2012

[OSNOVE JPEG KOMPRESIJE]

Na ovoj vježbi studenti će obraditi principe JPEG kompresije, sa naglaskom na diskretnu kosinusnu transformaciju (DCT) te će realizovati vlastiti kod za DCT koristeći Matlab.

Ime:
Prezime:
Broj indeksa:

UVODNE NAPOMENE:

- Preporuka je da se na vježbe donesu sve stranice isprintane.
- Izvještaj sa vježbe što uključuje popunjene stranice na predviđenim mjestima nakon naknadne obrade kod kuće treba dostaviti do datuma koji će biti naknadno definiran
- Izvještaji s vježbe su jedan od elemenata koji se analizira i na završnom ispitu .
- Pošto je ovaj tutorijal malo zahtjevniji, biće dovoljno da uradite prva tri poglavlja.

1. Diskretna kosinusna transformacija (DCT) u JPEG kompresiji

Diskretna kosinusna transformacija (DCT) je transformacija srodna Fourierovoj koja ima niz korisnih osobina koje je čine pogodnom za primjenu u kompresiji signala sa gubicima (lossy). Kao i Fourierova transformacija, DCT izražava neku primljenu funkciju kao sumu sinusoida različitih frekvencija i amplituda. DCT je diskretna transformacija što znači da se funkcija transformiše u nekom konačnom skupu tačaka umjesto na kompletnom domenu. Za razliku od Fouriera transformacije, DCT koristi samo kosinusne funkcije. Postoje četiri vrste DCT transformacije, a najpoznatija je DCT – II koja je našla svoju primjenu u mnogim multimedijalnim sistemima, njena najpoznatija primjena jeste u JPEG kompresiji.

Na proteklim tutorijalima, rastersku grafiku smo posmatrali kao 2D tabelu u kojoj svako polje predstavlja jedan piksel. U slučaju slike u nijansama sive, ovu tabelu možemo matematički opisati i kao funkciju dvije pozitivne cjelobrojne promjenljive f(x,y) čija vrijednost u nekoj tački odgovara intenzitetu svjetlosti. Ako je slika u boji, npr. koristeći RGB model boje, DCT moramo primijeniti odvojeno na svaku komponentu boje, pošto komponenta odgovara "intenzitetu". Ako primijenimo DCT na neku drugu vrstu vrijednosti npr. indeks boje u paleti dobićemo nezadovoljavajuće rezultate.

Kao što je već spomenuto, postoji više vrsta DCT-a. Najčešće korištena je DCT-II koja se naziva i samo DCT. DCT-II (ponekad i 2D DCT) je opisana formulom:

$$F(u,v) = 2\frac{C(u)\cdot C(v)}{\sqrt{M\cdot N}}\left[\sum_{x=0}^{M-1}\sum_{y=0}^{N-1} f(x,y)\cdot cos\frac{(2x+1)\cdot u\cdot \pi}{2M}\cdot \cos\frac{(2y+1)\cdot v\cdot \pi}{2N}\right] \dots (1)$$

gdje su *C(u)* i *C(v)* konstante koje su jednake

$$C(u) = C(v) = \begin{Bmatrix} \frac{1}{\sqrt{2}},\ za\ (u,v=0) \\ 1,\ za\ u=v\neq 0 \end{Bmatrix}$$

U gornjem izrazu veličine M i N određene su dimenzijama slike (dimenzije slike kao što ćemo vidjeti su M=8 i N=8 tj. *8x8 blok*), dok je rezultat diskretne kosinusne transformacije matrica dimenzija *u,v*.

Računarski najzahtjevniji dio ove formule je računanje kosinusa (vrijednosti korijena su u biti konstante). Da bi se DCT završio u nekom razumnom roku, mi našu rastersku sliku dijelimo na niz sitnih blokova i onda izračunavamo DCT funkciju *F(u,v)* za svaki od tih blokova. U JPEG standardu precizirano je da dimenzija tog bloka budu 8x8, što znači da je M=N=8. U teoriji veći blok bi trebao rezultirati kvalitetnijom kompresijom slike, ali i sporijim izvršavanjem, pa je 8x8 kompromis koji je usvojen kao dio JPEG standarda. Osim toga, iz izraza za *diskretnu kosinusnu transformaciju* jasno je vidljivo da je za svaki element

F(u,v) potrebno proći kroz sve elemente bloka, tj. izvršiti 64 operacije (minimalno) za jedan element. Kako blok ima 64 elementa, slijedi da je za jedan blok potrebno izvršiti minimalno 4096 operacija. Da budemo precizni: ***191 operacija zbrajanja, 513 operacija množenja, računanje 128 puta cos (kosinusne) funkcije te 128 operacija dijeljenja*** što je i za današnje računare puno (sva ova izračunavanja su potrebna za dobivanje *samo jednog* FDCT elementa). Naravno, postoje puno brže metode koje se svode na 5 operacija množenja i 29 operacija zbrajanja. Možda je sada i jasnije zašto je odabrana veličina bloka 8x8 piksela.

Koeficijenti DCT transformacije sadržavaju informaciju o frekventnom sadržaju slike. Koeficijent *F(0,0)* je ***istosmjerni*** ili ***DC koeficijent*** i on je *jednak srednjoj vrijednosti svih ostalih koeficijenata pomnoženoj sa 8* što se veoma lako može dokazati.

$$F(0,0) = 2\frac{C(0)\cdot C(0)}{7}\left[\sum_{x=0}^{7}\sum_{y=0}^{7} f(x,y)\cdot cos\frac{(2x+1)\cdot 0\cdot \pi}{2M}\cdot \cos\frac{(2y+1)\cdot 0\cdot \pi}{2N}\right] = \frac{1}{8}\left[\sum_{x=0}^{7}\sum_{y=0}^{7} f(x,y)\right] \dots (2)$$

Srednju vrijednost možemo izračunati prema standardnoj formuli kao sumu svih elemenata podijeljena sa brojem elemenata. Kako u našem slučaju imamo 64 elementa imamo sljedeću formulu:

$$S_{vrijednost} = \frac{1}{64}\left[\sum_{x=0}^{7}\sum_{y=0}^{7} f(x,y)\right] \dots (3)$$

Formulu (3) pomnožimo sa 64 i uvrstimo u izraz za *F(0,0),* dobijamo

$$F(0,0) = 8\cdot S_{vrijednost}$$

DC koeficijent sadrži najveći dio informacije o slici i najvažniji je za rekonstrukciju slike. Preostala 63 koeficijenta nazivaju se ***izmjeničnim*** ili ***AC koeficijentima***. AC koeficijenti sadržavaju informaciju o prostornim frekvencijama u bloku odnosno u slici, pri tome, AC koeficijenti koji se nalaze u okolini DC koeficijenta odgovaraju nižim prostornim frekvencijama, dok AC koeficijenti koji se smješteni prema donjem desnom uglu 8x8 matrice, opisuju veće prostorne frekvencije. DCT transformacija na temelju frekvencijske analize nad elementima bloka, provodi preraspodjelu energije koju nosi pojedini element u bloku. Najveći dio energije koncentriran je u DC koeficijentu te nisko frekvencijskim AC koeficijentima koji ga okružuju. AC koeficijenti nižih prostornih frekvencija nose više korisne informacije nego oni viših frekvencija. Ovakav način koncentracije energije u skladu je sa svojstvom ljudskog vizualnog sistema. Obično su vrijednosti AC koeficijenata za velike frekvencije po apsolutnoj vrijednosti manje od vrijednosti AC koeficijenata u okolini DC koeficijenta, što će se pokazati korisno u procesu kvantizacije, odnosno u procesu *RLC* (*Run Length Coding*) kodiranja.

Ako smo prilikom kompresije slike koristili DCT, logično je onda da prilikom dekompresije slike koristimo *Inverznu Diskretnu Kosinusnu Transformaciju* (*IDCT*) *F(u,v),*

pomoću koje ćemo dobiti polaznu sliku *f(i,j)* uz eventualno određene gubitke. Inverzna diskretna kosinusna transformacija (IDCT) u svom općem obliku glasi:

$$\tilde{f}(x,y) = 2\frac{C(u)\cdot C(v)}{\sqrt{M\cdot N}}\left[\sum_{x=0}^{M-1}\sum_{y=0}^{N-1} F(u,v)\cdot cos\frac{(2x+1)\cdot u\cdot \pi}{2M}\cdot \cos\frac{(2y+1)\cdot v\cdot \pi}{2N}\right] \ldots (4)$$

gdje su *i,j=0,1,...,7*. Razlog za tildu iznad naziva funkcije *f* je što ona nije ekvivalentna polaznoj funkciji *f*(i,j) jer je moguće došlo do određenih gubitaka.

Ovom prilikom ćemo naglasiti još jednom da DCT radi sa cjelobrojnim vrijednostima intenziteta svjetlosti koje su centrirane oko nule. Podsjetimo se da cjelobrojni RGB format u *Matlabu* pridružuje svakoj komponenti boje vrijednosti u opsegu 0-255. Ovdje ćemo navesti jedan implementacijski detalj, kojeg JPEG standard i nalaže. Prije nego izvršimo DCT nad bilo kojim 8x8 blokom slike, potrebno je da od svakog piksela svih komponenti slike (Y,C_b C_r) oduzmemo vrijednost 128, kako bismo kreirali sliku čija je srednja vrijednost pikselâ jednaka nuli (*zero – mean image*). Logično je sada da pri obrnutom procesu treba izvršiti dodavanje vrijednosti 128. O modelu boja YC_bC_r ćemo govoriti nešto kasnije.

Zadatak 1.1. *U Matabu učitajte kolor sliku po želji (npr. neku od slika koje dolaze uz image toolbox) u matricu koristeći funkciju imread(). Zatim napravite Matlab funkciju pod nazivom dct() koja iz primljene matrice izdvaja prvi blok dimenzija 8x8 i ispisuje ga na ekranu.*

Matlab kod upišite u prostor ispod.

Zadatak 1.2. *Prepravite ovu funkciju tako da ispisuje SVE blokove dimenzija 8x8 u matrici. Imajte na umu da dimenzije slike ne moraju biti djeljive sa 8! Preostale piksele u rubnim blokovima popunite nulama.*

NAPOMENA: JPEG standard zahtijeva da matrica F(u,v) bude djeljiva sa 8, što znači da njene dimenzije neće odgovarati originalnim dimenzijama slike. U tu svrhu kreirajte dvije promjenjive, unutar kojih ćete pohraniti dimenzije originalne slike. Također, JPEG standard zahtjeva da dodani redovi i kolone ne sadrže nule, nego vrijednosti posljednje kolone i reda originalne slike, ali radi jednostavnosti vi dodane redove i kolone popunite nulama.

Matlab kod upišite u prostor ispod.

Zadatak 1.3. *Prepravite napravljenu funkciju tako da za svaki blok dimenzija 8x8 (nazovimo ga f) izračunava ekvivalentan DCT blok F prema formuli (1) za DCT na bloku 8x8. Taj blok*

treba smjestiti u neku matricu rezultata koju će funkcija dct() vraćati. Matrica rezultata treba biti istih dimenzija kao primljena matrica.

Savjet: ne morate se posebno brinuti za pred-izračunavanje vrijednosti kosinusa i $\sqrt{2}/2$ pošto će *Matlab* vrlo efikasno cachirati te izračunate vrijednosti. No prilikom implementacije algoritma u C++ morate voditi računa o tome.

Matlab kod upišite u prostor ispod.

Zadatak 1.4. *Na osnovu ovako napravljene funkcije dct() napravite inverznu funkciju idct() koristeći formulu (4) za IDCT. Funkcije testirajte tako što ćete nad učitanom slikom pozvati dct() pa idct() i tako dobijenu sliku prikazati:*

```
A = imread('slika.png');
B = dct(A);
C = idct(B);
imshow(A); figure; imshow(C);
```

Matlab kod upišite u prostor ispod.

Ako ste sve odradili kako treba, trebali biste dobiti sliku praktično identičnu polaznoj pošto još ustvari nismo došli do lossy dijela algoritma.

NISTE DOBILI OČEKIVANI REZULTAT?

Slijedi spisak stvari koje biste trebali provjeriti:

1. Ako ste učitali sliku čiji je format paleta, DCT neće raditi jer DCT očekuje da primljene vrijednosti odgovaraju intenzitetu svjetlosti u datoj tački. Detaljnije je objašnjeno u Lab vježbi 5.
2. Ako je u pitanju kolor RGB slika, kod morate izvršiti zasebno za svaku od komponenti boje.
3. DCT očekuje da su primljene vrijednosti cijeli brojevi. Dakle format vaše slike bi trebao biti <XxYx3 uint8>. Ako ovo nije format slike, koristite neku prikladnu funkciju za konverziju (pogledati Lab vježbu 5). Uz Matlab dolazi jedna pogodna kolor RGB slika pod imenom autumn.tif.

