelfde b levert *(B)* wat komt door *(C)*.

Formuleer het antwoord in `vraag-1.md` door jouw invulling voor *(A)*, *(B)*, en *(C)* elk op één lijn van het antwoordbestand te noteren. Zorg er dus voor dat deze placeholders kort en grammaticaal correct zijn zonder extra zinnen te introduceren. Jouw finale bestand telt dus 3 lijnen.

2.2 Kleurenpalet opstellen

Vraag 2.1: Grijswaarden methode

Implementeer volgende functie waarbij je 2^b tinten grijs als kleurtabel opstelt (lopende van volledig zwart naar volledig wit). Besteed ook aandacht aan de gegeven *"docstring"*.

```
def make_grayscale_color_table(color_table_size):
```

Zoek zelf uit hoe je deze methode kan aanroepen via de commandolijn om deze zo te kunnen testen. Hier hoef je niets over te schrijven: de code zal gewoon gequoteerd worden. Je hebt deze techniek later nodig in het practicum.

Vraag 2.2: Random sampling

We proberen een beter kleurenpalet op te stellen aan de hand kleuren die effectief voorkomen in de afbeelding. Schrijf de code voor de volgende functie waarbij 2^b kleuren willekeurig gesampled worden uit de afbeelding:

```
def make_random_sample_color_table(color_table_size, img):
```

Codeer afbeelding met dezelfde b-waarden (2, 5, en 8). Doe de encodering opnieuw enkele malen per b-waarde en bekijk de resultaten. Wat zijn de bevindingen over de PSNR en de kwaliteit? Een oplijsting van de PSNR-waarden is nutteloos. Geef kwalitatieve inzichten: observeer en probeer te verklaren.

Belangrijk: Formuleer uw antwoord in: `vraag-2-2.md`
Verder wordt de code getest en gequoteerd.

Vraag 2.3: Median-cut algoritme

Typisch wordt voor de gif-encoder het *median-cut* algoritme voorgesteld, omdat het eenvoudig en snel is, terwijl het aanvaardbare kwaliteit levert. In het median-cut algoritme wordt de kleurenruimte steeds verder onderverdeeld in kleinere blokken tot dat er zoveel blokken zijn als dat er plaatsen zijn in het kleurenpalet. Het algoritme werkt min of meer als volgt:

1. Stop alle kleuren van de afbeelding in één blok.
2. Zolang er niet genoeg blokken zijn:
   (a) Zoek het blok *met* kleurkanaal (R, G, of B) met de grootste variantie,
   (b) Bepaal de waarde van de mediaan voor dit kanaal van de gekozen blok,
   (c) Splits het blok langsheen het gekozen kleurkanaal op in twee blokken op basis van deze mediaan.
3. Kies op basis van de geconstrueerde blokken representant kleuren.

Subvraag a: Implementeer het median-cut algoritme voor het opstellen van het kleurenpalet. Het algoritme dient niet-recursief geïmplementeerd te worden. Er is niet noodzakelijk één juist antwoord: er kunnen keuzes gemaakt worden. Let wel op enkele logische voorwaarden zoals:

1. Een afbeelding met 2^b unieke kleuren moet perfect worden gereconstureerd wanneer de kleurentabel dus ook 2^b elementen bevat. Je kan hiervoor `4shadesofgray_0.png` (b=2),

`4shadesofgray_1.png` (b=2), `geel.png` (b=1), en `rood.png` (b=1) gebruiken om dit te testen.

2. Elke unieke kleur komt in maximaal één blok terecht.

Vul het algoritme aan in:

```
def make_median_cut_color_table(color_table_size, img):
```

Verklaar welke ontwerpkeuzes u gemaakt heeft (in één korte zin per puntje):

- Wat gebeurt er als je een blok met maar één unieke kleurwaarde splitst?
- Met welke kleur stel je een blok voor?
- In welk blok komt de mediaan waarde zelf terecht?

