

Codierung

Klaus Kusche

Inhalt

- **Zahlen**
- **Zeichen und Zeichensätze**
- **Komprimierung**
- **AV-Daten**

Ziel

- Zahlen:
 - Verständnis der Tücken und Fehlerquellen
- Zeichen:
 - Wissen: Unicode & Darstellung
 - Kenntnis der Varianten
incl. Vor- und Nachteilen
- Komprimierung, AV-Daten:
 - Verständnis des Funktions-Prinzips

Voraussetzungen

Grundkenntnisse der Zahlendarstellung:

- Binärdarstellung, Zweierkomplement
- IEEE-Gleitkommadarstellung

Grundkenntnisse der Zeichendarstellung:

- ASCII

Int: Darstellung

Binärzahl,
bei negativen Zahlen
Zweierkomplement

(Effekt Zweierkomplement siehe Tafel, Grafik)

==> Vorderstes Bit ist Vorzeichen

==> Letztes Bit ist gerade / ungerade
(auch bei neg. Zahlen!)

==> * / % mit Zweierpotenzen
als Bit-Operation implementierbar

Int: Alternative Darstellungen

- *Einerkomplement*
- *Sign/Magnitude*

Aber:

- Unpraktischer, umständlicher zu implementieren
- Symmetrischer Wertebereich, aber getrennte Bitmuster für $+0$ und -0

==> In der Praxis kaum verwendet

Int: signed / unsigned (1)

“signed/unsigned” ist reine Interpretationssache:

- Gleiche Darstellung im Speicher
- Gleiche Rechenoperation + und –
- Bei Typ-Umwandlung signed \Leftrightarrow unsigned
ändert sich an den Bits im Speicher nichts!

Unterschied nur

- bei * / % (und Vergleich)
- bei Längen-Vergrößerung:
Zero-Extend oder Sign-Extend?

Int: signed / unsigned (2)

==> Bei Umwandlung unsigned ==> signed
(und bei Längen-Kürzungen):

Große Werte werden zu negativen Zahlen

==> Bei Umwandlung signed ==> unsigned:

Es wird nicht das Vorzeichen weggelassen!

Beispiel **-1**:

Wird nicht zu **+1**,

sondern zur größtmöglichen positiven Zahl

Int: Fehler und Überlauf

- ... / 0 und ... % 0 : Harter Programmabbruch
- Alle Überläufe:

Keine Meldung, es wird
“*im Kreis herum*” normal weitergerechnet!

Nach 2147483647 kommt -2147483648 und umgekehrt!
Auch zu große Konstanten werden stillschweigend “passend gemacht”!

Für Techniker:

Nur die hinteren 32 Bits des Ergebnisses werden gespeichert,
das Überlauf-Bit bzw. alles davor wird ignoriert.

Für Mathematiker:

Es wird so lange 2^{32} dazu- oder weggezählt, bis das Ergebnis
in $-2^{31} \dots 2^{31}-1$ bzw. $0 \dots 2^{32}-1$ liegt (d.h. Rechnung mod 2^{32}).

Int: % (Rest)

Achtung bei negativen Zahlen!

- Vor C99 war % für negative Zahlen undefiniert:
-10 % 6 durfte 2 oder -4 liefern!
 - Seit C99 ist das negative Ergebnis definiert,
d.h. % entspricht nicht dem “mod”-Operator in Mathe!
 - Leider würde man fast immer das **mod**-Verhalten brauchen
(d.h. ein garantiert positives Ergebnis, z.B. als Index!),
aber das gibt es in C nicht als fertigen Operator...
- ==> Am besten kein % für negative Zahlen verwenden,
wenn doch: *Aufpassen!*

Int: Tücken (1)

- int / int ergibt int

Ergebnis wird abgeschnitten!

Z.B. $3/4$ ergibt 0

- Achtung bei unärem Minus:

Kleinste neg. Zahl hat kein positives Gegenstück!

if (a < 0) { a = -a; }

Z.B. für 16 Bit a: -32768 bleibt -32768

Int: Tücken (2)

- Achtung bei unsigned-Bedingungen

for (i = max; i >= 0; --i) { ...

ist für unsigned **i** eine Endlosschleife!

