Übungsaufgabe 2
Signal und Bildverarbeitung 1
Human Centered Computing Master
Brabenetz, Wachert-Rabl

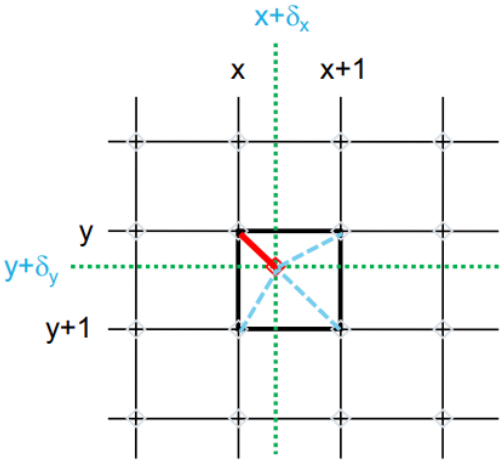
Inhaltsverzeichnis

Aufgabe 2.1 Resampling und Interpolation

a) **Implementieren Sie einen Resampling Filter, der die Größenveränderung eines Eingangsbildes erlaubt. Der Gleitkomma-Skalierungsfaktor ist dabei vom Benutzer einzugeben und soll eine Vergrößerung / Verkleinerung um max. Faktor 10.0 erlauben. Die Interpolation ist dabei als Nearest Neighbour zu realisieren. Retournieren Sie das Ausgangsbild in einem neuen Fenster. Diskutieren Sie Ihre gewählte Berechnungsstrategie für die Koordinaten Transformation (vgl. Folie 13-15).**

Um ein Bild in der Größe zu verändern, müssen die Pixel des neuen Bildes neu berechnet werden. Oft entstehen bei der Größenveränderung Pixel an Zwischenpositionen – um dann zu entscheiden, welchen Wert dieses neuen Pixels bekommt, gibt es verschiedene Interpolationsvarianten.

Bei der Nearest Neighbour Interpolation wird errechnet, welchem vorhandenen Pixel die neue Pixelposition am nächsten kommt – dieser nächstgelegene skalare Pixelwert wird dann auch an den neuesten Pixel übergeben (bzw. werden die Werte des Pixels schlichtweg gerundet). Das Problem bei dieser einfachen Variante der Neuberechnung: die Kantenverläufe des Bildes können eckig oder kantig erscheinen. Allerdings ist diese Art der Berechnung die richtige Wahl, wenn es um große Dateien geht, die eine schnelle Berechnung erfordern.



An der nebenstehenden Darstellung wird die Berechnung der neuen Koordinaten dargestellt *(Abbildung aus den Vorlesungsfolien*). Daran erkennt man, dass die neue Pixel zwischen 4 Punkten liegt. Da die Entfernung des neuen Pixels am nähesten zum links oben liegenden Pixels liegt, werden die Koordinatenwerte nach dem Runden die gleichen sein, wie die des linken oberen Pixels.

Implementiert wurde die *Nearest Neighbour Interpolation* folgendermaßen:

```
    public int[][] getnearestNeighbour(int[][] inImg, int width, int height, double factor) {
        int[][] returnImage = new int[(int) ((double) width * factor + 1.0)][(int) ((double)
height * factor + 1.0)];

        for (int w = 0; w < (int) ((double) width * factor); ++w) {
            for (int h = 0; h < (int) ((double) height * factor); ++h) {
                int posXint = (int) ((double) w / factor + 0.5);
                int posYint = (int) ((double) h / factor + 0.5);

                if (posXint < 0)
                    posXint = 0;
                if (posYint < 0)
                    posYint = 0;
                if (posXint >= width)
                    posXint = width - 1;
                if (posYint >= height)
                    posYint = height - 1;

                returnImage[w][h] = inImg[posXint][posYint];
            }
        }
        return returnImage;
    }
```

Jeweils vom Originalabild ausgehend sind in der folgenden Abbildung die Vergrößerungen mittels Nearest Neighbour Interpolation mit den Faktoren 2 | 5 | 10 zu sehen.

Faktor 2

Faktor 5

Faktor 10

In der untenstehenden Abbildung wird die Verkleinerung eines Bildes mittels Nearest Neighbour Interpolation mit den Faktoren -2 | -5 | -10 veranschaulicht, wobei die Interpolation jeweils vom Originalbild ausgeht.

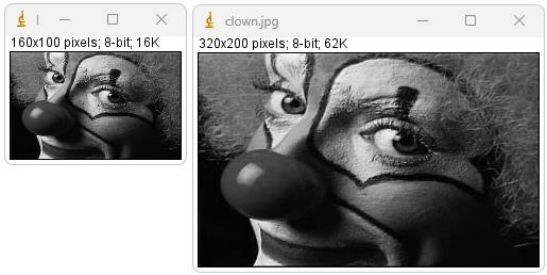
Faktor -2

Faktor -5

Faktor -10

Die Unterschiede zwischen Vergrößerung eines Bildes und Verkleinerung mittels Nearest Neighbour Interpolation werden untenstehend dargestellt. Unter Beibehaltung der Originalgrößen der veränderten Bilder sieht das Ergebnis gut aus – die neben dem Originalbild entstandenen Ergebnisbilder sind entweder halb so groß (Faktor -2) oder doppelt so groß (Faktor 2).