Belangrijk: Formuleer uw antwoord in: `vraag-2-3a.md`, door middel van enkele woorden of één **korte** zin per puntje. Respecteer deze limiet. Zet het antwoord per puntje op één lijn in de Markdown bestand. Uiteindelijk zal uw Markdown bestand dus 3 korte lijnen tekst bevatten.

Subvraag b: Codeer de afbeeldingen `test_noise_bin.png` en `test_bw.png` met $b = 1$. Beide afbeeldingen hebben evenveel pixels en evenveel unieke kleuren, namelijk zwart en wit. Waarom zijn deze resulterende GIFs verschillend in bestandsgrootte?

Hint: PSNR waardes zijn hier $+\infty$ aangezien je perfecte codering hebt (twee kleuren in de afbeelding en twee kleuren in de kleurentabel).

Belangrijk: Formuleer uw bevindingen in: `vraag-2-3b.md`, door middel van **één of maximaal twee korte** zinnen. Wees dus beknopt en beperk je tot de essentie. Respecteer deze limiet.

Vraag 2.4: Vector Quantization

Heel gelijkaardig aan kleurpaletten opbouwen is het concept van *Vector Quantization*. In plaats van een sequentie van scalaire getallen te kwantiseren, kwantiseren we een sequentie van vectoren. In ons geval willen we dus een lijst van (r,g,b)-vectoren kwantiseren. De bedoeling is om de tabel te vullen met (r,g,b)-vectoren die representatief zijn voor alle kleuren in de afbeelding. Idealiter willen we dat iedere waarde in de tabel een gelijk aantal originele pixels representeert.

Een goede manier om dit te doen is via de cluster methode *k-means*, een relatief simpele machine learning methode waarvan we de details achterwege laten. Je geeft aan hoeveel clusters je wilt hebben. Aan de hand van de `fit`-functie leert het algoritme wat een goede clusterverdeling is. Iedere cluster heeft dan een center-waarde.

```
from sklearn import cluster
kmeans = cluster.MiniBatchKMeans(n_clusters, n_init=4)
kmeans.fit(data)
```

Implementeer aan de hand van bovenstaande informatie een k-means kleurentabel als de volgende functie:

```
def make_kmeans_color_table(color_table_size, img):
```

Werkt dit algoritme beter dan median-cut? Was dit te verwachten?

Informatie: Je hoeft niets op te schrijven. De code wordt gewoon getest.

2.3 Dithering

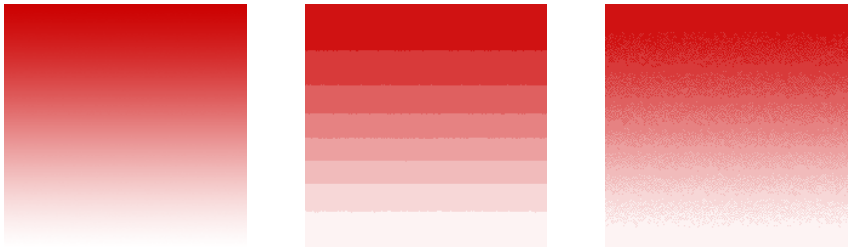
De eerste afbeelding in Fig. 1 toont David van Michelangelo in 256 grijswaarden, de tweede toont slechts 2 grijswaarden (zwart en wit). Merkwaardig genoeg toont de derde afbeelding ook maar twee grijswaarden. Enkel is in de derde afbeelding niet telkens gekozen voor het dichtstbijzijnde kleur in kleurenpalet. In de plaats wordt nu en dan gekozen voor een ander kleur.

(a) 256 grijswaarden (b) Twee grijswaarden (c) Dithered

Figuur 1: https://en.wikipedia.org/wiki/Dither

Er bestaat een groep aan methodes die deze patronen bepalen om de illusie te wekken dat er tussenliggende kleuren zijn, nl. *dithering*. In de lessen hebben je *ordered dithering* gezien, dewelke gebruik maakt van een dither matrix. Bij *pattern dithers* worden de lengte en breedte van de afbeelding vergroot met een factor k en wordt iedere pixel dan voorgesteld door een patroon van $k \times k$ pixels. Hierbij is er dus duidelijk een trade-off tussen spatiale resolutie en kleurenresolutie. Er bestaan ook andere dither-methoden. Bijvoorbeeld, aan de hand van ruistoevoeging wordt de gemaakte kwantisatiefout gerandomiseerd. M.a.w. er wordt niet telkens voor de dichtstbijzijnde kleur gekozen. Dit zorgt er voor dat grote structurele artefacten minder zichtbaar worden, zoals in het Fig. 1 en 2.