4. Ako ste slijedili savjet na stranici 3 da se prije početka DCT-a od komponenti oduzme vrijednost 128, obratite pažnju da prije toga morate konvertovati uint8 u neki drugi tip. Razlog je što će se vrijednosti manje od nule zaokružiti na nulu (slovo *u* u uint8 označava unsigned).
5. Množenje sa kosinusom će izvršiti automatsku konverziju rezultujućeg tipa u double, što se odnosi i na funkciju idct(). Dakle rezultujuća matrica će biti tipa <XxYx3 double> sa vrijednostima u opsegu 0-255 što ne predstavlja validnu RGB sliku. Morate na kraju funkcije idct() izvršiti konverziju nazad u tip uint8.
6. Ako sa donje i desne strane slike vidite crni okvir, to je zato što ste u DCT fazi popunili nedostajuće vrijednosti blokova nulama, ali ih niste izbacili iz matrice koju funkcija vraća kao rezultat. U obje funkcjie koje ste napisali rezultujuća matrica treba biti tačno istih dimenzija kao primljena matrica.

Kvantizacija i "quality" parametar

Do sada urađene funkcije za DCT i IDCT nisu unijele skoro nikakve gubitke u sliku zato što je konvertovana slika čuvana u matrici tipa double (64 bita), pa je greška uzrokovana samo zaokruživanjem tek na 15. decimali ili dalje. U realnosti, ovakva matrica zauzima 8 puta više memorije od polazne slike u kojoj je svaka komponenta boje opisana sa 8 bita. Nakon što matricu svedemo na format *uint8*, dobit ćemo određeni gubitak kvalitete.

Analiza ljudske percepcije svjetlosti pokazala je da spektar intenziteta svjetlosti nije linearan u odnosu na ljudsku sposobnost da uoči razliku između dva piksela. Drugim riječima, na grayscale slici čovjek će lakše uočiti razliku između tačaka kodiranih sa 10 i 11, nego tačaka kodiranih sa 249 i 250. U JPEG algoritmu pokušala se iskoristiti ova osobina ljudskog vida tako što će se veći značaj dati promjenama intenziteta u tamnijem dijelu spektra nego u svjetlijem. Ova nelinearna zavisnost matematički je izražena posebnom, ***matricom kvantizacije***. Riječ je o kvadratnoj matrici dimenzija 8x8 čiji elementi se nazivaju *kvantizatori*. Vrijednosti kvantizatora su unaprijed tačno definirane veličine, a određene su u skladu sa svojstvima ljudskog vizualnog sistema. Vrijednosti kvantizatora u matrici rastu od lijevog gornjeg ugla prema donjem desnom uglu, dakle obrnuto od raspodjele vrijednosti DCT koeficijenata baš kako je to prikazano pomoću sljedeće tabele.

$$
Q = \begin{bmatrix}
16 & 11 & 10 & 16 & 24 & 40 & 51 & 61 \\
12 & 12 & 14 & 19 & 26 & 58 & 60 & 55 \\
14 & 13 & 16 & 24 & 40 & 57 & 69 & 56 \\
14 & 17 & 22 & 29 & 51 & 87 & 80 & 62 \\
18 & 22 & 37 & 56 & 68 & 109 & 103 & 77 \\
24 & 35 & 55 & 64 & 81 & 104 & 113 & 92 \\
49 & 64 & 78 & 87 & 103 & 121 & 120 & 101 \\
72 & 92 & 95 & 98 & 112 & 100 & 103 & 99
\end{bmatrix} \ldots (4)
$$

NAPOMENA: Ova tablica se često ne koristi u realnim aplikacijama tj. obično je skalirana (npr. izvršeno je cjelobrojno dijeljenje sa 10). Razlog tome jeste dobivanje kvalitetnije slike, a vidjet ćemo iz kojeg razloga. Ovu tabelu sad za sad možete kopirati u vaš Matlab kod.

Proces kvantizacije je definiran, kao cjelobrojno dijeljenje definirano formulom:

$$K(u,v) = round(\frac{F(u,v)}{Q(u,v)})$$

gdje je *Q(u,v)* vrijednost elementa kvantizacijske tablice, čiji su elementi, kao što smo to ranije naveli, unaprijed zadane konstantne veličine. Matrica *K(u,v)* predstavlja rezultat kvantizacije. Kvantizacija se provodi kako bi se smanjio broj bita koji je potreban za predstavljanje koeficijenata dobivenih diskretnom kosinusnom transformacijom. ***Kvantizacija je korak unutar kojeg dolazi do nepovratnog gubitka podataka.*** Jasno je da do gubitka podataka dolazi zbog cjelobrojnog dijeljenja koje odbacuje sve cifre iza decimalne tačke.

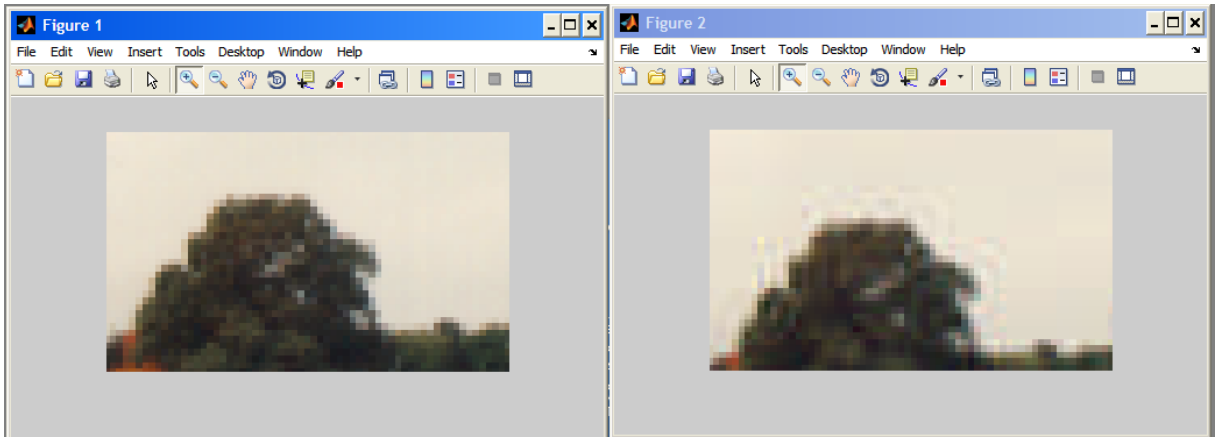
Kao što smo već spomenuli, navedena tabela *Q* se ne koristi u praktičnoj implementaciji JPEG filtera, iz razloga što su vrijednosti u matrici velike, što kao posljedicu može imati da su svi AC koeficijenti 8x8 bloka nakon cjelobrojnog dijeljenja jednaki 0, što u slučaju da želimo kvalitetniju sliku, nije poželjno. Stoga bi prije procesa kvantizacije bilo korisno da korisniku možemo omogućiti zadavanje kvalitete slike. To možemo učiniti na način da kvantizacijsku tablicu pomnožimo sa konstantom $m_{\text{koeficijent}}$ koja ima sljedeće osobine:

$$\left\{\begin{matrix} 0 < m_{koeficijent} < 1 \ \ bolja\ kvaliteta\ slike \\ 1 \leq m_{koeficijet}\ lošija\ kvaliteta\ slike \end{matrix}\right\} \ldots (5)$$

Ovisno od vrijednosti kvantizatora u matrici *Q*, subjektivno, slika može izgledati gore od početne, ali će zato veličina rezultujuće datoteke nakon kompresije biti minimalna koja se može postići JPEG kompresijom. Nekada autori slika odluče da im nije potrebna tako mala datoteka, nego bi radije zadržali kvalitetu slike uz nešto veću datoteku, zbog čega se uvodi upravo spomenuti koeficijent kvalitete *m* sa kojim se matrica Q množi. Logično slijedi da ako želite ispravno dekompresovati sliku, isti koeficijent morate koristiti i u DCT i u IDCT fazi. Koeficijent *m* ima vrijednosti u opsegu prikazanom izrazom (5), pri čemu vrijednost 0 nije dozvoljena (rezultira crnom slikom), manje vrijednosti u pravilu rezultiraju slikom veće

kvalitete i većom rezultirajućom datotekom, a vrijednosti veće od 1 nemaju smisla (dobiće se slika izuzetno loše kvalitete ali bez daljnjeg smanjenja veličine datoteke).

Zadatak 2.1. *Dodajte matricu kvantizacije u vaše funkcije dct() i idct() na način kako je opisano u prethodnom tekstu. Sada usporedite sliku prije i poslije kvantizacije. Da biste jasnije vidjeli efekat, zumirajte dio slike na kojem ima više detalja. Trebali biste vidjeti dobro poznate JPEG artefakte kao na slici*:



Zadatak 2.2. *U obje funkcije dodajte parametar kvaliteta. Ovaj parametar se koristi tako što se jednostavno matrica Q pomnoži s njim. Prilikom testiranja pazite da koristite istu vrijednost kod poziva obje funkcije.*

Matlab kod dobivenih funkcija upišite u prostor ispod.

Chroma subsampling

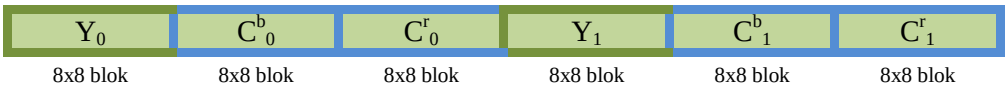
Model RGB nije pogodan za kompresiju zato što ljudski vid ima različitu osjetljivost na RGB komponente, a nju je komplicirano opisati matematički. Prilikom JPEG kompresije, najprije želimo da transformišemo ovaj model u neki drugi model u kojem ćemo određene komponente okarakterisati kao "važnije" ili "manje važne" te se u skladu s tim ponašati.

Kod JPEG standarda propisano je korištenje modela $Y'C_bC_r$. U ovom modelu Y' komponenta se naziva ***luminosity*** ili ***luma*** i predstavlja stepen subjektivne osvijetljenosti datog piksela, a komponente C_b i C_r se nazivaju ***chrominance*** ili ***chroma*** komponente i zajedno opisuju boju datog piksela. Kao što je ranije bilo govora, ovakav način opisivanja boje je prirodniji i bliži onome kako ljudsko oko procesira boju. Pokazuje se da je ljudski vid

znatno osjetljiviji na luma komponentu nego na chroma komponente. Nekad se (ušteda memorije) prilikom spremanja slike pohrane vrijednosti svih piksela *luma komponente*, dok se za *chroma komponente* sprema svaki drugi piksel bilo horizontalno, vertikalno ili i horizontalno i vertikalno. Ova tehnika se naziva ***chroma subsampling***, a unutar JPEG standarda definirane su četiri vrste chroma subsamplinga, koje ćemo opisati u nastavku.

Bez chroma subsamplinga

Ovaj format se označava ***4:4:4*** ili ***1x1*** chroma subsampling. U ovom slučaju informacija o *chroma* komponentama slike se ne zanemaruju, što daje maksimalnu kvalitetu slike, ali veće zauzeće memorije. Neka imamo sliku dimenzija 8x16 piksela, tada se za svaki upisani 8x8 blok *luma* komponente, upisuje po jedan 8x8 blok *chroma* komponente, zbog čega, bi logički gledano, u JPEG fajlu, podaci bili upisani na sljedeći način.



Ilustracija 1. Izgled blokova unutar JPEG fajla (1x1 chroma subsampling)

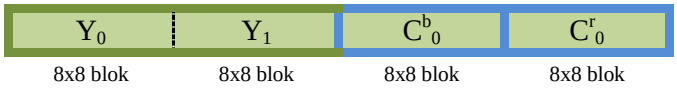
Ovakvim postupkom, niti jedan dio informacija za *chroma* komponente se ne ignorira, čime se postiže maksimalna kvaliteta slike.

Ovaj format se npr. koristi prilikom pohrane slike u Photoshopu ukoliko odaberete maksimalnu kvalitetu slike.

Izostavljanje horizontalnih chroma informacija (Horizontal subsampled)

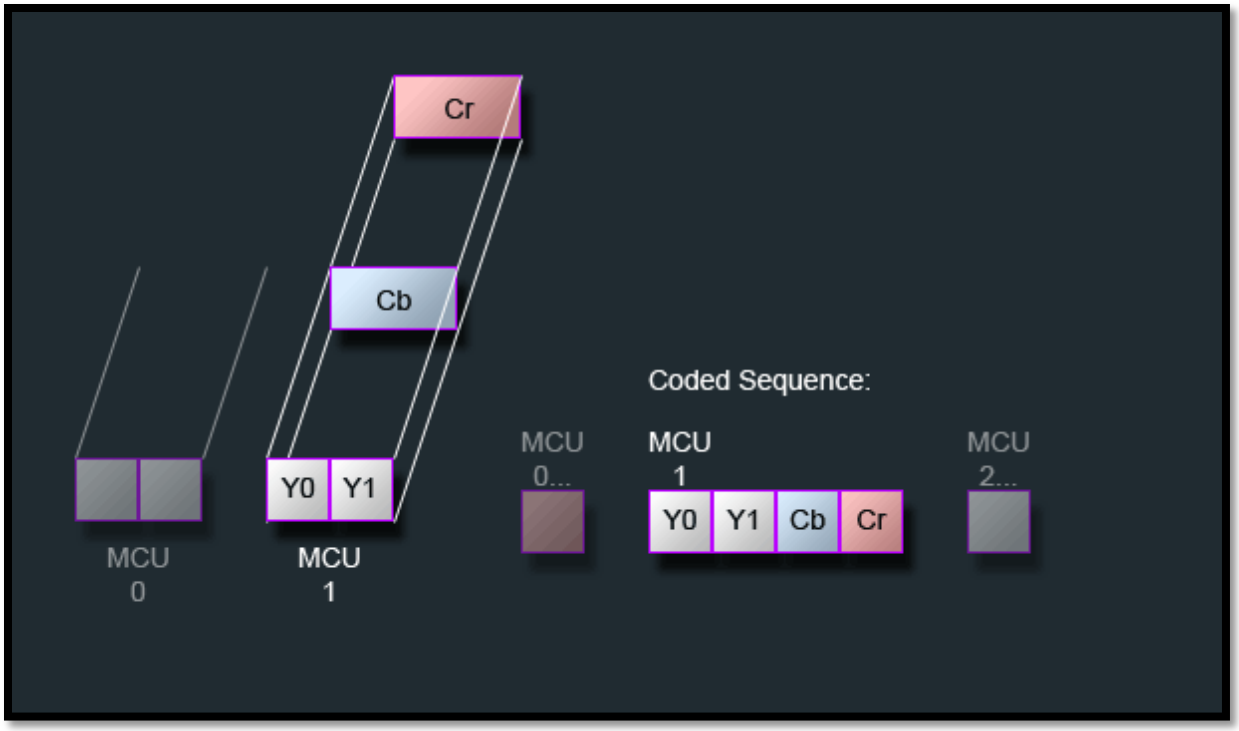
Ako se koristi chroma subsampling, onda se uglavnom koristi ovaj slučaj. Označava se kao ***4:2:2*** ili ***2x1***. U ovom slučaju, kod *chroma* komponenti *preskače se svaki drugi piksel horizontalno*. Veličina slike prije kompresije smanjuje se za 33%. Pri tome treba obratiti pažnju da se ne preskače isti piksel kod obje komponente (što se može postići ako za C_b komponentu preskakanje započnemo od nultog piksela, a za C_r komponentu, preskakanje započnemo od prvog piksela) radi subjektivno boljeg efekta.