Faktor -2 Originalbild



Faktor 2

Vergrößerung eines Ausschnittes der oben dargelegten Ergebnisbilder:

Faktor -2 Originalbild **Faktor 2**

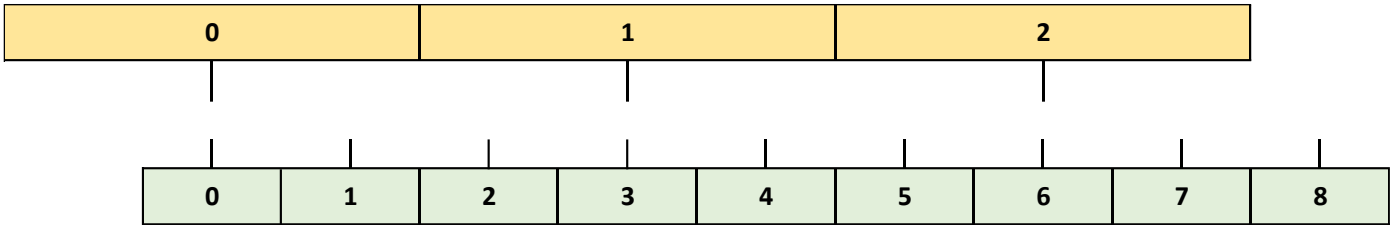
Bei nebeneinanderliegenden Ausschnitten des Bildes mit dem Originalbild in der Mitte, erkennt man bei einem Faktor -2 (also einer Verkleinerung des Bildes) natürlich einen starken Unterschied, da nur halb so viele Pixel verfügbar sind. Jedoch ist zwischen dem Originalbild und der Vergrößerung mittels Nearest Neighbour Interpolation mit dem Faktor 2 kaum ein Unterschied zu erkennen. Es stehen zwar doppelt so viele Pixel zur Verfügung, trotzdem weist das Ergebnis stufige Kanten auf und wirkt verpixelt.

Um ein möglichst gutes Ergebnis zu erhalten, müssen die Koordinaten umgerechnet werden, da die neue Breite (die sich aufgrund des selbst gewählten Faktors ergibt) auf die Breite des Originalbildes umgerechnet werden muss. Dazu gibt es mehrere Varianten, wobei die Berechnungen für ein besseres Ergebnis komplizierter sind.

Beim einfachsten Weg ***Variante A)*** erfolgt die direkte Umrechnung über den Skalierungsfaktor, der sich aus dem Verhältnis der Abmessungen der beiden Bilder (Original und Ergebnis) ergibt: $\mathbf{s = w_b/w_a}$ (neues Bild b / Originalbild a). Dadurch entstehen, je nachdem wo die Indizes beginnen, über- oder unterrepräsentierte Pixel am Beginn oder am Ende der Zeile. Die Berechnung des neuen Bildes b mit dieser Variante erfolgt durch **b = a * s** (wobei a die Positionen der Pixel des Originalbildes und b die Positionen der Pixel des neuen Bildes darstellen).

s = 3,0

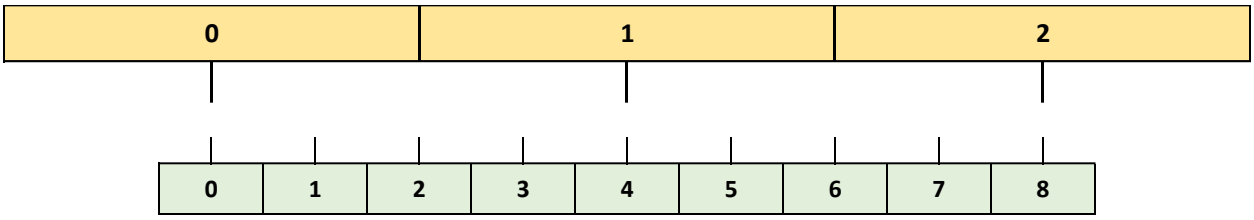
a	b
0	0
1	3
2	6



Die etwas kompliziertere ***Variante B)*** stellt sicher, dass die Mittelpunkte beider Zeilen übereinstimmen und es somit an beiden Enden überrepräsentierte Pixel gibt. Um dies zu erreichen, muss der Skalierungsfaktor so adaptiert werden, dass eine Übereinstimmung bei den Zielkoordinaten erzwungen werden kann. Um den verbesserten Skalierungsfaktor zu errechnen, muss die neue Breite – 1 durch die Originalbreite – 1 berechnet werden: **s‘ = (wb-1) / (wa-1)**. Das neue Bild wird demnach folgendermaßen errechnet: **b = a * s‘.**

s' = 4,0

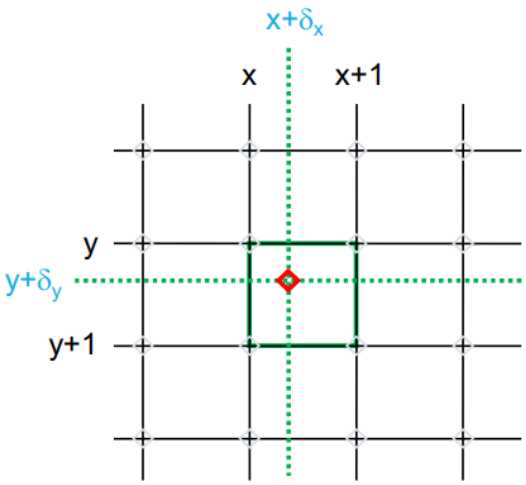
a	b
0	0
1	4
2	8



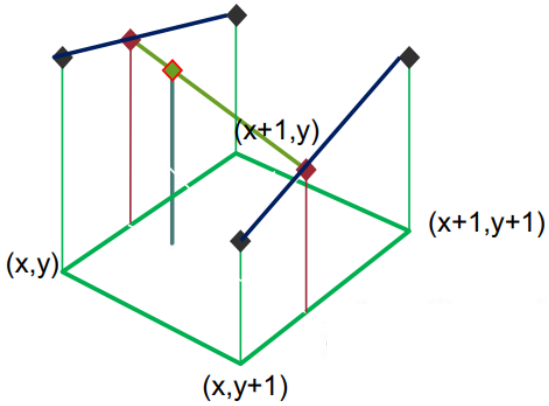
Bei unserer Implementierung wurde ***Variante B)*** verwendet um im Gegensatz zu Variante A) eine bessere Annäherung der neu errechneten Breite auf das Originalbild zu berechnen. Auch, wenn die Berechnung dieser Variante etwas aufwändiger ist, ist es ein guter Kompromiss, um eine bessere Qualität zu erzielen.