De middelste afbeelding van Fig. 2 toont een veelvoorkomend coderingsartefact, nl. *color banding*. Dit ontstaat doordat er een discrete sprong is waarbij consistent voor een andere dichtstbijzijnde kleur wordt gekozen. De rechtse afbeelding toont hoe dithering hier de indruk geeft van een veel vloeiendere gradiënt. Hoewel deze methode de spatiale resolutie niet verhoogt, kan deze zelfde methode ook toegepast worden in combinatie met het verhogen van de resolutie. Het resultaat zal er dan nog beter uitzien, aangezien onze ogen de kleuren dan beter kunnen mengen omdat de ruis fijner is. In dit practicum houden we de resolutie gelijk.



(a) 256 kleuren (b) 8 kleuren (c) Dithered

Figuur 2: https://www.lifewire.com/dithering-gif-images-4122770

Vraag 3.1: Floyd-Steinberg diffusie

In onze huidige versie van de GIF-encoder wordt steevast voor de meest naburige kleurwaarde in het kleurenpalet gekozen. Dit wilt zeggen dat er consistent naar dezelfde "afronding" wordt toegepast.

In deze opgave zullen we hierop voortbouwen door een dithering methode te implementeren die de afbeeldingsgrootte niet vergroot. Diffusie dithering is een methode die kwantisatiefout naar de naburige pixels doorschuift. Het idee is dat eenmaal een pixel naar beneden is afgerond, dat je dan de kans wilt verhogen dat de volgende pixel eens naar boven wordt afgerond.

Het Floyd-Steinberg diffusie-dithering algoritme overloopt de afbeelding rij per rij, en past voor elke pixel in die rij (van links naar rechts) de dithering stap toe. Deze stap bestaat uit kwantisatie en dan doorschuiven van de kwantisatiefout naar naburige pixels. Deze fout wordt op een gewogen manier verdeeld over de naburige pixels. Deze gewichten worden voorgesteld in de "dithermatrix" (of *kernel*). Voor het Floyd-Steinberg algoritme is de dithermatrix als volgt:

$$\frac{1}{16}\begin{bmatrix} - & * & 7 \\ 3 & 5 & 1 \end{bmatrix}. \tag{3}$$

Hierin komt het sterretje ($*$) overeen met de positie van de huidige pixel wiens kwantisatiefout zal verspreid worden. Merk op dat de fout niet kan doorschuiven naar reeds-bezochte pixels, of de huidige pixel zelf. Uiteraard is de som van de gewichten 1, zodat de volledige kwantisatiefout verdeeld wordt.

Implementeer diffusie-dithering in de functie:

```
def transform_image_to_indices_diffusion_dithering(img,
                                                    color_table,
                                                    dither_matrix,
                                                    anchor_col):
```

Gebruik van Python-loops is toegestaan. Deze functie krijgt als argument de dither matrix en de ankerpositie (het sterretje) van de dither matrix meegegeven. Deze functie wordt al opgeroepen voor jou in volgende functie:

```
def transform_image_to_indices_dithering_1(img, color_table):
```

Hint: Let op dat in de volgende opgave ook een tweede functie uw dithering algoritme zal aanroepen, maar een grotere dithermatrix gebruikt. Zorg er dus voor dat uw implementatie overweg kan met verschillende groottes van de dithermatrix.