Uvedimo ovdje pojam *MCU-a* (*Minimal Coded Unit*). Do sada bi vam svima trebalo biti jasno da se prilikom JPEG kompresije svaka slika dijeli na na blokove 8x8 piksela, međutim prilikom upisa u fajl, zbog korištenja chroma subsamplinga, može se desiti da se u JPEG fajl zaredom upisuju dva 8x8 bloka koji odgovaraju *luma* komponenti, iza kojih slijedi po jedan 8x8 blok za svaku chroma komponentu (za razliku od 1x1 chroma subsamplinga za koji smo rekli da sa svakim 8x8 blokom *luma* komponente, ide po jedan 8x8 blok *chroma* komponenti). Neka opet imamo sliku dimenzija 8x16 koju spremamo fajl korištenjem 2x1 chroma subsamplinga. Kada podatke za ovu sliku upišemo u fajl, 8x8 blokovi su logički upisani na način kako to prikazuje sljedeća slika(nemojte se zabuniti pa shvatiti da su blokovi ovako poredani na slici, *ovo je raspored blokova kad su oni upisani u JPEG fajl*). *Ako ovu sliku usporedite sa slikom za slučaj 1x1 chroma subsamplinga, uštede u veličini upisanih informacija su itekako očite, jer u ovom slučaju imamo dva „kvadratića“ manje.*



Ilustracija 2. Izgled blokova unutar JPEG fajla (2x1 chroma subsampling)

Ako blokove L_0 i L_1 horizontalno spojimo u jedan blok (vidimo da su razdvojeni isprekidanom linijom čime se naglašava da oni predstavljaju jednu logičku cjelinu), dobijamo matricu dimenzija 8x16 što predstavlja naš MCU, dakle u ovom slučaju, MCU je matrica dimenzija 8x16 piksela (za chroma subsampling 1x1 MCU je matrica 8x8 piksela, a zatšo, to zaključite sami). Kako je luma MCU matrica 8x16 piksela, tako i chroma MCU-ovi moraju imati iste dimenzije. Ako obratite pažnju, za chroma komponente upisan je samo po jedan 8x8 blok u fajl (dakle dio informacija je izostavljen upravo zbog chroma subsamplinga), a mi trebamo kreirati matrice 8x16. Da bi postigli željeni cilj, svaki *chroma* blok ćemo proširiti na dimenzije 8x16 (faktički proširili smo ih novim 8x8 blokom horizontalno), a vrijednosti originalnog 8x8 bloka ćemo prosto kopirati u novododani 8x8 blok. *Da smo kojim slučajem koristili 1x2 chroma subsampling, svaki chroma blok bi proširili na dimenzije 16x8, dakle proširili bi ga vertikalno.*



Slika 1. Izgled pohrane 8x8 blokova prilikom korištenja 2x1 chroma subsamplinga i generiranje MCU-a

Na *Slici 1*, blokovi Y_0 i Y_1 su zajedno označeni kao jedan *MCU*, što zaista ima smisla, jer mi, logički gledano, u fajl zapisujemo matricu 8x16, a otuda i naziv MCU, jer minimalna količina podataka koju upisujemo za svaku komponentu prilikom korištenja chroma subsamplinga 2x1 je upravo matrica 8x16 (prvo upisujemo 8x8 blokove L_{00} i L_{10} koji kad se horizontalno spoje, zaista tvore matricu 8x16 piksela). Na *Slici 1* desno, prikazano je kako su 8x8 blokovi upisani u fajl, a lijevo kako su oni spojeni u MCU.

Ovdje itekako treba paziti, da ne bi došlo do zabune. Podaci se originalno u fajl zapisuju kao 8x8 blokovi, međutim logički gledano, dva 8x8 bloka L_0 i L_1 tvore MCU dimenzija 8x16, pa zato kažemo da se u fajl upisuje MCU, ali to je samo logički gledano.

Veličina MCU-a određen je tipom chroma subsamplinga. Ako se koristi chroma subsampling 1x1 MCU je matrica dimenzija 8x8, 2x1 MCU je matrica dimenzija 8x16, 1x2 MCU je matrica dimenzija 16x8 i na kraju 2x2 MCU je matrica dimenzija 16x16, baš kako je to objašnjeno kroz sljedeći tekst.

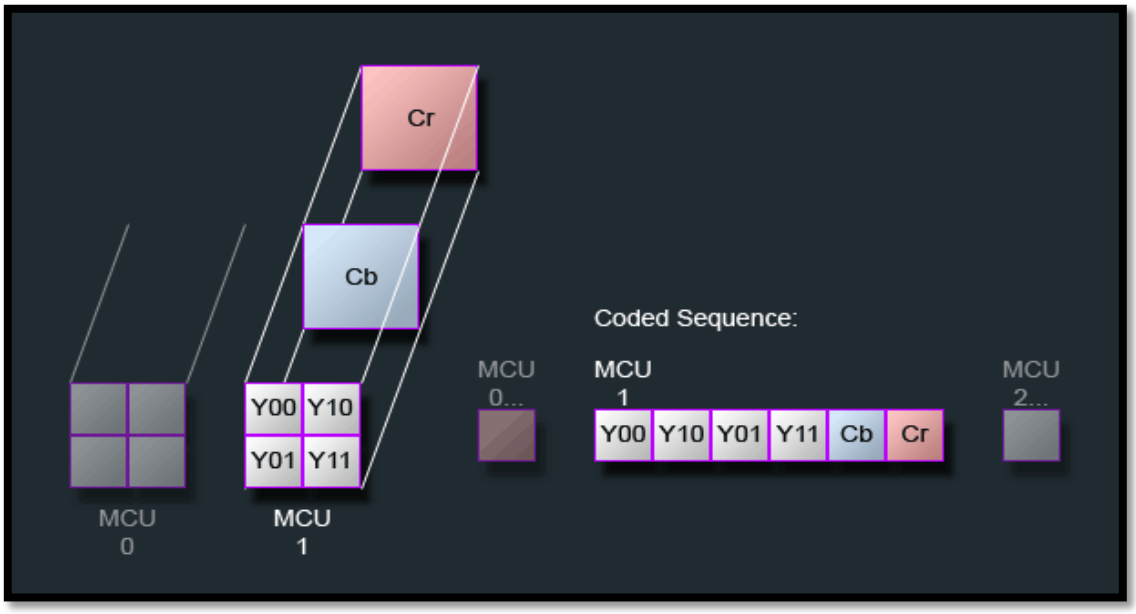
Izostavljanje vertikalnih chroma informacija (Vertical subsampled)

O ovom slučaju nećemo govoriti posebno, obzirom da se on od prethodnog razlikuje samo po tome što se chroma komponente zanemaruju vertikalno, umjesto horizontalno. Oznaka je 4:2:2 ili 1x2, a MCU je matrica dimenzija 16x8, i ovaj slučaj se jako rijetko koristi.

Izostavljanje i vertikalnih i horizontalnih chroma informacija (Horizontal & Vertical subsampled)

U ovom slučaju, pored preskakanja svakog drugog piksela horizontalno, preskače se i svaki drugi piksel vertikalno. Ukupna veličina slike prije kompresije se smanjuje za 50%, što je sa stanovišta memorije odlično, ali s druge strane narušena je kvaliteta slike. Ovaj način pohrane podataka koristi *MS Paint* prilikom pohrane slike u JPEG format. Da bismo prikazali smisao 2x2 chroma subsamplinga, uzmimo u obzir sliku dimenzija 16x16 piksela, tada interno u fajlu, blokovi su upisani na sljedeći način.

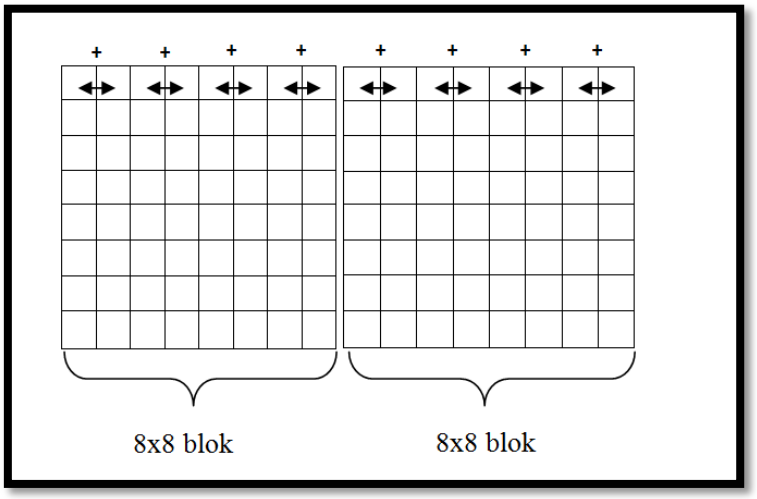
Ilustracija 3. Izgled blokova unutar JPEG fajla (2x2 chroma subsampling)



Slika 2. Izgled pohrane 8x8 blokova prilikom korištenja 2x2 chroma subsamplinga i generiranje MCU-a

U ovom slučaju, u fajl se zaredom upisuju četiri 8x8 bloka *luma* komponente, iza čega slijedi po jedan 8x8 blok svake od *chroma* komponenti. Kada dekoder čita ove podatke, tada četiri *luma* bloka treba složiti u matricu 16x16, kako to pokazuje *Slika 2 (lijevo).*

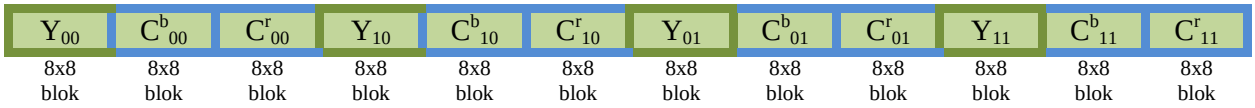
Očito je u ovom slučaju MCU jednak matrici dimenzija 16x16 piksela. Možda vas još buni MCU, ali pogledajte desni dio ove slike (ovaj niz). Kao što se vidi sa slike, prvo idu Y_{00}, Y_{10},Y_{01}, Y_{11} blokovi (nemojte da vas zabune indeksi elemenata Y_{10},Y_{01}, naime kod slika, prva cifra indeksa označava kolonu, a druga red). Ovdje se zaista radi o 8x8 blokovima koji se zatim slažu u matricu 16x16 kako bi formirali MCU (onim redom kako je to prikazano na lijevoj polovici slike). Dakle, u JPEG fajlu originalno su zapisani blokovi 8x8, a na programeru je posao da ih on složi u defaultni MCU koji je u ovom slučaju dimenzija 16x16 piksela. Dimenzije defaultnog MCU-a određuje, kao što smo već rekli, tip chroma subsamplinga. Ako bolje pogledate, možete primijetiti da su u nizu (desna polovina slike) upisani blokovi 8x8 za *chroma* componente, a MCU je dimenzija 16x16. Da bi popunili MCU za *chroma* komponente koji je također matrica 16x16 kao što to pokazuje lijevi dio *Slike 2*, svaku *chroma* komponentu ćete prosto kopirati 4 puta i dobit ćete *chroma* MCU dimenzija 16x16. S druge strane, prilikom spremanja slike, da biste od dva 8x8 bloka (dakle od matrice 8x16) dobili jedan 8x8 blok za *chroma* komponente (dakle koristite npr. 2x1 chroma subsampling) to možete uraditi tako što ćete svaki drugi piksel prosto ignorirati, ili da napravite funkciju koja će za dva susjedna piksela, naći srednju vrijednost, a rezultat pohraniti u novi 8x8 block (ovaj postupak se preporučuje prilikom implementacije JPEG enkodera). Dakle postupak bi bio sljedeći:



Kao što vidimo, od matrice dimenzija 8x16 piksela dobit ćemo matricu 8x8 piksela, čime smo postigli 2x1 chroma subsampling. Na analogan način, mogu se implementirati i svi ostali slučajevi chroma subsamplinga. Broj luminance 8x8 blokova koji se jedan do drugog upisuju u fajl, određen je umnoškom 2x1=2 ili 1x2=2 ili 1x1=1 ili 2x2=4.