- “Gemischte” Rechnungen:

Signed-Werte werden als unsigned betrachtet

=> *gerechnet wird unsigned!*

- Unterschied signed / unsigned bei >>
(Bit-Shift nach rechts)!

Int: Länge (1)

Länge und Wertebereich sind im C-Standard

nicht festgelegt!

Einzigste Garantie:

$$\begin{aligned} \text{sizeof(char)} &\leq \text{sizeof(short)} \leq \text{sizeof(int)} \\ &\leq \text{sizeof(long)} \leq \text{sizeof(long long)} \end{aligned}$$

Realität:

Implementierungs-abhängig: **short / int / long / long long**

Windows/Linux 32 bit: 2 / 4 / 4 / 8 Bytes = 16 / 32 / 32 / 64 Bits

Windows/Linux 64 bit: 2 / 4 / 8 / 8 Bytes = 16 / 32 / 64 / 64 Bits

Mikro-Kontroller: 1 oder 2 / 2 / 4 / - Bytes = 8 oder 16 / 16 / 32 / - Bits

Int: Länge (2)

==> Wenn fixe Längen-Anforderung:

Spezialtypen verwenden, nicht **int** !

- int's fixer Länge laut C-Std. (aus **stdint.h**):
int8_t, int16_t, int32_t, int64_t
(analog **uint8_t, ...**)
- Parameter- / Returntyp für Größen & Längen:
size_t
- Andere Spezialtypen: **time_t, pid_t, ...**

Int: Länge (3)

- Niemals Pointer auf **int** (oder long) und zurück casten!!!

Teile der Pointer-Daten gehen verloren, wenn **int** und Pointer verschieden lang sind!

==> Resultierender Pointer ist ungültig!

==> **ptrdiff_t** verwenden

(besser: Vermeiden!!!)

Int: Länge (4)

- Niemals Pointer casten
 - von **int*** auf **char*** oder **short***
 - von **long*** auf **int*** oder **short***

und dann Wert lesen/schreiben

*Funktioniert nur auf Little-Endian-CPU's,
nicht auf Big-Endian-CPU's!*

(warum???)

Double: IEEE-Darstellung

- *Sign/Magnitude*

=> es gibt $+0$ und -0 , Unterschied z.B. bei $/0$

- Als *Gleitkomma-Binärzahl* (Zweierpotenz):

$1.XXX * 2^y$ (mit diversen Offsets & Tricks)

=> *Rundungs-Verluste!*

Z.B.: 0.1 dezimal ergibt

unendlich periodische Binär-Komma-Zahl!

- *Sonderwerte* für **+inf**, **-inf**, **nan**

(+ Funktionen **isinf**, **isnan**, ...)

Double: Fehlerbehandlung

- **Default:**
Rechnet bei Fehlern (z.B. / \emptyset , neg. Wurzel, ...) ganz normal mit **inf** oder **nan** weiter
- **inf** und **nan** pflanzt sich stillschweigend fort!
- Überlauf wird stillschweigend **inf**
Unterlauf wird \emptyset
- **Option:**
Abbruch bzw. Aufruf von Fehler-Funktionen

Double: Rundung (1)

- **4 Rundungs-Modi** wählbar:
 - *To nearest* (default)
 - *To 0*
 - *Down* (to neg. infinity)
 - *Up* (to pos. infinity)
- Zuweisung & Typumwandlung
double ==> **int**
schneidet immer ab,
unabhängig vom Rundungsmodus!
==> Funktionen **round**, **lround**

Double: Rundung (2)

- Vorsicht bei ==:
“trifft” wegen Rundungsfehler ev. nicht genau!
=> Delta-Vergleich machen!
- Genauigkeit hängt auch von der Rechen-Reihenfolge ab (vor allem bei langen Summen usw.)!
- Ein **inf** ist normal vergleichbar, aber ein **nan** vergleicht immer “falsch”
=> es gilt nicht immer < oder == oder > !