Hint: De afbeeldingen zitten in *row-major order* in het geheugen. Dit betekent dat de afbeelding rij per rij in het geheugen zit. Twee horizontaal naburige pixels zitten dus naast elkaar in het geheugen. Twee verticaal naburige pixels zitten dus ver uit elkaar in het geheugen. Numpy indexeert ook in row-major volgorde: de eerste index is de rij, de tweede is de kolom.

Vraag 3.2: Eerste tests

Codeer jouw kozen testafbeelding én `david.png` met enkele b waarden. Vergelijk het `median-cut` algoritme met de `grayscale` kleurentabel voor één bit. Welke 1-bit kleurentabel geeft het visueel mooiste resultaat, en hoe komt dit?

Hint: Wanneer je `david.png` encodeert met dithering gebruik makend van een 1-bit median-cut kleurtabel, is de afbeelding rechts *niet* het juiste resultaat. Dithering behoudt normaal gezien de details, dewelke hier volledig verdwenen zijn.

Belangrijk: Gegeven volgende structuur:

De visueel mooiste resultaten worden bekomen met de *(A)* kleurentabel.
Dit komt door het feit dat *(B)*.

Formuleer het antwoord in `vraag-3-2.md` door jouw invulling voor *(A)* en *(B)* elk op één lijn van het antwoordbestand te noteren. Zorg er dus voor dat deze placeholders kort en grammaticaal correct zijn zonder extra zinnen te introduceren. Jouw finale bestand telt dus 2 lijnen. Je noteert dus enkel jouw invulling voor *(A)* en *(B)*, zonder de gegeven structuur zelf over te nemen in het antwoord bestand.

Vraag 3.3: Bestandsgroottes

Vergelijk de bestandsgroottes met en zonder dithering. Waarom zijn GIF bestanden met dithering typisch groter? Verklaar kort.

Belangrijk: Formuleer jouw antwoord in `vraag-3-3.md` door middel van **één of maximum twee heel korte zinnen**. Respecteer deze limiet.

Vraag 3.4: Kwaliteit

Vergelijk de kwaliteit PSNR waarden met en zonder dithering. Waarom is de PSNR van geditherde afbeeldingen lager, hoewel het resultaat er visueel toch beter uit ziet?

Belangrijk: Formuleer jouw antwoord in `vraag-3-4.md` door middel van **één of maximum twee heel korte zinnen**. Respecteer deze limiet.

Vraag 3.5: Minimized Averaged Error

Het "minimized average error" diffusie dithering methode werkt op dezelfde manier. Enkel wordt de kwantisatiefout verder verspreid. De spreidingsmatrix ziet er namelijk zo uit:

$$\frac{1}{48}\begin{bmatrix} - & - & * & 7 & 5 \\ 3 & 5 & 7 & 5 & 3 \\ 1 & 3 & 5 & 3 & 1 \end{bmatrix}$$

Implementeer deze techniek door jouw reeds-bestaande dither-implementatie aan te roepen, maar met deze nieuwe kernel in de functie:

```
def transform_image_to_indices_dithering_2(img, color_table):
```

Vergelijk de twee methoden visueel voor gelijke b waardes. Gebruik opnieuw dezelfde twee afbeeldingen: de zelf gekozen afbeelding en `david.png`. Welke methode verkies je?

Belangrijk: Formuleer jouw antwoord in: `vraag-3-5.md`. Als eerste lijn in het antwoordbestand noteer je ofwel '`floyd`' ofwel '`mae`'. Op de volgende regel schrijf je één korte zin waarom je deze techniek verkiest.

3 In te dienen bestanden

Pak jouw oplossing in in een ZIP-file, en stuur jouw ZIP door via de Opdracht op Ufora, met de bestandsnaam:

`practicum_kleurpaletten.zip`

Bij deze bestandsnaam vul je jouw naam in. In deze ZIP zitten jouw Python files (`gifencoder.py` en `lzw.py`) samen met de antwoord bestanden (`*.md`). Stuur dus **geen afbeeldingen** mee.

Inpakken in ZIP-file kan met (bijvoorbeeld met Bash, Zshell, etc...):

```
zip practicum_kleurpaletten.zip *.md *.py
```