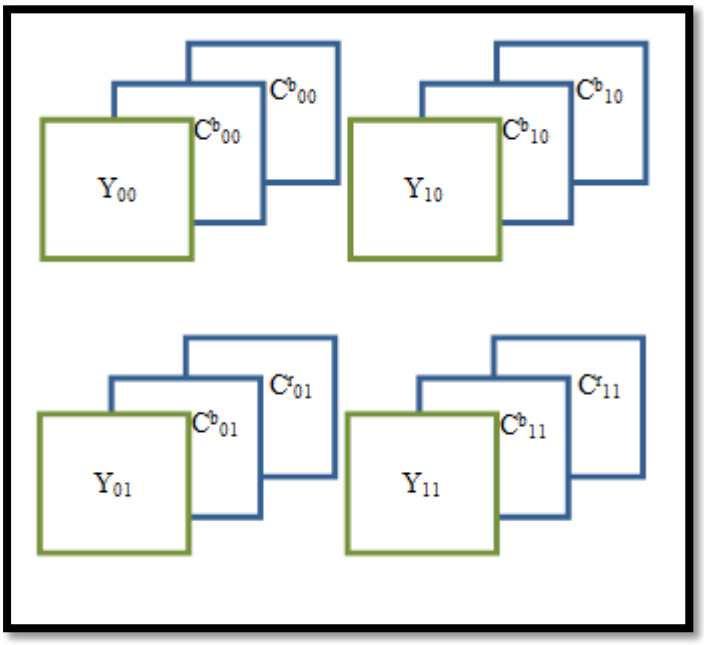
Pored ovoga, JPEG format dozvoljava i korištenje RGB modela boje kod slika najviše kvalitete, mada se to rjeđe koristi.

Za kraj ćemo sve naučeno sada primijeniti na sliku dimenzija 16x16 piksela. Kada bismo ovu sliku spremili bez gubitka informacijâ za *chroma* komponente (1x1 chroma subsampling), tada bismo logički unutar fajla imali sljedeći poredak blokova.



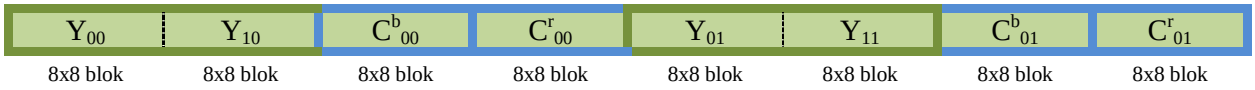
Ilustracija 4. Izgled blokova unutar JPEG fajla (1x1 chroma subsampling)

Kada blokove iz JPEG fajla „redamo“ u sliku, dobili bismo sljedeći rezultat.



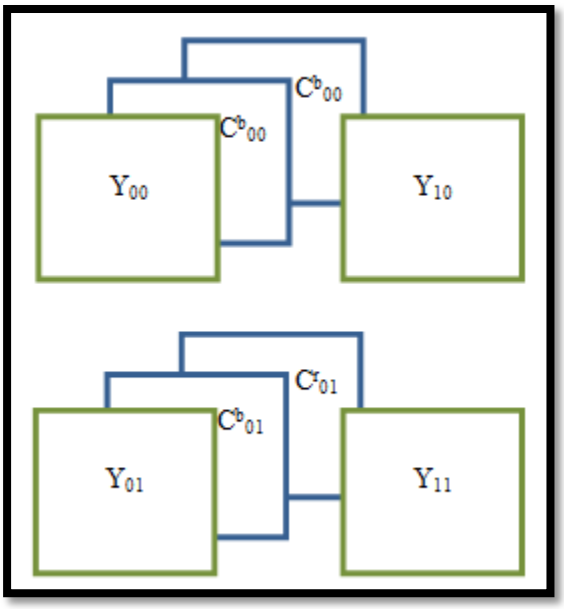
Ilustracija 5. Izgled blokova prilikom redanja u originalnu sliku

Ako istu sliku pohranimo uz korištenje 2x1 chroma subsamplinga, blokovi unutar JPEG fajla izgledat će kako je to prikazano na sljedećoj slici.



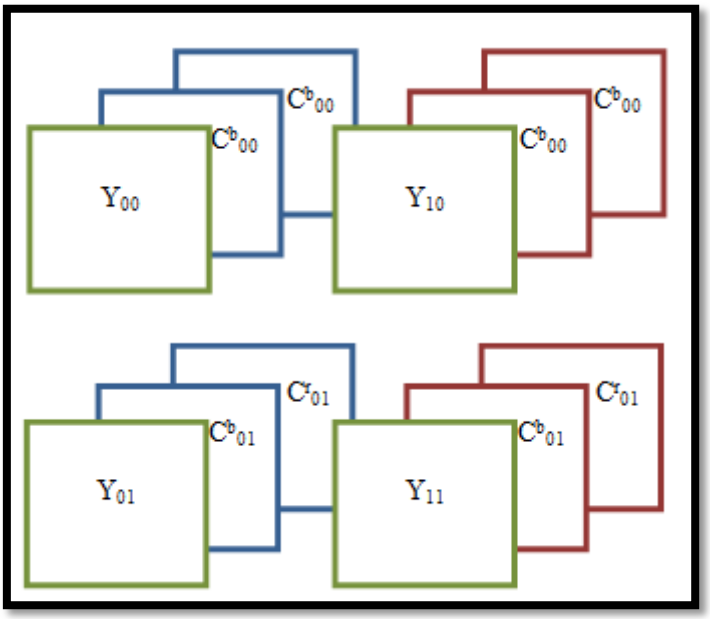
Ilustracija 6. Izgled blokova unutar JPEG fajla (2x1 chroma subsampling)

Ako bismo sada pokušali dobiti originalnu sliku, tada bismo dobili sljedeći rezultat.



Ilustracija 7. Izgled blokova prilikom redanja u originalnu sliku (2x1 chroma subsampling)

Na *ilustraciji iznad* jasno se vidi da Y_{10} i Y_{11} blokovi nemaju svoje odgovarajuće *chroma* blokove (odakle dobijamo uštedu memorije). Da bismo zaista dobili odgovarajuće *chroma* blokove i za *luma* Y_{10} i Y_{11} blokove tada ćemo postojeće *chroma* blokove prosto kopirati i dobiti sljedeći rezultat (crveni kvadrati su zapravo kopirani blokovi).



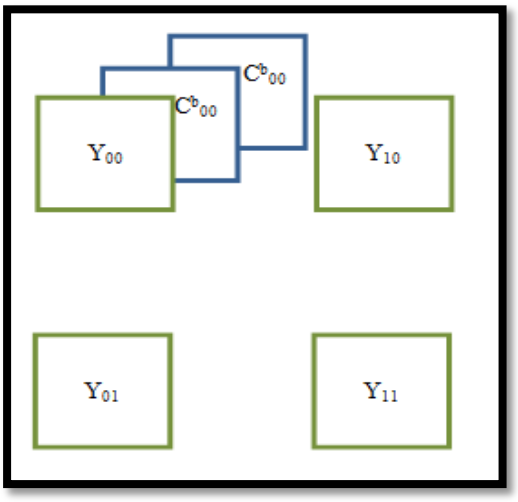
Ilustracija 8. Izgled blokova prilikom redanja u originalnu sliku, nakon kopiranja chroma blokova na preostale luma blokove (2x1 chroma subsampling)

Obzirom da je slučaj 1x2 chroma subsamplinga veoma sličan 2x1 chroma subsamplingu, a i rijetko se koristi, ovdje nećemo navoditi navedeni slučaj, a možete ga samostalno uraditi za vježbu.

Za kraj ćemo izložiti i slučaj 2x2 chroma subsamplinga. Za istu sliku, logički, 8x8 blokovi unutar JPEG fajla bi izgledali na sljedeći način.

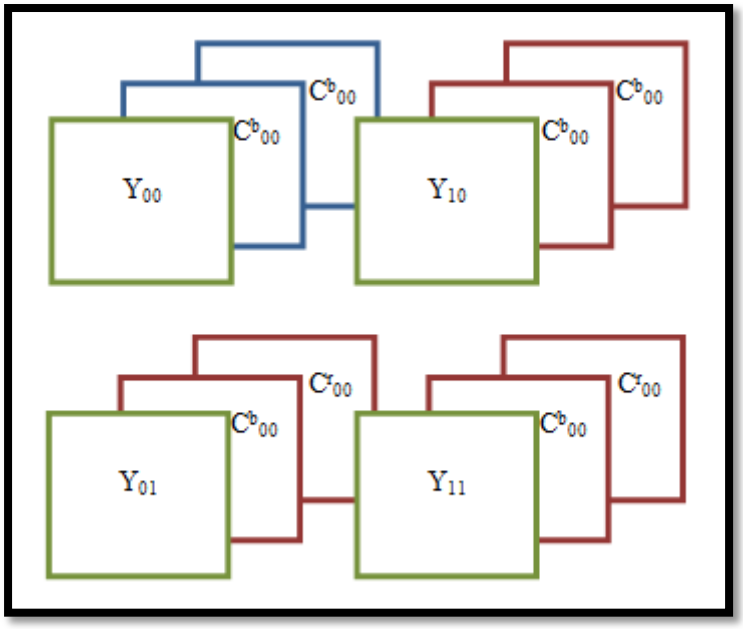
Ilustracija 9. Izgled blokova unutar JPEG fajla (2x2 chroma subsampling)

Ako bismo iz fajla restaurirali sliku, tada bismo dobili sljedeći rezultat.



Ilustracija 10. Izgled blokova prilikom redanja u originalnu sliku (2x2 chroma subsampling)

Sa slike se može vidjeti kako su za elemente Y_{10}, Y_{01}, Y_{11} izostavljeni *chroma* blokovi, što je rezultat chroma subsamplinga, a da bismo za svaki *luma* blok dobili odgovarajući *chroma* blok, jednostavno kopirajmo *chroma* blokove tri puta, kako je to prikazano na sljedećoj slici (crveni kvadratići označavaju kopirane *chroma* blokove).



Ilustracija 11. Izgled blokova prilikom redanja u originalnu sliku, nakon kopiranja chroma blokova na preostale luma blokove (2x2 chroma subsampling) – crveno su kopirani blokovi

Konverzija u $Y'C_bC_r$ model boja

Konverzija modela boja je složeno pitanje o kojem smo već diskutovali. Osnovni problem je što je RGB model "relativan", odnosno on nam ne govori ništa o osobinama uređaja za prikaz, dok su ostali modeli "apsolutni" pa je potrebno najprije RGB sliku konvertovati u apsolutnu RGB podvarijantu ili barem uraditi gama korekciju takve slike. Dalje, nakon konverzije po određenim formulama dobit će se format koji se naziva YP_bP_r koji neefikasno koristi dostupni bandwidth te ga je potrebno dodatno transformisati kako bi se dobio željeni $Y'C_bC_r$.

Srećom, sam JPEG standard propisuje formulu za konverziju direktno iz RGB u $Y'C_bC_r$ (tema za razmišljanje je šta onda predstavlja tako konvertovana slika, ali nama je samo važno da je dekompresovana slika što sličnija originalnoj). Formula fiksno glasi:

$$Y = 0.299 * R + 0.587 * G + 0.114 * B$$
$$Cb = 128 - 0.168746 * R - 0.331264 * G + 0.5 * B$$
$$Cr = 128 + 0.5 * R - 0.418688 * G - 0.081312 * B$$

Ako bolje pogledamo gornje formule, možemo primijetiti da se Y' komponenta zapravo ponaša kao filter. Kako je oko najosjetljivije na zelenu boju, to je i koeficijent uz G komponentu i najveći, zatim ga slijedi koeficijent uz crvenu R, a zatim uz plavu B komponentu.

Iz ovoga se može izvesti inverzna formula za konverziju $Y'C_bC_r$ u RGB:

$$R = Y - 179.4561 + 1.402 * Cr$$
$$G = Y + 135.4581 - 0.714136 * Cr - 0.34413186 * Cb$$
$$B = Y - 226.8146 + 1.772 * Cb$$

Kod praktične implementacije treba obratiti pažnju na nekoliko stvari. Zbog velikog broja decimalnih mjesta praktično je garantovano da će doći do određene greške zaokruživanja. Pri tome treba imati na umu da je gamut modela $Y'C_bC_r$ veći od gamuta RGB modela. U kombinaciji, ovo dvoje može rezultirati ilegalnim vrijednostima!

Neka je data boja R=0, G=255, B=255 (čista cijan). Prema formulama za konverziju u $Y'C_bC_r$ model boja dobijamo: Y=178.755, C_b=171.02768, C_r =0.5. Ako samo konvertujemo podatke u tip int, odbacit će se cifre iza decimalne tačke pa ćemo imati: Y=178, Cb=171, Cr=0.