Double: Intervall-Rechnung

Z.B. bei phys. Berechnungen,
wenn Genauigkeits-Abschätzung
des Ergebnisses nötig ist:

- Zwei double pro Zahl
für obere & untere Grenze
=> "Echtes" Ergebnis liegt sicher dazwischen
- Jede Rechnung zwei Mal
mit gegensätzlichen Rundungs-Modi machen

BCD-Zahlen

“Binary Coded Decimal”

==> Darstellung und Rechnung
im Zehner-System
(in Software)

- Für kaufmännische Rechnungen
- 4 Bits / Ziffer (0 ... 9)
- *Sign/Magnitude, Fixkomma*

Beliebig lange Zahlen (1)

- Meist Array oder Liste normaler **int**'s
=> Eine "Ziffer" ist ein **int**,
d.h. z.B. ein Wert 0 ... 4294967295
- Dann "ziffernweise" Rechnung,
wie händisch im Zehner-System
- Oft Sign / Magnitude,
nicht Zweierkomplement

Beliebig lange Zahlen (2)

Zum Rechnen braucht man:

- Für +, – und Vergleich:
1 Bit mehr oder trickreiche Prüfungen
- Für *, / und % :
Doppelte Genauigkeit (zwei “Ziffern”)
(z.B. “Ziffern” sind 32 bit **int**
=> Rechnungen in 64 bit **long** machen)

=> Ev. in Assembler programmieren!
(Carry-Flag nutzen!)

Char: signed / unsigned ?

Der Standard lässt dem Compiler freie Wahl, ob der “normale” **char** signed oder unsigned ist!

Auf Intel ist **char** fast immer signed char!

Bei Umlauten & Sonderzeichen, bei Unicode:

==> Die Umwandlung in **int**
kann negative Zahlen liefern

==> Eine Verwendung von **char** als Array-Index
ist eine ganz schlechte Idee!

Zeichensätze (1)

Ursprünglich:

- ASCII (seit “ewig”: 1963):
 - Nur 7 Bit: 0 ... 127 (==> signed/unsigned *ist egal!*)
 - “*US-Zeichensatz*”:
 - Keine Umlaute oder nationalen Sonderzeichen
 - 0...31 & 127: Steuerzeichen (z.B. \n, \t, ...)
 - 32: Zwischenraum
 - 33-126: Sichtbare Zeichen
- EBCDIC (noch früher, IBM Großrechner):
heute kaum noch verwendet

Zeichensätze (2)

“Nationale” 8-Bit-Zeichensätze:

- *“Wild gewachsen”, je nach Land verschieden!*
- Haben nur *max. 256 Zeichen!*
- 0...127 = ASCII
128...255 = Sonderzeichen je nach Land
- Hersteller-spezifisch: z.B. ***MS-Codepages***
Westeuropa: CP 850 (DOS), CP 1252 (Windows)
- Standardisiert: ***ISO 8859-x***
Westeuropa:
ISO Latin 1 = ISO 8859-1, ISO Latin 9 = ISO 8859-15 (mit €)

Zeichensätze (3)

Vorteile:

- Typ **char**, 1 Zeichen = 1 **char** (Platz-effizient!)
- “Normaler” Code, leicht zu programmieren: bekannte String-Funktionen, printf, ...

Nachteile:

- Gespeicherte Daten nicht portabel auf andere Zeichensätze / in andere Länder ==> Es werden “falsche” Zeichen angezeigt!
- Max. 128 (bzw. 95) zusätzliche Zeichen

Unicode (1)

- Jedes Zeichen auf der Welt soll eine **weltweit eindeutige Nummer** bekommen
- Derzeit über 100000 Codes definiert, wächst, über 1 Mio. ($17 * 2^{16}$) möglich:
Chinesisch, arabisch, mathemat. Symbole, ...
Auch Emojis, Musik-Noten, Ligaturen, ...
- Definiert seit 1993, hersteller-unabhängig:
In **ISO 10646** und vom **Unicode-Konsortium**
- Wieder gilt: $0..127$ ist ident zu ASCII !

Unicode (2)

Problem:

Darstellung der Codes im Speicher
(haben *nicht in 1 Byte* Platz!)