b) **Implementieren Sie zusätzlich eine Bi-Lineare Interpolation zum Resampling des Eingangsbildes. Retournieren Sie auch diesen Output in einem neuen Fenster. Berechnen Sie ferner ein Differenzbild und zeigen Sie dieses an. Wodurch entstehen die Unterschiede in den skalaren Pixelwerten?**

Die Bi-Lineare Interpolation ist der beste Kompromiss zwischen Laufzeit und Qualität. Die Kantenbildung, die es bei der Nearest Neighbour Interpolation gibt, kann umgangen werden. Berechnet wird bei der Bi-Linearen Interpolation der gewichtete Mittelwert der benachbarten skalaren Werte. Dabei werden die umliegenden Pixel zur Veranschaulichung in eine 3D-Darstellung gebracht.



An dieser Darstellung *(aus den Vorlesungsfolien)* wird veranschaulicht, wo sich der neue Pixel befinden soll – er muss zwischen 4 Punkten liegen.



Nebenstehend wird die 3D-Darstellung zur Verbildlichung der Bi-Linearen Interpolation dargelegt *(aus den Vorlesungsfolien)*. Der jeweilige Wert der Pixel wird hier als Höhe angegeben. Die Punkte werden jeweils in x-Richtung miteinander verbunden. Danach wird an der Position δ_x eine Verbindung geschaffen, um so an der Stelle δ_y die Höhe (und damit den Wert) des neuen Pixels zu errechnen.

Implementiert wurde die *Bi-Lineare Interpolation* wie folgt:

```
public int[][] getBilinear(int[][] inImg, int width, int height, double factor) {
    int[][] returnImage = new int[(int) ((double) width * factor + 1.0)][(int) ((double) height *
factor + 1.0)];
    for (int w = 0; w < (int) ((double) width * factor); ++w) {
        for (int h = 0; h < (int) ((double) height * factor); ++h) {
            int x = (int) ((double) h / factor);
            int y = (int) ((double) w / factor);
            int x1 = (int) Math.ceil((double) h / factor);
            int y1 = (int) Math.ceil((double) w / factor);
            if (y1 < width && x1 < height) {
                double x_weight = (double) h / factor - (double) x;
                double y_weight = (double) w / factor - (double) y;
                int a = inImg[y][x];
                int b = inImg[y][x1];
                int c = inImg[y1][x];
                int d = inImg[y1][x1];
                int pixel = (int) ((double) a * (1.0 - x_weight) * (1.0 - y_weight) + (double) b
* x_weight * (1.0 - y_weight) +
                        (double) c * y_weight * (1.0 - x_weight) + (double) d * x_weight *
y_weight);
                returnImage[w][h] = pixel;
            }
        }
    }
    return returnImage;
}
```

In der untenstehenden Abbildung wird die Vergrößerung eines Bildes mittels Bi-Linearer Interpolation mit den Faktoren -2 | -5 | -10 dargestellt, wobei die Interpolation jeweils vom Originalbild ausgeht.

Faktor 2

Faktor 5

Faktor 10

Jeweils vom Originalabild ausgehend sind in der folgenden Abbildung die Verkleinerungen mittels Bi-Linearer Interpolation mit den Faktoren -2 | -5 | -10 zu sehen.

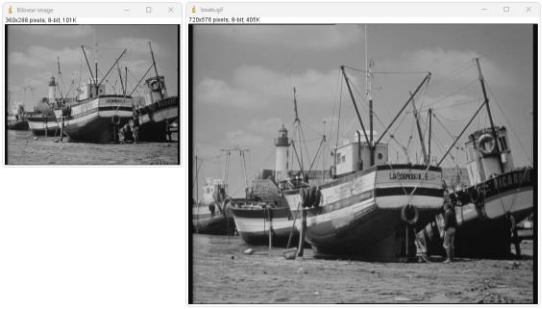
Faktor -2

Faktor -5

Faktor -10

Um die Größenänderung eines Bildes mittels Bi-Linearer Interpolation besser zu verdeutlichen, wird ein Testbild verkleinert (Faktor -2) und vergrößert (Faktor 2). Die Ergebnisse sind in der untenstehenden Abbildung zu betrachten. Während bei der Originalgröße der veränderten Bilder im Vergleich zur Nearest Neighbour Interpolation kein Unterschied erkennbar ist, so kann bei genauerer Betrachtung eines Ausschnittes eine deutliche Verbesserung beobachtet werden.