Nakon konverzije nazad u RGB koristeći formule za konverziju u RGB model boja imamo: R=-1.4561, G=254.6116, B=254.1974. Primijetimo da je *R negativno, dakle izvan opsega.*

Ako bismo u pretprošlom koraku koristili funkciju round, tako da bude Y=179, dobili bismo nešto bolji rezultat: R=-0.4561, G=255.6116, B=255.1974. *Obratite pažnju da je round (G)=256 što je ponovo izvan opsega.*

Dakle, pri ovim konverzijama će nužno doći do jedne nove vrste kvantizacijske greške odnosno clippinga, ali se smatra da ona nije velika.

Također, kod implementacije u C++ imajte na umu da će množenja sa decimalnim brojevima biti relativno spora. Postoje načini da se ta množenja ubrzaju.

Zadatak 3.1. *U vašu funkciju dct() dodajte konverziju iz RGB u $Y'C_bC_r$ model boje prije ulaska u DCT algoritam. Analogno, u funkciju idct() dodajte konverziju iz $Y'C_bC_r$ u RGB nakon završetka svih ostalih faza. Ne zaboravite da naše RGB komponente trebaju imati vrijednosti u opsegu 0-255 a DCT očekuje vrijednosti u opsegu -128 - 127.*

U ovom trenutku treba spomenuti da je matrica kvantizacije Q koju smo dali ranije *(4)* primjenjiva samo na Y' komponentu boje. Ako se primijeni na chroma komponente dobit ćemo prilično loše rezultate, pa bi trebalo dodati još jednu matricu za chroma komponente:

$$
\begin{bmatrix}
17 & 18 & 24 & 47 & 99 & 99 & 99 & 99 \\
18 & 21 & 26 & 66 & 99 & 99 & 99 & 99 \\
24 & 26 & 59 & 99 & 99 & 99 & 99 & 99 \\
47 & 66 & 99 & 99 & 99 & 99 & 99 & 99 \\
99 & 99 & 99 & 99 & 99 & 99 & 99 & 99 \\
99 & 99 & 99 & 99 & 99 & 99 & 99 & 99 \\
99 & 99 & 99 & 99 & 99 & 99 & 99 & 99 \\
99 & 99 & 99 & 99 & 99 & 99 & 99 & 99
\end{bmatrix} \quad (4')
$$

Zadatak 3.2. *U funkcijama dct() i idct() implementirajte chroma subsampling po modelu 4:2:2.*

Savjet: Najprije napravite dio koda koji vrši konverziju RGB u $Y'C_bC_r$ i chroma subsampling. Rezultat ove konverzije mogu biti dvije matrice: Y (luminance) matrica i C (chrominance) matrica u koju se naizmjenično stavljaju vrijednosti C_b i C_r. Zatim prepravite do sada urađenu DCT petlju tako da se najprije izvrši nad jednom matricom a zatim nad drugom.

(Nakon primjene chroma subsamplinga kvantizacija mi je pravila slike loše kvalitete... moram to još istražiti.)

DPCM i RLE kodiranje

Napomena: *Naredna poglavlja predstavljaju nešto naprednije teme za studente koji žele da implementiraju kompletnu podršku za JPEG/JFIF u Matlabu.*

Posmatrajmo jedan tipičan blok 8x8 nakon izvršenih svih koraka JPEG kompresije koje smo do sada implementirali (u Matlabu dvokliknite na naziv matrice izlaza iz dct() s desne strane):

Možemo uočiti dvije stvari:

1. Broj u gornjem lijevom uglu je po apsolutnoj vrijednosti znatno veći od ostalih članova matrice.
2. Kako idemo dijagonalno prema donjem desnom uglu, članovi matrice (AC koeficijenti) su uglavnom nule.

Upravo ove osobine su posljedica korištenja DCT transformacije i $Y'C_bC_r$ modela boje i to je ono što čini ovako kreirane blokove pogodnim za kompresiju.

Prije nego što počnemo izlaganje pojmova spomenutih u naslovu, iznesimo nekoliko rečenica o *Cik Cak kodiranju* (*Zig Zag coding*).

Cik Cak kodiranje

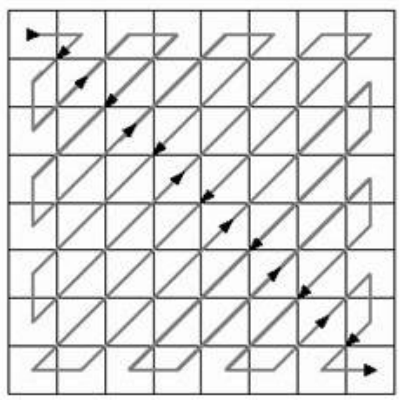
Na sljedećoj slici prikazan je jedan 8x8 blok dobiven na način da je nad njim izvršena DCT, a zatim kvantizacija.

$$B = \begin{bmatrix} -26 & -3 & -6 & 2 & 2 & -1 & 0 & 0 \\ 0 & -2 & -4 & 1 & 1 & 0 & 0 & 0 \\ -3 & 1 & 5 & -1 & -1 & 0 & 0 & 0 \\ -3 & 1 & 2 & -1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}.$$

Dakle, DC element ovog bloka je element *B(0,0)* koji je jednak -26. Ostali elementi su AC elementi.

Jedna osobina ovog bloka jeste da se vrijednost elementa matrice, krećući se dijagonalno iz lijevog gornjeg ugla prema desnom donjem uglu, jednaki nuli tj. vrijednosti elemenata se sve više smanjuju što se više krećemo prema donjem desnom uglu matrice. Kao što ćemo vidjeti, RLC algoritam kompresije daje odlične rezultate ukoliko se elementi sa jednakim vrijednostima nalaze jedan pored drugog. Upravo *Cik Cak* kodiranje je korak kojim ćemo postići željeni cilj.

Cik-Cak kodiranje prikazano na sljedećoj slici.



Slika 3. Cik - Cak kodiranje

Pomoću *Cik-Cak* kodiranja omogućava se efikasnija primjena algoritama za kompresiju kao što je RLC (*Run Level Coding*) kodiranje. Zašto se baš koristi *Cik-Cak* kodiranje? Kako su koeficijenti uz kosinusne funkcije visokih frekvencija (a koji su dobiveni diskretnom kosinusnom transformacijom) smješteni u donjem desnom kutu 8x8 bloka i koji imaju osobinu da su često jednaki nuli, to bi veoma korisno bilo da od ove matrice možemo generirati niz elemenata kod kojeg će elementi koji su jednaki nuli biti blizu jedan drugog (upravo radi RLC algoritma) tj. da elementi iz donjeg desnog kuta ove matrice budu jedan uz drugog. *Cik-Cak* kodiranje nam upravo to omogućava, pa ćemo iz ove matrice zapravo generirati ***niz*** čiji su elementi poredani na sljedeći način.

K(0,0)	***K(0,1)***	***K(1,0)***	***K(2,0)***	***K(1,2)***	***K(0,3)***	**…**	***K(7,5)***	***K(6,6)***	***K(5,7)***	***K(6,7)***	***K(7,6)***	***K(7,7)***

Tabela 1. Rezultat Cik-Cak kodiranja

Na ovaj način postigli smo cilj tj. elementi koji imaju vrijednost 0, bit će poredani jedan pored drugog što je dobar preduvjet za primjenu RLC algoritma za kompresiju bez gubitaka. Rezultat *Cik Cak* kodiranja za gornju matricu B je: -26 -3 0 -3 -2 -6 2 -4 1 -3 1 1 5 1 2 -1 1 -1 2 0 0 0 0 0 -1 -1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0.

DPCM kodiranje DC koeficijenata

Zbog prostorne redudancije na slici, sasvim je jasno da će vrijednosti DC koeficijenata susjednih 8x8 blokova biti približno iste (najveća razlika između njih bit će na naglim prijelazima slike). Obzirom da DC koeficijenti mogu imati velike vrijednosti, bilo bi odlično naći šemu pomoću koje bi se oni kodirali sa što manje bita. Upravo spomenuta osobina (dakle razlika između DC koeficijenata susjednih blokova je malena) DPCM (*Differential Pulse-Code Modulatio*n) metodu čini idealnom za kodiranje DC elemenata. Radi se o tome da se svaki DC element kodira na sljedeći način:

$$Razlika = DC_i - DC_{i-1} \rightarrow gdje\ je\ za\ i = 0, DC_{i-1} = 0$$

Neka DC koeficijenti 8x8 blokova imaju sljedeće vrijednosti: $(50)_1$, $(70)_2$,$(35)_3$, $(63)_4$, $(115)_5$, $(270)_6$, $(580)_7$, gdje broj u indeksu označava redni broj 8x8 bloka (dakle vrijednost DC koeficijenta prvog 8x8 bloka slike je 50, drugog 8x8 bloka 150 itd…), tada će oni prema gornjoj formuli biti kodirani kao: $(50)_1$, $(20)_2$, $(-35)_3$, $(28)_4$, $(52)_5$, $(155)_6$, $(310)_7$. Ovim postupkom se smanjuje broj bita potreban za predstavu svakog DC elementa, čime se smanjuje ukupna veličina JPEG fajla. Navedeni proces se koristi prilikom pohrane slike, a suprotan prilikom učitavanja i to po formuli:

$$DC_i = Razlika + DC_{i-1} \rightarrow gdje\ je\ za\ i = 0, DC_{i-1} = 0$$

Ako ovu formulu primijenimo na niz: $(50)_1$, $(20)_2$, $(-35)_3$, $(28)_4$, $(52)_5$, $(155)_6$, $(310)_7$, dobit će polazni niz: $(50)_1$, $(70)_2$, $(35)_3$, $(63)_4$, $(115)_5$, $(270)_6$, $(580)_7$.

Za razliku od DC koeficijenata, AC koeficijenti se kodiraju na nešto drugačiji način, kao što ćemo vidjeti u nastavku.

Zadatak 4.1. *Prepravite vaše funkcije dct() i idct() tako da se koristi DPCM. Odnosno, svaki DC element sa indeksom (0,0) (a u Matlabu to je (1,1)) zamijenite razlikom u odnosu na prethodni DC element. Da bi to moglo raditi, u dct() funkciji morate u nekoj privremenoj varijabli čuvati prethodnu vrijednost DC elementa.*

Matlab kod dobivenih funkcija upišite u prostor ispod:

RLC Algoritam

RLC algoritam je učinkovit u slučaju kada u nekom nizu dođe do ponavljanja određenih brojeve/karaktera/simbola. Ovaj algoritam je samo specijalan slučaj RLE (eng. *Run Length Encoding*) agoritma koji se primjenjuje na bilo kakav niz karaktera, dok se RLC uglavnom koristi za nizove brojeva koji sadrže veliki broj nula.

Neka imamo sljedeći niz: 15 0 0 0 0 0 0 1 2 3 65 4 47 0 0 0 8 19 21 0 0 0 0 0 0 0 0. Jedan način na koji je moguće kodirati ovaj niz brojeva je sljedeći (0,15), (6,1), (0,2), (0,3), (0,65), (0,4), (0,47), (3,8), (0,19), (0,21), (8,EOB). Što bi se moglo pročitati na sljedeći način. Prije broja 15 nemamo niti jednu nulu tj. niz započinje brojem 15. Iza broja 15 nalazi se šest nula nakon kojih slijedi broj 1. Iza jedinice dolaze brojevi 2, 3, 65, 4, 47, nakon čega slijede tri nule pa broj 8 itd… Dakle u uređenom paru *(x,y)* element *x* predstavlja broj nula koje se pojavljuju prije elementa *y*. Ovo je ukratko osnovna ideja *Run Level Coding* algoritma.

Na isti način se kodiraju AC koeficijenti nekog 8x8 bloka. Sada postaje jasno zašto se koristi Cik Cak kodiranje i kakve su njegove posljedice na primjenu RLC algoritma.

JPEG standard ovdje uvodi dva specijalna karaktera. Prvi jeste EOB (kodira se kao 00h) koji označava da su svi elementi iza posljednjeg AC elementa koji je različit od nule, jednaki nuli. Dakle neka je rezultat Cik Cak kodiranja niz -25 0 0 0 5 0 0 0 0….0 (gdje tri tačke označavaju da su svi elementi do kraja bloka jednaki nuli). Ukoliko na ovaj niz primijenimo RLC kodiranje, imat ćemo: (0,25), (3,5), EOB. Razlog za ovo je očit, ušteda memorije.

Drugi karakter na koji moramo obratiti pažnju jeste 0xF0 karakter. Ovaj karakter se koristi svaki put kada u nizu prebrojimo 16 uzastopnih nula, a do kraja niza postoji barem jedan AC element koji je različit od nule.

Primjer takvog niza bi bio: -25 0 0 5 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 12 0 0 0 5 0 0 0 …0. Očito kod ovog niza imamo 19 uzastopnih nula, ali iza njih do kraja niza postoji barem jedan AC element koji je različit od nule. Ako navedeni niz kodiramo pomoću RLC algoritma, tada imamo sljedeći rezultat: (0,25), (2,5), (15,0), (3,12), (3,5), EOB. Uređeni par (15,0) je naš specijalni karakter koji kaže da smo kodirali 16 nula (nemojte da vas zavara broj 15 i da mislite da se radi o 15 nula. Broj dolazi od $(15)_{10}=(F)_{16}$, a uređeni par (15,0) čitamo kao: *prije AC elementa koji je jednak nuli, prebrojali smo još 15 nula, dakle sve ukupno 16 nula*, isto kao što bi znak (15,3) pročitali kao: *prije AC elementa koji je jednak 3, prebrojali smo 15 nula*).