2 wichtige Standards, konkurrierend:

- UTF-8
- UCS-2 bzw. UTF-16

=> *Dieselben Codes* (z.B. € = $20AC_{16} = 8364_{10}$)
werden *völlig unterschiedlich gespeichert!*

UTF-8

Wo:

- Unix & Linux, Mac
- Internet: UTF-8 = **RFC 3629**
- XML

Wie:

- Byteweise, normaler Typ **char**
- Aber variabel lange Codierung:
1 Zeichen kann 1 bis 4 **char**'s Platz belegen

UTF-8, Codierung (1)

- 0...127, 7 Bits: 1 Byte 0xxxxxxx
 - ==> nur-ASCII-Dokumente in UTF-8 sind ident zu “altem” ASCII, altes ASCII ist ohne Konvertierung gültiges UTF-8
 - ==> alle “wichtigen” Zeichen sind “platzsparend” dargestellt
- Alle Zeichen ≥ 128 sind **Multi-Byte**:
 - Alle Bytes der Codierung sind ≥ 128
 - Erstes Byte legt die **Länge** fest:
 - 128-2047, 11 Bits: 2 Bytes 110xxxxx 10xxxxxx
 - 2048-65535, 16 Bits: 3 Bytes 1110xxxx 10xxxxxx 10xxxxxx
 - 65536-1114111, 21 Bits: 4 Bytes 11110xxx 10xxxxxx 10xxxxxx 10xxxxxx
 - ==> Platz für gesamten Unicode
- Fortsetzbar bis 8 Byte, 42 Bit Nutzdaten (> 4 Billionen Zeichen)

UTF-8, Codierung (2)

- Platzbedarf im Vergleich zu UCS-2 / UTF-16:
Normaler Text ist viel effizienter,
Schriftzeichen-Text ist etwas ineffizienter
- UTF-8 ist byteweise richtig sortierbar
- Der Anfang eines Multi-Byte-Zeichens ist
auch von hinten / von mittendrin feststellbar:

0xxxxxxx ... Single Byte

10xxxxxx ... Multibyte Folgebyte

11xxxxxx ... Multibyte Startbyte

==> UTF-8 ist auch von hinten durchsuchbar

UTF-8, Codierung (3)

Es gibt ungültige Zeichenfolgen:

- Fehlende Folgebytes,
Folgebytes ohne Startbytes
- Redundante Codierungen
(z.B. Zeichen < 128 als 2, 3 oder 4 Byte)
- Startbytes für 5, 6, 7, 8-Byte-Folge
- Vom Standard verbotene Codes, u.a. **D800-DFFF**
(historisch aus UTF-16-Codierung bedingt,
kommen leider fälschlicherweise sehr oft vor!)

==> Ersetzt durch / angezeigt als Fehlerzeichen FFFD

UTF-8 vs. ISO 8859

Anzeige mit falschem Zeichensatz:

- Als Unicode gespeichert,
als 8 Bit ISO Latin angezeigt:
=> Statt jedem Umlaut
(= 2-Byte-Unicode-Sequenz)
werden 2 Sonderzeichen angezeigt
- Als ISO Latin gespeichert,
als Unicode angezeigt:
=> Jeder Umlaut ist ungültiger UTF-8-Code,
wird als ein "Fehlerzeichen" angezeigt

UTF-8: Programmierung

- Speicherung und Verarbeitung “ganz normal”:
 - Stringende in C ist nach wie vor $\backslash 0$ -Byte
 - strcpy, strcat usw. funktionieren wie bisher
- Änderungen nur
 - bei Ein- und Ausgabe
 - überall, wo es auf die “sichtbare Länge” ankommt
 - bei Deklarationen (mehr Platz reservieren!!!)

Weil “Länge in Bytes / **char**’s bis zum $\backslash 0$ ”
ist ungleich “Anzahl der sichtbaren Zeichen”

UCS-2

Wo früher:

- Microsoft Windows & Office
- Java & viele GUI-Toolkits

*Alt, abgekündigt, ersetzt durch Nachfolger **UTF-16***

Wie:

- Fix 2 Bytes pro Zeichen
- Ende-Markierung = Doppel-Null-Byte
(denn ein UCS-2-String
enthält viele Einfach-Null-Bytes!)

UCS-2, Programmierung

Code-Logik bleibt unverändert,
nur der Typ ändert sich von **char** auf

wchar_t (= “wide character”)

==> Da C kein Overloading kennt:

Alle Funktionen müssen umbenannt werden:
fwprintf, fgetwc, wcscpy, wcscmp, wcslen, ...
(header **wchar.h**)

Theoretisches Problem von **wchar_t**:

Größe im Standard nicht definiert, könnte auch 1 oder 4 Bytes sein!