Faktor -2 Originalbild



Faktor 2

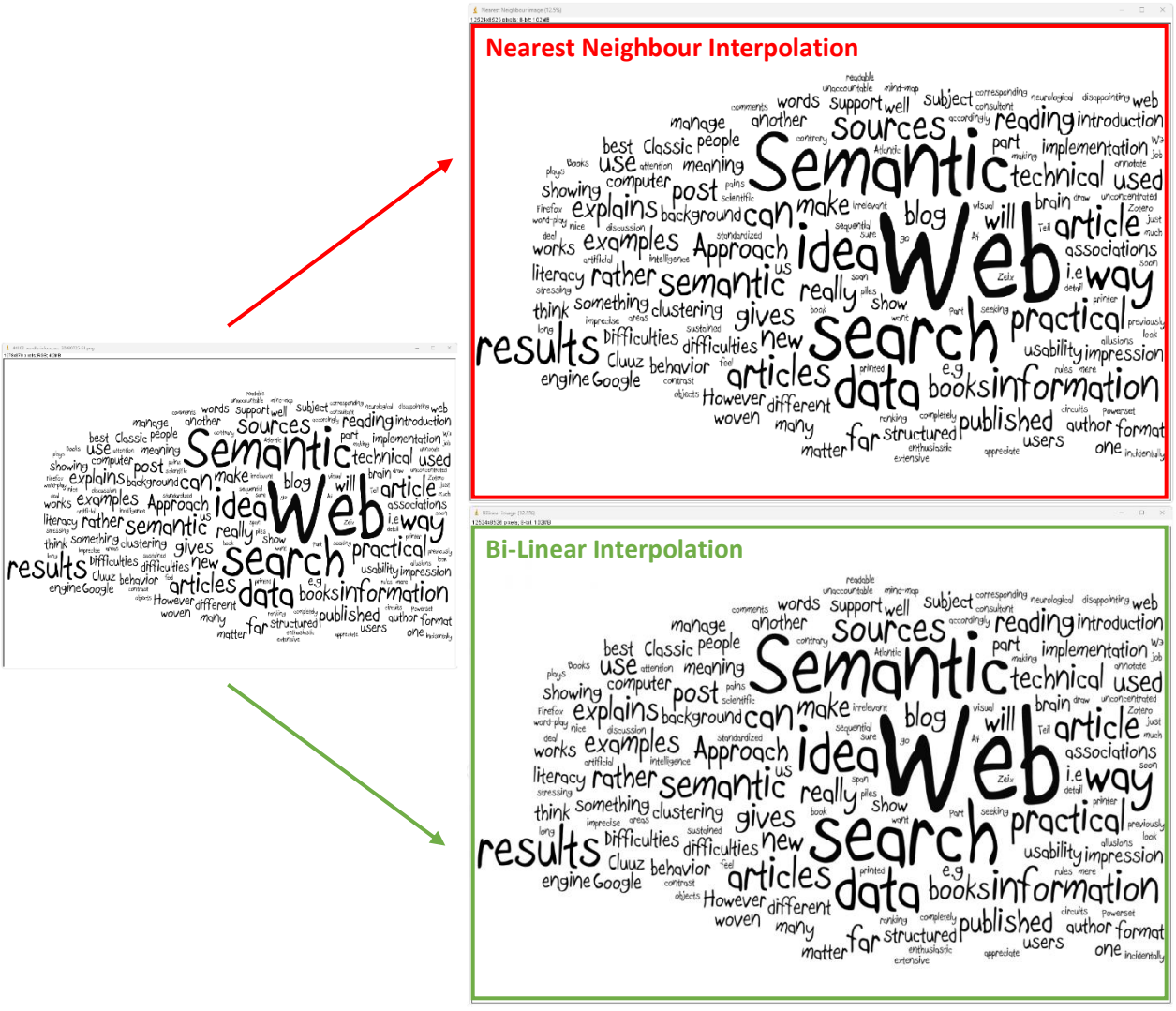
Vergrößerung eines Ausschnittes der oben dargelegten Ergebnisbilder:

Faktor -2 Originalbild **Faktor 2**

Wie bereits bei der Nearest Neighbour Interpolation ist der Unterschied bei der Verkleinerung eines Bildes deutlicher erkennbar. Jedoch unterscheidet sich das durch die Bi-Lineare Interpolation vergrößerte Bild stark. Die im Bild befindlichen Kanten sind nicht so eckig wie bei der zuvor erwähnten Interpolation – somit kann ein sichtbar besseres Ergebnis erzielt werden, da nicht nur die gleichen Werte wie des am nächsten liegenden Pixels verwendet werden, sondern ein eigener Wert für die Koordinaten des neuen Pixels errechnet wird.

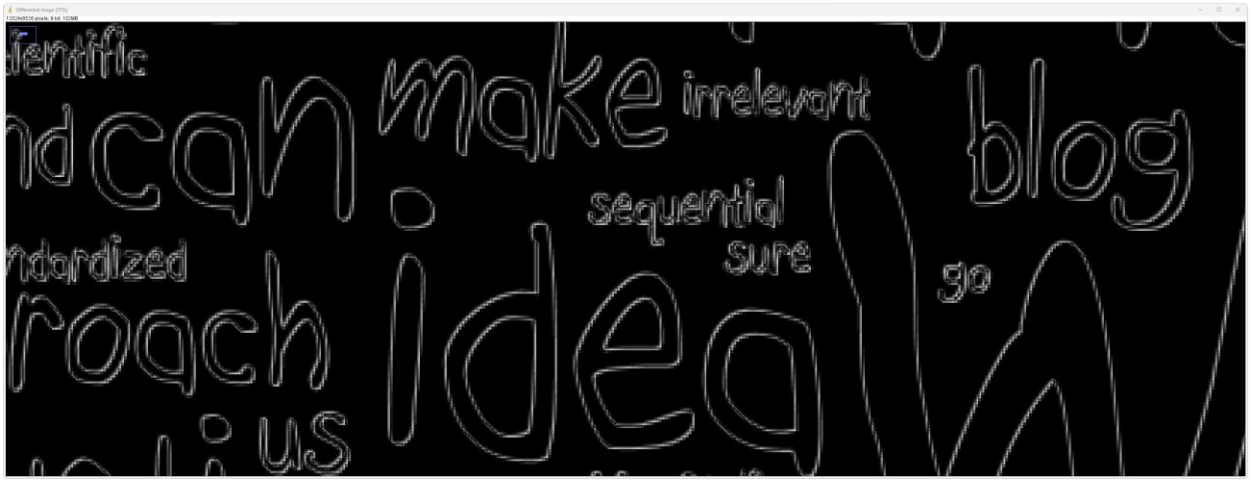
Um ein *Differenzbild* darzustellen, welches die Unterschiede zwischen den beiden Interpolations-Varianten darlegt, wurde folgendes implementiert:

```
public int[][] getDifferentialImage(int[][] nnImg, int[][] biImg, int width, int height, double
factor) {
    int[][] differentialImage = new int[(int) ((double) width * factor + 1.0)][(int) ((double)
height * factor + 1.0)];
    for (int w = 0; w < (int) ((double) width * factor); ++w) {
        for (int h = 0; h < (int) ((double) height * factor); ++h) {
            if ((nnImg[w][h] - biImg[w][h]) < 0) {
                differentialImage[w][h] = (nnImg[w][h] - biImg[w][h]) * (-1);
            } else {
                differentialImage[w][h] = nnImg[w][h] - biImg[w][h];
            }
        }
    }
    return differentialImage;
}
```