Ako bi imali 15 uzastopnih nula, iza kojih dolazi AC element različit od nule kao u sljedećem nizu: -25 0 0 5 0 0 0 0 0 0 0 0 0 0 0 0 0 12 0 0 0 5 0 0 0 …0, tada bi ovaj niz kodirali kao: (0,25), (2,5), (15,12), (3,5), EOB. Razlika je očita, odnosno (15,12) nije specijalni karakter čime smo izbjegli dvosmislenost u odnosu na specijalni karakter (15,0).

Ako ste barem malo razmislili, onda je logično pitanje zašto je baš uveden znak 0xF0 odnosno (15,0). Zašto ne bi mogli napisati (17,0)? Kao što ćemo vidjeti u nastavku, broj bita

kojim predstavljamo koliko nula se nalazi ispred nekog nenultog AC elementa je 4, pa je maksimalna vrijednost koju možemo predstaviti sa 4 bita 2^4-1=15 + 1=16, pri čemu +1 dolazi od elementa *y* uređenog para (15,0) kojim govorimo da iza 15 nula ima još jedna nula.

Projektanti standarda su radili sve moguće „akrobacije“ kako bi što više smanjili veličinu fajla, a zadržali kvalitetu slike. U tu svrhu, vrijednosti DC i AC elemenata se kao takvi nisu pohranjivali u fajl, nego su se nad njima vršile određene transformacije. O čemu se radi, pogledajmo u nastavku.

Kodiranje AC elemenata

Da bi vam sve bilo jasnije, krenut ćemo od jednog 8x8 bloka nad kojim su izvršene DCT, kvantizacija.

-752	-12	15	0	0	7	0	0
0	0	0	1	2	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Ako primijenimo i Cik Cak kodiranje, tada dobijamo sljedeći niz: -752 -57 0 0 0 15 0 0 0 0 0 0 0 1 0 7 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0.

Kodirajmo gornji niz RLC algoritmom: (0,-752), (0,-57), (3,15), (7,1), (1,7), (0,2), EOB. Obzirom da je -752 DC element, njega ćemo ovdje prosto ignorirati, a način kodiranja DC elemenata izložit ćemo nešto kasnije.

Skoncentrirajmo se sada na element *y* uređenog para *(x,y)*. U obzir uzmimo uređeni par (0,-57). Umjesto da pohranimo konkretnu vrijednost -57, JPEG standard nalaže da pronađemo kategoriju vrijednosti -57, a da samu vrijednost predstavimo binarnim zapisom. Na svu sreću, JPEG standard predlaže i tabelu pomoću koje možemo uraditi sve potrebne transformacije. Tabela je sljedeća:

Kategorija	Binarna vrijednost		AC Vrijednost	
0			0	
1	0	1	-1	1
2	00,01	10,11	-3,-2	2,3
3	000,001,010,011	100,101,110,111	-7,-6,-5,-4	4,5,6,7
4	0000,...,0111	1000,...,1111	-15,...,-8	8,...,15
5	0 0000,...	...,1 1111	-31,...,-16	16,...,31
6	00 0000,...	...,11 1111	-63,...,-32	32,...,63
7	000 0000,...	...,111 1111	-127,...,-64	64,...,127
8	0000 0000,...	...,1111 1111	-255,...,-128	128,...,255
9	0 0000 0000,...	...,1 1111 1111	-511,...,-256	256,...,511

10	00 0000 0000,...	...,11 1111 1111	-1023,...,-512	512,...,1023
11	000 0000 0000,...	...,111 1111 1111	-2047,...,-1024	1024,...,2047

Tabela 2. Tabela za transformaciju

Na osnovu tabele lako možemo vidjeti da broj -57 spada u kategoriju 6 (ako se pitate šta je to kategorija onda je odgovor jednostavan: kategorija predstavlja broj kojim se označava koliko bita je potrebno da se binarno predstavi AC koeficijent. U našem slučaju AC koeficijent je -57 i spada u kategoriju 6, odakle slijedi da je potrebno 6 bita za njegovo kodiranje). Dakle na mjesto -57 sada pišemo 6, pa dobijamo uređeni par (0,6). Preostaje još da pronađemo binarnu reprezentaciju broja -57. Ako je broj pozitivan, nađemo njegov binarni ekvivalent i zapišemo ga. Ukoliko je broj negativan, nađemo njegov prvi komplement i zapišemo ga. Da se podsjetimo. Prvi komplement ima smisla tražiti za negativne brojeve, a nađemo ga tako što zapišemo |-57|, a zatim sve cifre invertujemo (| |-označava apsolutnu vrijednost). U našem slučaju broj -57 je negativan pa ćemo potražiti prvi komplement broja -57. Prvi komplement broja $-(57)_{10}$ je $(000110)_{1k}$. Očito binarni zapis ima 6 cifri, što je jednako kategoriji (nule ispred prve jedinice ne smijemo zanemariti). Dakle uređeni par (0,-57) ćemo kodirati kao:

$$\frac{(0,-57)}{(0,6),000110}$$

Prije nego što kodiramo i ostale uređene parove, napomenimo da standard propisuje da se uređeni par (*x*,*y*) mora kodirati sa 8 bita. Kako gornja T*abela 2* propisuje maksimalno 15 kategorija, to je za zapis *y* elementa uređenog para potrebno 4 bita (minimalan broj bita da predstavimo broj 15 binarno je 4). Preostala 4 bita odlaze na element *x*. Sada je sasvim jasno zašto u specijalnom karakteru (15,0) ne možemo pisati (17,0), jer kako ćemo broj 17 binarno napisati na 4 bita!?

Ako kodiramo ostale elemente niza dobijamo sljedeće rezultate:

Prije transformacije	(0,-57)	(3,15)	(7,1)	(1,7)	(0,2)	EOB
Poslije transformacija	***(0,6),000110***	***(3,4),1111***	***(7,1),1***	***(1,3),111***	***(0,2),10***	***(0,0)***

Tabela 3. Transformacija uređenih parova i binarno prestavljanje vrijednosti AC koeficijenata.

Obzirom da uređeni par (*x*,*y*) predstavlja jedan bajt, onda je sasvim normalno možemo čuvati u promjenjivoj tipa *char*.

Ako napravite inverznu transformaciju lako se može pokazati da ćemo doći do našeg početnog 8x8 bloka.

Već do ovog koraka proces je postao komplikovan, no ovdje nije kraj. Uređeni parovi (*x*,*y*) se često mogu ponavljati (ako vam je lakše, uređeni par je jedna promjenjiva tipa *char* pa samim time uređene parove možete gledati kao simbole ASCII tabele), što ih čini izvanrednim za *Huffmanovo kodiranje* koje je spomenuto u jednom od ranijih tutorijala. Proces generiranja huffmanovih kodova za slučaj JPEG enkodera je jako složen, te ga ovdje nećemo objašnjavati, ali da bi do kraja vidjeli kako radi JPEG enkoder, uređenim parovima u

Tabeli 3 dodijelit ćemo proizvoljne huffmanove kodove, kako je to prikazano u sljedećoj tabeli.

Uređeni par	*ASCII simbol uređenog para*	*Huffmanov kod*
(0,6)		**1110**
(3,4)	4	**110**
(7,1)	q	**101**
(1,3)	DC3	**100**
(0,2)	STX	**10**
(0,0)	NULL	**0**

Tabela 4. Huffmanovi kodovi uređenih parova

JPEG standard propisuje 2 osobine koje huffmanovi kodovi moraju zadovoljavati:

1. *Niti jedan huffmanov kod ne smije biti sačinjen samo od jedinica (npr. 111), a zašto pročitajte na samom kraju ovog dokumenta.*
2. *Niti jedan huffmanov kod ne smije biti sadržan u drugom, krećući od početka*. Npr. Neka imamo tri huffmanova koda kojim su kodirani sljedeći simboli: 0=a, 01=b i 10=c. U huffmanovom kodu 01 sadržan je kod 0, ali u kodu 10 nije. Zašto? Ako bismo imali sljedeći niz bita: 0 01 10 kojim smo u fajl htjeli kodirati simbole *abc*, tada bi dekoder pročitao prvu nulu i povezao je sa simbolom *a* koji je kodiran sa 0. Zatim bi pročitao drugu nulu i povezao je opet sa simbolom *a*, jer dekoder nikako ne može znati da druga nula zapravo pripada simbolu *b* koji je kodiran kao 01. Dalje bi dekoder pročitao cifru 1 i zaključio da ne postoji niti jedan simbol kodiran sa jedinicom, te bi pročitao sljedeću cifru, a to je opet jedinica, te bi zaključio da ne postoji simbol kodiran sa 11 i tako sve do kraja fajla. Nakon što pročita posljednju cifru, dekoder će zaključiti da ne postoji simbol kodiran sa 110. Na ovaj način dekodiran je sljedeći tekst: *aa* umjesto *abc*. Ukoliko su zadovoljena gornja pravila, dekoder nikad neće pogriješiti. Kasnije će biti objašnjeno kako dobiti huffmanove kodove sa ovakvim osobinama.

Sada imamo sve potrebne informacije za upis informacija u fajl. Prije nego što to izvršimo, dužni smo ostali reći kako se kodiraju i DC elementi.

Obzirom da je svaki DC element prvi u nizu koji dobijemo cik cak kodiranjem 8x8 bloka, to će vrijednost *x* uređenog para (*x*,*y*) uvijek biti jednaka 0 (jer ne postoji niti jedan element prije DC elementa koji je jednak nuli što je logično, jer je DC prvi element). Kako smo rekli da se uređeni par (*x*,*y*) kodira na 8 bita (4 odlazi na element *x* i 4 na element *y*) tada znamo da će gornja četiri bita uvijek biti jednaka nuli. Uzmimo sada uređeni par (0,-752) što je zapravo DC element 8x8 bloka navedenog na početku poglavlja, a kojeg smo dobili primjenom DCT→Cik Cak Kodiranje→RLC kodiranje. Isto kao i kod AC elemenata skoncentrirajmo se na element *y* uređenog para (*x*,*y*). U našem slučaju to je -752. Ako pogledamo u *Tabelu 2*, -752 pripada kategoriji 10. Dakle umjesto -752 pišemo broj 10, čime dobijamo uređeni par (0,A)($A_{16}=10_{10}$). Kako je broj negativan pronađimo njegov prvi komplement

$(0100001111)_{1k}$=-$(752)_{10}$. Dakle, postupak je potpuno isti kao kod kodiranja AC elemenata, s tim što je element *x* uređenog para (*x*,*y*) uvijek jednak nuli.

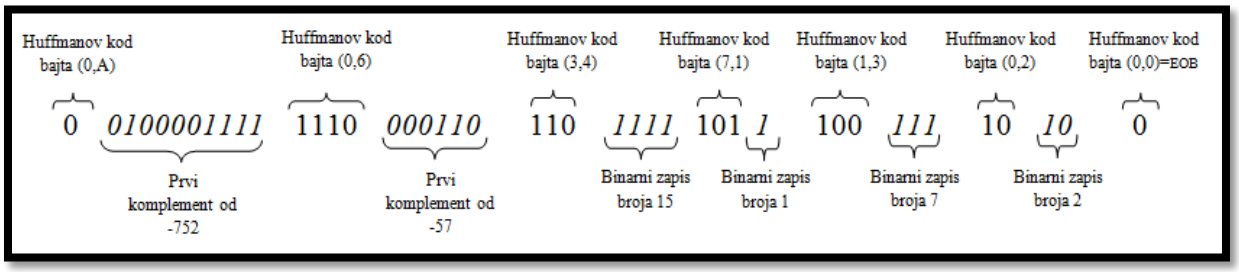
Isto kao i kod AC elemenata, uređenom paru (0,A) moramo dodijeliti huffmanov kod. JPEG standard nalaže da se za DC i AC elemente generiranje huffmanovih kodova mora praviti odvojeno, tj. prave se dvije „tabele“. Jedna za DC elemente, druga za AC elemente. Dakle luminance komponenta imat će dvije huffmanove tabele: jednu za DC i drugu za AC komponente. Chroma komponente C_b i C_r će imati zajedničke tabele za DC i AC elemente. Ovo može dovesti do toga da npr. DC i AC element luma komponente slike mogu imati isti huffmanov kod, što na prvi pogled može izgledati kao problem, međutim, dekoder je „svjestan“ da kad dekodira DC element, treba „gledati“ u huffmanovu tabelu za DC elemente, a isto tako kada dekodira AC elemente treba gledati u huffmanovu tabelu kojom su kodirani AC elementi, tako da ne može doći do greške.

Ovo napominjem iz razloga što ću u sljedećoj tabeli napisati huffmanov kod za uređeni par (0,A) kojim je zapravo kodiran naš DC element -752, a isti taj kod ima element (0,0)=EOB koji pripada huffmanovoj tabeli za AC elemente.

Uređeni par	***ASCII simbol uređenog para***	***Huffmanov kod***
(0,A)		**0**

Tabela 5. Huffmanovi kodovi za DC elemente

Sada je zaista sve spremno za upis podataka u fajl. Prvo se upisuje DC element 8x8 bloka, a zatim svi ostali AC elementi. Binarni zapis sa objašnjenjem prikazan je na slici ispod.