==> besser **char16_t** verwenden?!

UCS-2, Nachteile (1)

- Viel Platzverschwendung (fast 50 %) bei “normalen” Texten:
Vorderes Byte fast aller Zeichen ist 0
(effizient nur bei Texten mit vielen Schriftzeichen)
- Reicht nicht für gesamten Unicode
(nur bis 65535 = max. 16-Bit-Zahl)
=> *Manche Sonderzeichen nicht darstellbar!*
- Völlig inkompatibel mit ASCII- und UTF-8-Strings:
Explizite Umwandlungsfunktionen nötig!

UCS-2, Nachteile (2)

Bei externer Speicherung, Netz-Übertragung, ...:

Endian-abhängig!

==> ***BOM*** als erstes Zeichen speichern / schicken
“***Byte Order Mark***” = unsichtbares Zeichen FFFF

Weil: FFFE (FFFF verkehrt) ist illegaler Code

==> wenn FFFE als erstes Zeichen gelesen wurde:
Alle Byte-Paare beim Lesen umdrehen!

- Intel, ARM, ... ist *little endian*, Standard sagt “*Default Big Endian*”?!
- Am Internet: Meist kein BOM,
sondern explizite Charset-Angabe utf-16le oder utf-16be

UTF-16

Wie:

- Codierung variabler Länge:
1 oder 2 Doppel-Bytes (**wchar_t**'s) pro Zeichen
- Code bis 65535 ==> direkt gespeichert
- Code ab 65536:
Zuerst 65536 abziehen (==> max. 20 Bits), dann
 - erster wchar: 110110xxxxxxxxxxx
 - zweiter wchar: 110111xxxxxxxxxxx

(das sind die Bereiche D800 bis DBFF und DC00 bis DFFF, die im Unicode ungültige Codes sind)

UTF-16, Nachteile

Kombiniert die Nachteile von UCS-2 und UTF-8:

- Typ **wchar_t** statt **char**, Doppel-0 statt **\0**
- Sichtbare Länge ungleich Speicher-Länge, Zeichenzahl muss mühsam berechnet werden
- Code-Logik komplett neu, wieder eigene Funktions-Namen
- Bei normalem Text Platz-ineffizient
- ASCII-String-inkompatibel
- Big-Endian / Little-Endian-Problem bleibt!

Andere Unicode-Darstellungen

- ***Punycode:***

Darstellung von beliebigen Unicode-Strings nur mit den ASCII-Zeichen **a-z 0-9 -**

u.a. für internationale Domain-Namen

- ***UCS-4 bzw. UTF-32: Fix 4 Bytes pro Zeichen, enthalten den Unicode-Wert als **int** (uncodiert)***

Logik ganz einfach, aber noch mehr

Platzverschwendung: Fast 75 % Null-Bytes!

- ***UTF-7: Variabel lange Codierung in 7-Bit-char's***

Bild & Ton

Bild:

- *Pixelweise*
- oder als *Vektoren / Kurven* (Fonts, svg, ...)

Ton:

- Abgetastetes & *digitalisiertes Analog-Signal*:
Parameter *Sample-Frequenz* + *Auflösung*
- oder als *MIDI*: “*Instrumente*” + “*Noten*”

Pixel-Formate, Farbräume

3 Werte pro Pixel:

- **RGB** (rot, grün, blau: Computer, Bildschirme)
- **CMYK** (4 Werte: cyan, magenta, yellow, key =Schwarzanteil: Drucker)
- **YUV, YCbCr, ...** (Helligkeit + 2 mal Farbe: Fernsehen, komprimierte Bilder & Videos)
- **HSV** (Farbton = Winkelgrad im Farbkreis, Farbsättigung, Helligkeit: Fotos & Kunst)

=> Ineinander umrechenbar!