Differenzbild der oben angeführten Ergebnisse:

Anhand der weißen Linien erkennt man, wo die Unterschiede der beiden Interpolations-Varianten liegen. Die schwarzen Bereiche zeigen, wo beide Bilder ident sind, an den weißen Linien unterscheiden sich diese voneinander da die Pixel an diesen Stellen durch die verschiedenen Interpolationen unterschiedliche Werte erhalten haben.



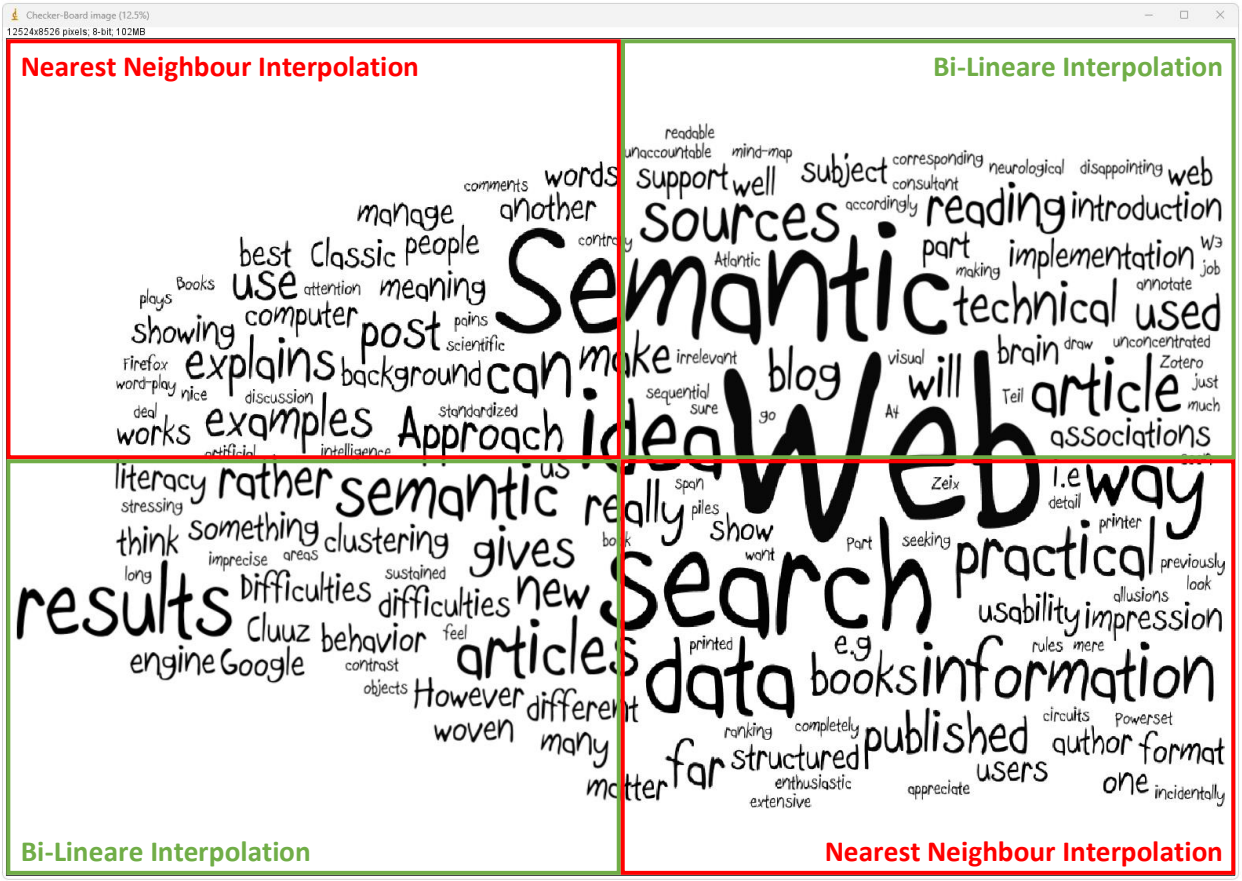
Man kann erkennen, dass die Unterschiede nur in den Randbereichen liegen, wenn nebeneinanderliegende Pixel unterschiedliche Werte aufweisen. Wenn die Werte aller umliegender Pixel gleich sind, dann erhält der neue, in der Mitte befindliche Pixel ebenfalls den gleichen Wert, egal um welche Interpolationsmethode es sich handelt.