Slika 4. Izgled binarnog zapisa 8x8 bloka

Na ovaj način smo zapisali jedan 8x8 blok u fajl. Na isti način bi se upisao i bilo koji drugi blok. Iz gornjeg binarnog zapisa ***u potpunosti možemo dekodirati naš početni blok***. Ovdje ćemo to uraditi samo za dva elementa, a ostale elemente čitatelj/ka može uraditi za vježbu.

Dakle, imamo sljedeći niz bita: 001000011111110 000110110111110111001111010 0. Dekoder zna da prvo čita podatke za DC element. Uzima prvu nulu i provjerava u huffmanovu tabelu za DC elemente (to je *Tabela 5*). Dekoder zaključuje da je huffmanov kod 0 dodjeljen bajtu 0x0A. Uzmimo donja četiri bita (tj. element *y* uređenog para (*x*,*y*)=(0,A)). Kako je $A_{16}=10_{10}$ to dekoder zaključuje da je DC element kategorije 10, što znači da sljedećih 10 bita predstavlja pravu vrijednost našeg DC elementa. Sljedećih 10 bita je: 0100001111. Obzirom da je najznačajniji bit jednak 0, to slijedi da je ovo zapravo prvi komplement broja, pa da bismo dobili pravu vrijednost DC elementa, invertujmo sve cifre,

čime dobijamo binarni broj $(752)_{10}$=($1011110000)_2$. Obzirom da je broj bio zapisan u prvom komplementu, zaključujemo da je broj negativan. Dakle, vrijednost našeg DC elementa je -752. Pronašli smo vrijednost našeg DC elementa, sada trebamo naći AC elemente. Crvenim su označeni pročitani bitî: 0010000111111110 0001101101111101110011110100.

Dekoder čita jedinice sve dok ne naiđe na nulu (sjetimo se da ne smije postojati huffmanov kod sačinjen samo od jedinica). Prvi takav kod je 1110. Dekoder provjerava u huffmanovu tabelu za AC elemente (*Tabela 4*) da li postoji element kojemu je dodijeljen huffmanov kod 1110. Takav element postoji, i to je element 06h. Kako su gornja četiri biti jednaka nuli, to znači da ne postoji niti jedan AC element, koji prethodi ovom AC elementu, a čija je vrijednost 0 (da je vrijednost gornja četiri bita bila jednaka 2, to bi značilo da prije ovog AC elementa, postoje 2 AC elementa čija je vrijednost jednaka 0 i njih bi upisali u niz). Na osnovu donja četiri bita saznajemo da je ovaj AC element kategorije 6, pa uzmimo sljedećih 6 bita: 000110. Kako je vrijednost najznačajnijeg bita jednaka 0, to zaključujemo da je ovaj element napisan u prvom komplement kodu, a samim tim da je i negativan. Invertujmo sve cifre da dobijemo originalni binarni zapis. Dobili smo broj -57, što je zaista vrijednost prvog AC elementa. Crvenim su označeni pročitani bitî: 0010000111111100 0011011011111011100111101000. Istim postupkom možemo dekodirati i sve ostale AC elemente čime ćemo dobiti niz od 64 elementa. Da bismo dobili početni 8x8 blok, nad dobijenim nizom treba primijeniti inverzno cik cak kodiranje.

Ovdje je samo ukratko objašnjen postupak JPEG kodiranja, u kojem smo izostavili generiranje huffmanovih kodova što predstavlja još jedan dodatni korak. Proces se dodatno usložnjava ukoliko koristimo chroma subsampling 2x1, 1x2 ili 2x2, ali o tome ovdje nećemo pisati jer bi nam oduzelo previše prostora.

Zadatak 4.2. *Prepravite vaše funkcije dct() i idct() tako da koriste RLC po cik-cak aranžmanu.*

*Obratite pažnju da nakon ove izmjene funkcija dct() više neće vraćati matricu dimenzija XxYx2 nego jednodimenzionalni niz dimenzije Z gdje je Z=X*Y*2. Pošto je dozvoljeni opseg vrijednosti -127 do 127, uvedite da je vrijednost EOB=-128. Ali također imajte na umu i probleme sa greškom zaokruživanja i gamutom, tako da se trebate prethodno pobrinuti da se vrijednosti svedu u dozvoljeni opseg.*

Matlab kod dobivenih funkcija upišite u prostor ispod:

Huffmanovo kodiranje

Već smo vidjeli skoro sve tehnike koje se koriste prilikom JPEG kompresije. Jedna od tehnika koja se koristi prilikom kompresije, a koju do sada nismo objasnili je huffmanovo kodiranje. JPEG kompresija ne koristi tehniku huffmanovog kodiranja koju smo obradili u jednom od prethodnih vježbi, a ako se pitate zašto, odgovor je jednostavan. Huffmanovi kodovi koje se generiraju „šetanjem“ kroz grane huffmanovog stabla, koje smo generirali na jednoj od prethodnih vježbi, ne zadavoljavaju niti jedno od dva pravila koja smo već naveli, što se lako možete uvjeriti. Dovoljno je da imate dva elementa/lista čije ćete frekvencije spojiti u jedan čvor koji postaje korijen stabla. Kada generirate huffmanove kodove, jedan element će imati kod 0, a drugi 1. Obzirom da ne smije postojati huffmanov kod koji sadrži sve jedinice, jasno je da su pravila narušena. Ovo zapravo nije posljedica samog stabla, obzirom da stablo obrazujemo na osnovu statističkih podataka, odakle slijedi da se prilikom JPEG kompresije koriste nešto drugačije statističke procedure (opisane u Aneksu C i K JPEG standarda), od procedura koje smo koristili u vježbi o huffmanovoj kompresiji. Na osnovu statističkih podataka , generira se huffmanovo stablo, a samim time i huffmanovi kodovi za elemente stabla (tehnički rečeno, prilikom implementacije JPEG enkodera/dekodera stablo se ne mora čak ni generirati, ali o tim detaljima ukratko ćemo reći nešto kasnije). Kao što ćemo vidjeti, ovako generirano huffmanovo stablo mora zadovoljavati određene osobine, a upravo zbog njih je moguće generirati huffmanove kodove koji zadovoljavaju oba pomenuta pravila.

Proces generiranja huffmanovih kodova je nešto jednostavniji prilikom implementacije JPEG dekodera, pa ćemo taj proces ovdje i objasniti. Svaki JPEG fajl u svojoj strukturi sastoji se od niza *markera* (specijalan niz karaktera koji logički dijeli JPEG fajl na cjeline). Svaki marker započinje FF bajtom, iza čega slijedi novi bajt koji označava o kojoj vrsti markera je riječ. Tako npr. JPEG fajl počinje FFD8 markerom, a završava FFD9 makerom. Kvantizacijske tabele definirane su unutar FFDB markera, SOS (*start of scan maker*) definiran je unutar FFDA markera. *Podaci potrebni za generiranje huffmanovih tabela pohranjeni su unutar FFC4 markera*. Da je zaista tako, JPEG fajl otvorite pomoću nekog *Hex editora* i potražite navedene markere. Svaki FFC4 marker ima sljedeći izgled (ako ćemo baš ići u detalje, ovo je pojednostavljena šema FFC4 markera, no bez obzira na to, ako bismo unutar JPEG fajla koristili ovaj dizajn, fajl bi bio sasvim ispravan):

DHT	L_h	T_c	T_h	L_1	L_2	…	L_{16}	Elementi
16 bita	16 bita	4 bita	4 bita	8bita	8 bita		8 bita	8 bita

Polje	Opis
DHT	*Define Huffman Table (FF C4)marker*
L_h	*Veličina huffmanove tabele u bajtima*
T_c	*Klasa huffmanove tabele. 0-DC tabela 1-AC tabela*
T_h	*Table ID – određujemo da li se radi o Luma ili Chroma tabeli*
L_n	*Broj huffmanovih kodova duljine n bita*

Tabela 6. Značenje poljâ FFC4 markera

Sad za sad značenje pojedinih polja možda nije jasno, ali u nastavku ćemo pokušati otkloniti sve moguće nejasnoće. Krenimo od jednog običnog JPEG fajla. U razmatranje ćemo uzeti neku sliku malih dimenzija, i ne baš previše boja, kako bi bilo što manje huffmanovih elemenata. Sasvim lijepo može poslužiti sljedeća slika:

Slika 5. Testna slika

Kada navedenu sliku otvorimo pomoću hex editora, te pronađemo FFC4 marker, dobijamo sljedeći sadržaj (ovisno od programa pomoću kojeg ste pohranili sliku, sadržaj vašeg FFC4 markera možda i neće biti ovakav, ali sljedeći primjer vam može pomoći u razumijevanju):

DHT	L_h	T_c	T_h	$L_1 - L_{16}$	Elementi
FF C4	00 1F	0	0	00 01 05 01 01 01 01 01 01 00 00 00 00 00 00 00	00 01 02 03 04 05 06 07 08 09 0A 0B

Svaki, pa tako i ovaj DHT marker počinje FF C4 bajtom. U polje T_h upisana je vrijednost 00 1F. Kako navedeno polje definira veličinu huffmanove tabele, slijedi da ova tabela zauzima 31 bajt (ako prebrojite bajte krećući od polja L_h trebali bi prebrojati 31 bajt. Primijetite da FF C4 marker ne ulazi u veličinu tabele). Na osnovu poljâ T_c=0 i T_h=0 zaključujemo da se radi o luminance huffmanovoj tabeli za DC elemente. Sada slijedi najzanimljiviji dio. Polje L_1 ima vrijednost 00, polje L_2 vrijednost 01, polje L_3 05 itd… i šta zapravo to znači? Općenito, L_n oznaka označava ***broj huffmanovih kodova kodiranih sa n bita***. Ako je tako, onda slijedi da oznaka L_1 označava broj huffmanovih kodova kodiranih sa jednim bitom. Očito nema takvih kodova obzirom da je vrijednost L_1=00. Broj huffmanovih kodova kodiranih sa dva bita L_2 je 01. Dakle, imamo samo jedan element koji je kodiran sa dva bita. Ako se pitamo koji je to element, onda je to lako, uzmimo prvi element iz polja *elementi*. Dakle *element 00* je kodiran sa dva bita (ako se sjetimo ovo je naš EOB element, koji je kodiran sa 2 bita što ima smisla obzirom da se EOB pojavljuje skoro u svakom 8x8 bloku, a nešto kasnije ćemo odrediti koja su to dva bita). Vrijednost polja L_3 je 05, što znači da imamo 5 elemenata koji su kodirani sa 3 bita. To su elementi 01, 02, 03, 04, 05. Polje L_4 ima vrijednost 01, odakle slijedi da je samo jedan element kodiran sa 4 bita, a to je element 06 i tako dalje sve do polja L_{16} koje je jednako 0 odakle slijedi da nema elemenata koji su kodirani huffmanovim kodom od 16 bita.

Ako poštujemo gore opisanu proceduru, veoma lako možemo generirati sljedeću tabelu:

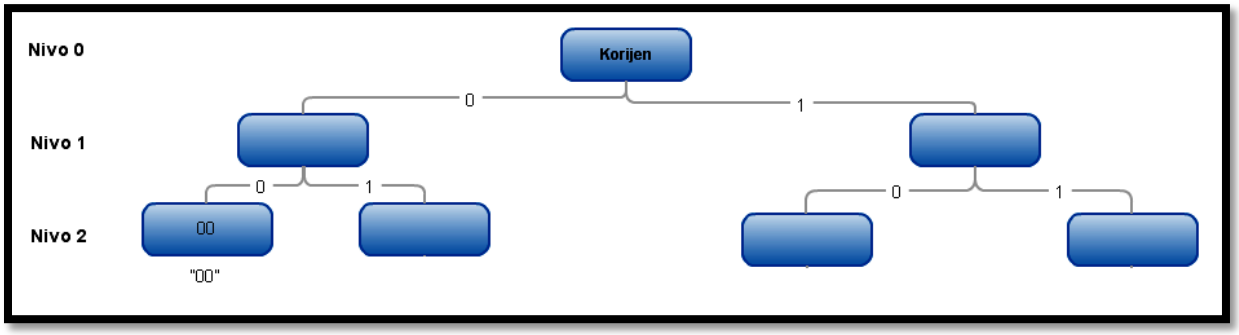
Broj huffmanovih kodova dužine 1 bit	00	
Broj huffmanovih kodova dužine 2 bita	**01**	**00**
Broj huffmanovih kodova dužine 3 bita	**05**	**01,02, 03, 04, 05**
Broj huffmanovih kodova dužine 4 bita	**01**	**06**
Broj huffmanovih kodova dužine 5 bita	**01**	**07**
Broj huffmanovih kodova dužine 6 bita	**01**	**08**
Broj huffmanovih kodova dužine 7 bita	**01**	**09**

Broj huffmanovih kodova dužine 8 bita	**01**	**0A**
Broj huffmanovih kodova dužine 9 bita	**01**	**0B**
Broj huffmanovih kodova dužine 10 bita	00	
Broj huffmanovih kodova dužine 11 bita	00	
Broj huffmanovih kodova dužine 12 bita	00	
Broj huffmanovih kodova dužine 13 bita	00	
Broj huffmanovih kodova dužine 14 bita	00	
Broj huffmanovih kodova dužine 15 bita	00	
Broj huffmanovih kodova dužine 16 bita	00	

Tabela 7. Generirana tabela na osnovu podataka iz FFC4 markera

Do sada smo pronašli o kojim elementima je riječ, s koliko bita su oni kodirani, ali još nismo odredili huffmanove kodove za pojedine elemente. To možemo učiniti tako da generiramo huffmanovo stablo (kasnije ćemo pokazati i puno brži način i to bez generiranja stabla).