Komprimierung

- **Verlustfrei** (für Daten):

Dekomprimat ist bitweise ident zum Original

- **Verlustbehaftet** (für Bilder, Töne, Filme):

Original ist nicht wieder herstellbar

“Gegenpole”

- Rechenzeit / Datendurchsatz / Verzögerung beim Komprimieren
- Speicherbedarf beim Komprimieren
- Rechenzeit / Datendurchsatz / Verzögerung beim Dekomprimieren
- Speicherbedarf beim Dekomprimieren
- Kompressions-Faktor bzw. Datenrate
- Bei verlustbehafteter Komprimierung: Qualitätsverlust

“Gegenpole”, Beispiele (1)

- **Download-Archiv:**
Ziel: Möglichst hoher Faktor
Dafür: Kompression darf sehr aufwändig sein!
- **Video-Streaming:**
Ziel 1: Datenraten-Limit einhalten
Ziel 2: Keine Aussetzer
Ziel 3: Auf beiden Seiten:
Minimale Verzögerung & begrenzter Aufwand
- **Daten-Backup** (ev. in HW implementiert!):
Ziel: Hohe Geschwindigkeit (einige 100 MB/s !)

“Gegenpole”, Beispiele (2)

Varianten der Lempel-Ziv-Komprimierung:

- ***LZO, LZ4, ...:***
 - Sehr schnell und einfach ==> hoher Durchsatz!
 - Annähernd symmetrisch
(Aufwand Kompression - Aufwand Dekompression)
 - Schlechte Kompression
- ***LZMA = .xz:***
 - Viel langsamer
 - Sehr asymmetrisch bzgl. Speicher & Rechenzeit
(Kompression viel aufwändiger)
 - Sehr gute Kompression

Verlustfreie Komprimierung (1)

Nutzt “*Redundanz*” =

Wiederholungen, regelmäßige Muster, ...

==> Abhängig vom Inhalt der Daten

Texte und binärer Programmcode sind gut komprimierbar:
30-80 % Kompression möglich!

==> Es gibt immer Daten,
die nicht komprimierbar sind!!!

*Zufallszahlen, “weißes Rauschen”,
verschlüsselte Daten
(schon komprimierte Daten)*

Verlustfreie Komprimierung (2)

==> Komprimierbarkeit umso besser,
je größer das “Sichtfenster” bzw. “Gedächtnis” ist!
==> Aber: Mehr Speicherbedarf!

Block-basierte Verfahren (z.B. Bzip, ...):

- Teilt die Daten in Blöcke fixer Größe
(16 KB bis einige MB)
- Komprimiert jeden Block für sich

==> Redundanz zwischen den Blöcken
wird nicht erkannt / genutzt!

(etwas besser: “gleitendes Fenster”, z.B. letzte 64 KB)

Verlustfreie Komprimierung (3)

Bei Archiven:

Komprimierung *pro Datei* (z.B. *.zip)
oder *Datei-übergreifend* (z.B. rar, tar)?

Bei übergreifend:

- + Bessere Komprimierung,
vor allem bei vielen ähnlichen Dateien
- Braucht mehr RAM
- Nur alles dekodierbar, keine einzelnen Dateien
- **1 Datenfehler ==> alles kaputt!**

Run Length Encoding

Das primitivste Verfahren:

Ab 3 *aufeinanderfolgenden gleichen* Zeichen:

Ersetze *durch 2 Zeichen + Anzahl*
(oder z.B. Anzahl nach jedem Zeichen)

Alte verlustfreie Grafik-Formate,
vor allem schwarz-weiß:

- Fax
- Windows Bitmap

Sonst kaum noch relevant!

Huffman Coding

Häufige Zeichen bekommen kurze Bitfolgen,
seltene Zeichen bekommen lange Bitfolgen

Beispiel: **Morse-Code**

Algorithmus: Baum liefert optimale Zuordnung
von Codes zu Zeichen

Früher allein, heute kaum noch allein, aber oft
als “**Nachbrenner**” am Output anderer Verfahren:

- Alle “...zip”, gif, tiff und png Grafik, pdf, ...
= Wörterbuch + Huffman
- Jpeg & Mpeg = DCT + Huffman

Wörterbuch = Lempel-Ziv-xxx

- Baut “***Wörterbuch***”
aus allen Zeichenfolgen im Input auf
- Ersetzt schon vorgekommene Zeichenfolgen
durch ihre Nummer im Wörterbuch
- Bei manchen Verfahren (z.B. Brotli):
Beginnt mit vordefiniertem statt leerem Wörterbuch:
Allgemein häufige Zeichenfolgen schon drin

Fast alle heutigen Verfahren (.zip usw.)!