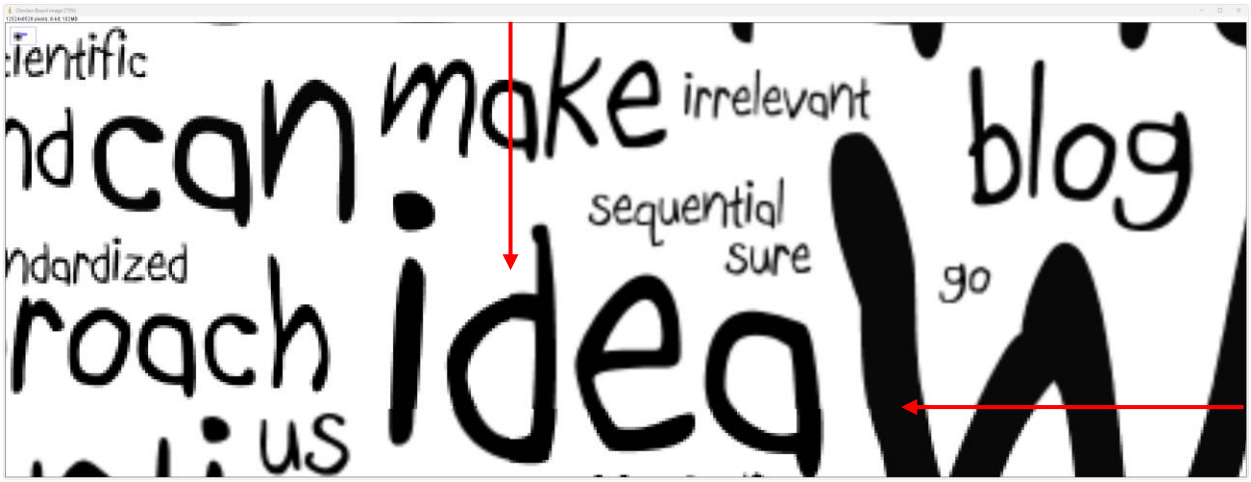
c) **Implementieren Sie eine Checker-Board Darstellung und vergleichen Sie die beiden Interpolationsstrategien. Wie würden Sie die beiden Interpolationsstrategien in Bezug auf Laufzeit und erzielbare Qualität charakterisieren?**

Die *Checker-Board Darstellung* vereinfacht den direkten Vergleich zwischen der Nearest Neighbour und der Bi-Linearen Interpolation, da beide Ergebnisse in einem Bild dargestellt werden. Diese Darstellung wurde folgendermaßen implementiert:

```
public int[][] getCheckerImage(int[][] nnImg, int[][] biImg, int width, int height, double
factor) {
    int[][] checkerImg = new int[(int) ((double) width * factor + 1.0)][(int) ((double) height *
factor + 1.0)];
    for (int w = 0; w < (int) ((double) width * factor); ++w) {
        for (int h = 0; h < (int) ((double) height * factor); ++h) {
            if (w < (((int) ((double) width * factor)) / 2) && h < (((int) ((double) height *
factor)) / 2)
                    || w >= (((int) ((double) width * factor)) / 2) && h >= (((int) ((double)
height * factor)) / 2)) {
                checkerImg[w][h] = nnImg[w][h];
            } else {
                checkerImg[w][h] = biImg[w][h];
            }
        }
    }
    return checkerImg;
}
```

An einem Beispielbild mit starken Kanten ist die Demonstration der beiden Interpolationen am deutlichsten zu erkennen.





An den roten Pfeilen kann man sehen, wo sich die verschiedenen Interpolationen begegnen. Es fällt stark auf, dass der Bereich der Nearest Neighbour Interpolation wesentlich kantiger erscheint als der Bereich der Bi-Linearen Interpolation.

Nearest Neighbour Interpolation

Bi-Lineare Interpolation

Auch beim direkten Vergleich zweier Ausschnitte der beiden Methoden ist deutlich zu erkennen, dass die Nearest Neighbour Interpolation zwar auf den ersten Blick schärfer aussehen mag, diese aber in der Kantenabbildung wesentlich stufiger und ungenauer abgebildet wird. Die Bi-Lineare Interpolation hingegen weist weichere Verläufe auf – das Bild kann als qualitativ hochwertiger bezeichnet werden.

Doch bei der Laufzeit liefert die Bi-Lineare Interpolation schlechtere Ergebnisse als die Nearest Neighbour Interpolation. Hier müssen nicht nur die Koordinaten des neuen Pixels auf ganzzahlige Werte gerundet werden, sondern es muss ein komplett neuer Wert errechnet werden

Aufgrund beider Faktoren - Qualität und Laufzeit - wird die Bi-Lineare Interpolation als der beste Kompromiss betrachtet. Die Größenänderung mittels Nearest Neighbour Interpolation wird nur dann verwendet, wenn die Berechnung schnell gehen muss.

Aufgabe 2.2 Klassifizierung mittels Kompression

a) **Klassifizieren Sie Sprachen (textuelle Repräsentation) mittels Kompression. Wählen Sie dabei zumindest n=8 Klassen (d.h. Sprachen), für die Sie repräsentative Vergleichsdatensätze aufbereiten. Welche Vorverarbeitungen sind beim Aufbau der Vergleichsdatensätze bzw. Testdatensätze sinnvoll bzw. notwendig? Klassifizieren Sie dann Beispieltexte und werten Sie die Ergebnisse statistisch aus (exakte Treffer bzw. Rang, ev. Verwendung einer Konfusions-Matrix). Können die erzielten Ergebnisse bei vorliegender Testanzahl als statistisch signifikant betrachtet werden? Wie hängt die erzielbare Klassifikationsgenauigkeit mit der Anzahl der zu diskriminierenden Klassen zusammen?**

Für diese Übung wurde versucht mittels Zip Komprimierung und dessen statistischer Funktionsweise mittels Wörterbuch, Texte einer jeweiligen Sprache zuzuordnen (Klassifizieren).