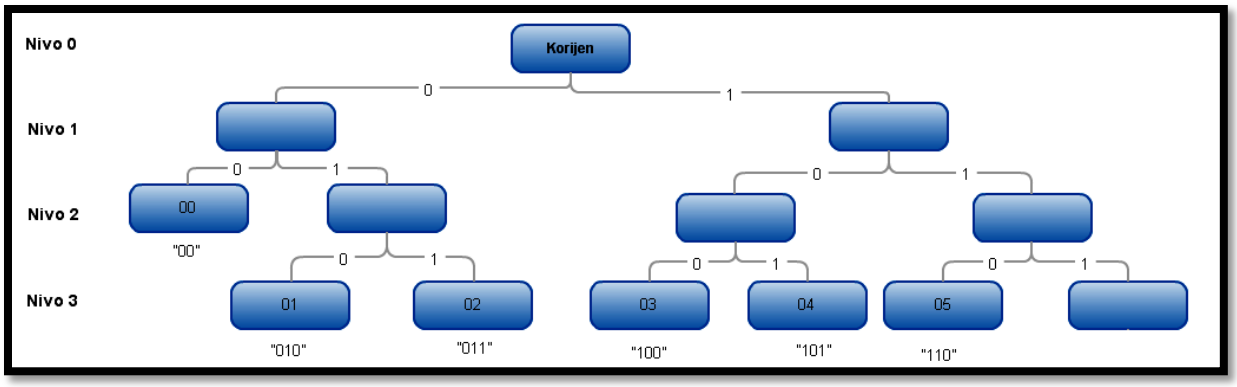
Da bismo kreirali huffmanovo stablo, krenimo od korijena (za razliku od generiranja huffmanovog stabla iz jedne od prethodnih vježbi, kada smo započinjali od listova). Korijen ima dva djeteta, *lijevo* i *desno* dijete, koja se nalaze na *Nivou 1*. Broj nivoa odgovara broju bita koji su potrebni da bi se djeca binarno predstavila (broju bita u huffmanovom kodu). U našem slučaju, niti jedan element nije kodiran sa jednim bitom (kao što možemo vidjeti u *Tabeli 7*), zbog čega i *lijevo* i *desno* dijete korijena dobijaju vlastitu djecu, kako je to prikazano na sljedećoj slici:



Slika 6. Izgled huffmanovog stabla nakon dvije iteracije

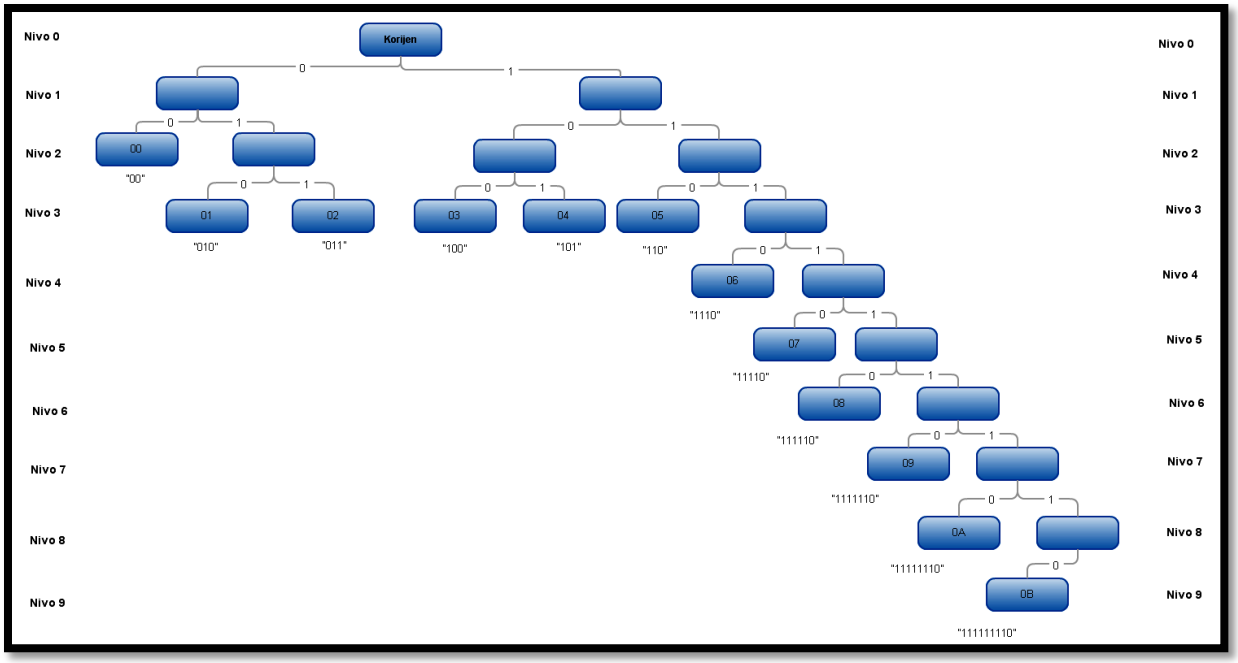
Ako pogledamo u *Tabelu 7*, vidimo da postoji jedan element koji je kodiran sa dva bita (to je razlog zbog čega se nalazi i na *Nivou 2*) i to element 00. Zbog toga, *lijevo dijete*, lijevog djeteta korijena dobija vrijednost *00* i on više ne može imati djece. Obzirom da je element 00 jedini element kodiran sa dva bita, sva ostala djeca na *Nivou 2* dobijaju vlastitu djecu, kako je to prikazano na *Slici 7.*

Ako pogledamo *Tabelu 7*, tada vidimo da postoji 5 elemenata koji su kodirani sa tri bita, a to su sljedeći elementi: 01, 02, 03, 04, 05. Na *Nivou 3* samo jednom djetetu nije dodijeljena vrijednost, te iz tog razloga on dobija vlastitu djecu baš kako je to prikazano na *Slici 7.*



Slika 7. Izgled huffmanovog stabla nakon treće iteracije

Procedura već sada postaje kristalno jasna, i ako ste sve radili kako treba, trebali biste dobiti stablo sljedećeg izgleda:



Slika 8. Izgled huffmanovog stabla

Sada veoma jednostavno možemo generirati huffmanove kodove za sve elemente, tako što ćemo svim *lijevim granama* dodijeliti kod 0, a *desnim granama* kod 1. Ukoliko želimo naći huffmanov kod za element *02*, tada krenimo od korijena, prateći grane sve do elementa zaključujemo da je ***huffmanov kod elementa 02 jednak 011.*** Na ovaj način možemo dobiti huffmanove kodove svih elemenata u stablu što je prikazano u sljedećoj tabeli:

Broj kodova duljine 2 bita	Element (heksadecimalno)	Huffmanov kod (binarno)
1	**00**	**00**
Broj kodova duljine 3 bita	Element (heksadecimalno)	Huffmanov kod (binarno)
5	**01**	**010**
	02	**011**
	03	**100**
	04	**101**
	05	**110**
Broj kodova duljine 4 bita	Element (heksadecimalno)	Huffmanov kod (binarno)
1	**06**	**1110**
Broj kodova duljine 5 bita	Element (heksadecimalno)	Huffmanov kod (binarno)
1	**07**	**11110**
Broj kodova duljine 6 bita	Element (heksadecimalno)	Huffmanov kod (binarno)
1	**08**	**111110**
Broj kodova duljine 7 bita	Element (heksadecimalno)	Huffmanov kod (binarno)
1	**09**	**1111110**
Broj kodova duljine 8 bita	Element (heksadecimalno)	Huffmanov kod (binarno)
1	**0A**	**11111110**
Broj kodova duljine 9 bita	Element (heksadecimalno)	Huffmanov kod (binarno)
1	**0B**	**111111110**

Tabela 8. Huffmanovi kodovi elemenata

Primijetite da huffmanovo stablo na *Slici 8* ima određene osobine. Prvo što smo uočili jeste da se ovo stablo kreira krećući od korijena, a ne od listova. Drugo, na svakom nivou postoji barem jedan čvor koji ima djecu (u suprotnom bi postojao huffmanov kod koji sadrži sve jedinice). Ako na nekom nivou postoji samo jedan čvor koji ima djecu, onda je to prvi čvor s desne strane na posmatranom nivou (npr. *Nivo 3*). Ako imate dobro oko i nešto znanja iz matematike i binarne aritmetike, veoma lako možete uočiti činjenicu da ***svi susjedni elementi na nekom nivou imaju huffmanove kodove koji se razlikuju za 1 binarno*** (npr. analizirajte hufmanove kodove za elemente nivoa 3)***. Prvi čvor sljedećeg nivoa kojemu je dodijeljen element, ima huffmanov kod koji je za jednu cifru duži od prethodnog, pri čemu je vrijednost njegovog huffmanovog koda jednaka vrijednosti posljednjeg huffmanovog koda prethodnog nivoa, uvećana za jedan, a zatim šiftana jedno mjesto lijevo (što je ekvivalentno množenju sa 2). Uz poznavanje ovih činjenicâ, generiranje huffmanovog stabla čak nije ni potrebno.***

Navedena pravila se veoma lako mogu provjeriti kroz *Tabelu 8.* Na nivou 2, samo jednom od čvorova je dodijeljen element 00h, a huffmanov kod tog elementa je $(00)_2$. Obzirom da niti jednom preostalom čvoru na nivou 2 nije dodijeljen element, prelazimo na nivo 3. Na nivou 3, pet čvorova dobilo je svoje elemente, a to su elementi: 01, 02, 03, 04, 05. Huffmanov kod čvora kojemu je dodijeljen element 01h određujemo tako da uzmemo huffmanov kod posljednjeg elementa prethodnog nivoa, proširimo ga za jednu cifru, njegovu vrijednost uvećamo za 1, a zatim rezultat šiftamo jedno mjesto lijevo. Jedini element nivoa 2 je element 00h, čiji je huffmanov kod $(00)_2$. Proširimo ovaj kod sa još jednom cifrom,

uvećajmo ga za jedan, a zatim dobiveni rezultat šiftajmo za jedno mjesto u lijevo. 000+1=001<<1=010, što predstavlja huffmanov kod elementa 01h.

Obzirom da susjedni elementi *istog nivoa* imaju huffmanove kodove koji se razlikuju za 1, to ćemo huffmanov kod elementa 02h dobiti tako što ćemo huffmanov kod elementa 01h uvećati za jedan. Dakle huffmanov kod elementa 02h je $(011)_2$. Na potpuno isti način možemo generirati huffmanove kodove svih ostalih elemenata stabla.

Na programeru je odabrati koju od gore dvije opisane metode će koristiti (naravno, programer može koristiti i neku sasvim treću metodu, nitko mu to ne brani). Zapravo JPEG standard nas ne obavezuje na koji način ćemo doći do huffmanovih kodova, na koji način ćemo npr. izvršiti FDCT ili bilo koju proceduru JPEG kompresije, i tu ostavlja potpunu programersku slobodu. Jedino što standard zahtjeva jeste da se kodovi zaista generiraju, i to da zadovoljavaju već prethodno spomenuta dva pravila, što huffmanovi kodovi koje smo generirali gore opisanim načinima (generiranje stabla ili analitički način) svakako zadovoljavaju (ako provjerite, niti jedan kod nije sadržan u drugom u smislu kako smo to definirali u poglavlju „Kodiranje AC elemenata“, niti je ijedan kod sačinjen samo od jedinica).

Za kraj, ako se baš pitate zašto huffmanov kod ne smije sadržavati sve jedinice, onda je odgovor sljedeći. Neka imamo huffmanov kod čije su sve cifre jedinice npr. $(111)_2$. JPEG standard nalaže da ukoliko nemamo dovoljno bita da kreiramo čitav bajt, preostale cifre bajta popunjavamo prosto jedinicama. Dakle, neka u fajl upisujemo niz bita: 01110000 110. Obzirom da je ovaj podatak dugačak 11 bita, preostali biti trebaju biti jedinice (dakle dodajemo 5 jedinica). U fajl upisujemo dva bajta: 01110000 11011111 gdje su crvenim označeni „dodani“ *bitovi podataka i oni ne čine originalni dio informacije i njih prilikom dekodiranja prosto treba ignorirati*. Kada bi dekoder generirao jedan huffmanov kod koji se sadrži sve jedinice, kao što je u našem slučaju kod $(111)_2$, tada bi se desilo sljedeće: dekoder bi ispravno dekodirao bite originalne informacije 01110000 110, zatim bi, umjesto da ignorira preostale bite, pročitao tri jedinice 111 koje bi zatim povezao sa huffmanovim kodom $(111)_2$ i napravio pogrešku, obzirom da 111 zapravo ne predstavlja dio originalne informacije.

U ovom tekstu samo smo načeli temu JPEG kompresije, ali smo izložili osnovnu ideju i principe kompresije, i svatko tko prođe ovaj tutorijal sa razumijevanjem ne bi trebao imati implementacijskih poteškoća, naravno uz razmatranje neke dodatne literature.