Sonderfall bzip2

“Burrows-Wheeler-Transformation”

Daten werden (umkehrbar) blockweise umsortiert,
sodass gleiche Zeichen häufig aufeinanderfolgen

*==> Anschließende Positions- & Huffman-Codierung
ist besonders wirkungsvoll!*

Lange Zeit sehr beliebt, weil

- relativ gute Kompression
- Lizenz- und patentfrei!

Heute immer weniger verwendet...

Verlustbehaftete Komprimierung

Ziel:

*Qualität dort verschlechtern,
wo es dem Menschen am wenigsten auffällt!*

Geht je nach Daten besser oder schlechter,
aber Daten für immer vernichten geht *immer!*

*==> Man kann immer
verlustbehaftet komprimieren!
(bzw. eine fixes Kompressionsverhältnis erreichen)*

mp3 Ton (1)

Erkenntnisse aus der Psychoakustik:

- Hohe Töne sind schlechter / ungenauer hörbar
- Laute Töne verdecken leise Töne, auch kurz vorher/nachher
- In der Tonhöhe ähnliche Töne sind schwer unterscheidbar

mp3 Ton (2)

- Stereo durch 1 Kanal + Differenz ersetzen:
Differenz ist besser komprimierbar!
(und hohe Töne haben weniger Stereo-Eindruck als tiefe Töne)
- **Fourier-Transformation** ==> Frequenzen
- Genauigkeit festlegen = “**Quantisierung**” = Skalierungsfaktor für jede Frequenz separat:
 - Hohe Anteile zunehmend ungenauer speichern
 - Genauigkeit neben starken Frequenzen absenken
- Am Ende: Huffman Coding (u.a. weil viele 0-Werte)

Triviale Bildkompression

Der Mensch ist bei Helligkeits-Schwankungen viel empfindlicher als bei Farbton-Schwankungen

==> YUV-Farbmodell,
Y doppelt so genau speichern wie UV
(bzw. UV nur bei jedem 2. / 4. Pixel speichern
oder gemittelt aus 2 bzw. 4 Pixeln:
YUV 422 oder YUV 420)

==> 37,5 % Ersparnis!

jpeg Bild (1)

- Umwandlung in *YUV*, Farbreduktion:
Farbe nur für je 4 Pixel im Mittel
- Aufteilung in ***Blöcke zu 8x8 Pixel***
- ***Zweidimensionale DCT***
(“*Diskrete Cosinus-Transformation*”):
Transformation der 64 Pixel-Werte
in 64 Funktions-Koeffizienten

Aufbau der *DCT-Matrix* (siehe Wikipedia):

Links oben sind ist der “***grobe***” ***Bildaufbau***
Rechts unten sind die ***feinen Details!***

jpeg Bild (2)

DCT ist verlustfrei bis auf Rundungsfehler!

Quantisierung: Jedes Feld der Matrix

- mit einem Faktor multiplizieren / dividieren
- und auf n Bits diskretisieren

“Stellschraube” für Datenmenge und Qualität:

Wie ungenau speichern bzw. was fällt weg?

Rechts unten in der Matrix ungenauer quantisieren:

Groben Bildaufbau relativ gut erhalten,
hohe Frequenzen bzw. starke Muster

benachbarter Pixel ungenauer speichern!

jpeg Bild (3)

==> Viele 0 bzw. viele kleine, gleiche Werte (z.B. -2...+2)
rechts unten in der Matrix

- Matrix diagonal in Schlangenlinien
durchlaufen bzw. linearisieren

==> *Besser komprimierbare Anordnung:*
Wichtige Werte vorne, dann
viele 0 und ein paar 1 oder -1 nacheinander

- *Huffman Coding*

Video-Komprimierung

- Gelegentlich *Vollbilder*,
komprimiert wie bei JPEG
- Dazwischen nur *Verschiebe-Information*:

*Welche Bildteile haben sie wie
gegenüber dem vorigen Bild
verschoben?*

“The end”

Fragen?