Dabei wurde als erstes ein Text der jeweiligen Sprache (n = 8 Sprachen im Sample) mittels Zip in einer .txt File komprimiert. Wir haben hierfür Märchentexte der Gebrüder Grimm herangezogen und jeweils 5000 Zeichen in die 8 Sprachen übersetzt. Wichtig ist hierbei, dass ähnlich lange Texte gewählt werden. In unserem Beispiel wurde das Märchen von Schnewittchen als Trainings Datensatz für alle 8 Sprachen gewählt.

Im nächsten Schritt wurde das Märchen von Aschenputtel in die 8 Sprachen übersetzt und als Test File für das Klassifizierungs-Modell genutzt. Dabei ist es wichtig den Test Text direkt an den Train Text anzuhängen und beide Texte zusammen, zu komprimieren.

Nun muss nur noch Delta (Unterschied) zwischen der Zip (Trainings Text + Test Text) minus Zip (Trainings Text) überprüft werden. Je nachdem wo der Unterschied am geringsten ausfällt, wurde eine Klassifizierung getroffen.

Auswerting der Sprachen Text Erkennung mittels Zip Klassifizierung:

Test / Train	Deutsch	Dänisch	Englisch	Französisch	Indonesisch	Türkisch	Portugiesisch	Ungarisch	Match
Deutsch	**1622**	1223	1234	1394	1309	1285	1384	1415	**Yes**
Dänisch	1798	**1091**	1235	1385	1316	1293	1382	1441	**Yes**
Englisch	1806	1239	**1109**	1400	1315	1296	1389	1428	**Yes**
Französisch	1807	1232	1239	**1224**	1347	1299	1348	1424	**Yes**
Indonesisch	1814	1249	1242	1420	**1148**	**1286**	1398	1444	**Yes/No**
Türkisch	1814	1249	1242	1420	**1148**	**1286**	1398	1444	**No/Yes**
Portugiesisch	1815	1235	1242	1373	1316	1293	**1207**	1411	**Yes**
Ungarisch	1821	1251	1248	1405	1319	1292	1376	**1213**	**Yes**
Genauigkeit	100	100	100	100%	92%	92%	100%	100	98%
Genauigkeit	1	1	1	1	1 - (1/6)/2	2 - (1/6)/2	1	1	Mean(B12-I12)

Das Klassifizierungs-Model hat weit über dem Durchschnitt korrekt klassifiziert und hat lediglich zwischen Indonesisch und Türkisch, jeweils beide Test Texte gleich wertig klassifiziert. Man könnte hier nun sprachwissenschaftlich argumentieren, dass es möglicherweise Ähnlichkeiten zwischen den beiden Sprachen gibt (dies liegt aber nicht im Fokus der Übung).

Die Methode der Zip Komprimierung und Klassifizierung beruht auf der statistischen Funktion des Zip Wörterbuchs. Ähnliche Texte haben ähnliche Codes im Wörterbuch abgespeichert und können

gemeinsam auf diese zugreifen. Daraus ergibt sich ein geringerer Speicherbedarf und man kann anhand von Delta klassifizieren.

Man sollte auf jeden Fall beachten, dass die Texte ungefähr dieselbe länge haben und generell nicht zu lange sind, ansonsten wird ein zweites Wörterbuch während der Komprimierung gestartet und es kann zu Ungenauigkeiten führen.

(Die genaue Excel Auswertung ist im Anhang zu finden)

Je mehr Texte verwendet werden umso eher kann es zu Outlier kommen, dennoch ist dieses Modell der Klassifizierung statistisch relevant.

b) **OPTIONAL – nur für Interessierte/Expert*innen: Klassifizieren Sie Beispielbilder (8bit Grauwert) mittels Kompression. Wählen Sie zumindest n=4 Klassen, für die Sie repräsentative Vergleichsdatensätze aufbereiten. Welche Vorverarbeitungen sind beim Aufbau der Vergleichsdatensätze bzw. Testdatensätze sinnvoll bzw. notwendig? Klassifizieren Sie dann Beispieltexte und werten Sie die Ergebnisse statistisch aus (exakte Treffer bzw. Rang). Können die erzielten Ergebnisse bei vorliegender Testanzahl als statistisch signifikant betrachtet werden? Wie hängt die Klassifikationsgenauigkeit mit der Anzahl der zu diskriminierenden Klassen zusammen?**

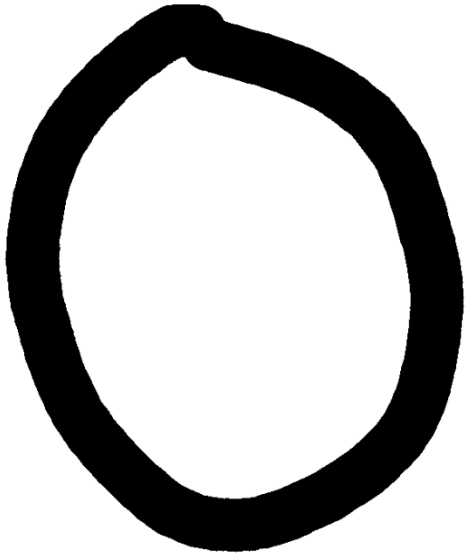
Um Bilder zu klassifizieren, wird grundsätzlich genauso vorgegangen, wie bei den Text Klassifizierungen. Die Train & Test Bilder werden als Ascii Code in einem Editor ausgegeben und dann als Text File komprimiert. Hier wird wieder das Delta zwischen Zip (Trainings Image+ Test Image) minus Zip (Trainings Image) zur Klassifizierung herangezogen.

Zur Vorbereitung: Wichtig ist, dass die Bilder Klassen ähnliche Objekte beinhalten, ich habe mich hierbei für eine Testdatenbank aus handschriftlichen Zahlen entschieden. Außerdem müssen die Bilder dieselbe Größe, Auflösung und Farbraum (8 Bit Graustufen) haben. Es sollte zudem darauf geachtet werden nicht zu Große Bilder zu benutzen da es ansonsten wieder zu mehreren Wörterbüchern während der Komprimierung kommen kann und diese die Klassifizierung verfälschen können.

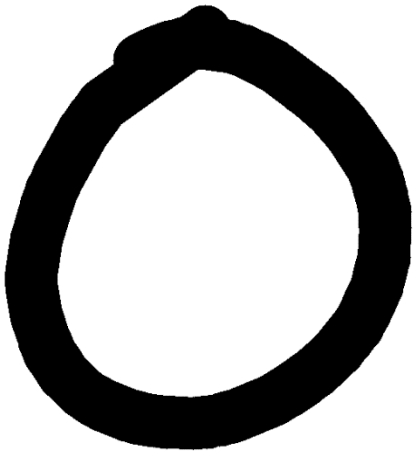
Hier die Trainings und zugehörigen Test Bilder:

Zahl 0

Trainings Bild:



Test Bild



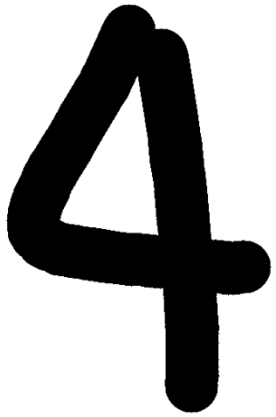
Zahl 3

Trainings Bild

Test Bild

Zahl 4

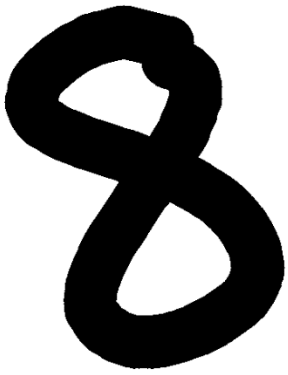
Trainings Bild



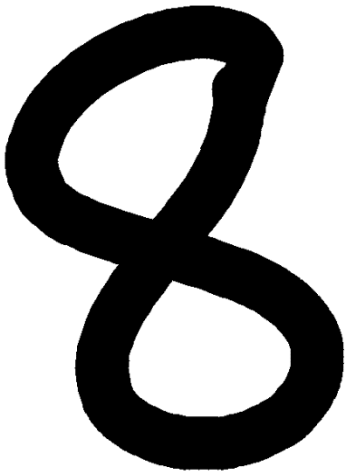
Test Bild

Zahl 8

Trainings Bild



Test Bild



Statistische Auswertung der Bilder Klassifizierung per Zip Komprimierung:

	Train 0	Train 3	Train 4	Train 8	Match
Test 0	**7082**	6871	3553	7543	**Yes**
Test 3	7198	**6869**	3562	7581	**Yes**
Test 4	7274	6949	**3537**	7610	**Yes**
Test 8	7174	6900	3550	**7537**	**Yes**

Die Treffergenauigkeit lag bei unserem Versuch bei 100%, dies zeigt, dass das Modell auch zur Bildklassifzierung herangezogen werden kann. Die verglichenen Bilder sind sich natürlich sehr ähnlich und mit weiterer Komplexität der Bildinhalte sowie einer größeren Menge (n) an Test kann die Trefferquote abnehmen.

Aufgabe 2.3 Kompression und Code-Transformation

a) **Komprimieren Sie die Sequenz dabbababababbbaaaababababccdd (n=4 Symbole: {a,b,c,d}) händisch mittels Lempel-Ziv Kompression. Berechnen Sie die erzielbare Kompressionsrate.**

Bei der Lempel-Ziv Kompression handelt es sich um eine verlustfreie Kompression. Besonders effizient ist sie bei sich öfter wiederholenden Mustern oder (Zeichen-) Ketten. Bei dem Kompressionsvorgang wird ein sg. Wörterbuch erstellt, das bereits vorhandene Muster enthält. Kommt eines dieser Muster noch einmal vor, dann gibt es dafür bereits einen im Wörterbuch eingetragenen Wert, der an dieser Stelle verwendet werden kann. Ein Eintrag wird dann eingetragen, wenn er aus mindestens 2 Elementen besteht. Wenn es eine Kette (best. aus mind. 2 Elementen) bereits im Wörterbuch gibt, dann wird ein neuer Eintrag mit dem darauffolgenden Zeichen hineingeschrieben.

Um ein komprimiertes Element zu dekomprimieren, kann der Vorgang erneut durchgeführt werden. Aus der Kompression kann immer auf das ursprüngliche Zeichen zurückgeschlossen werden. Jede kodierte Zeichenkette enthält somit die *Information* und gleichzeitig die *Code-Transformation*.

Aktuelles Zeichen	Nächstes Zeichen	Bereits im Wörterbuch? (Y/N)	Neuer Eintrag
d	**a**	N -> d	da <256>
a	**b**	N -> a	ab <257>
b	**b**	N -> b	bb <258>
b	**a**	N -> b	ba <259>
a	**b**	Y -> 257	aba <260>
a	**b**	Y: aba -> 260	abab <261>
b	**b**	Y -> 258	bbb <262>
b	**a**	Y -> 259	baa <263>
a	**a**	N -> a	aa <264>
a	**a**	Y -> 264	aab <265>
b	**a**	Y -> 259	bab <266>
b	**a**	Y: bab -> 266	babc <267>
c	**c**	N -> c	cc <268>
c	**b**	N -> c	cb <269>
b	**d**	N -> b	bd <270>
d		N -> d	

Ergebnis:

d	a	b	b	ab	aba	bb	ba	a	aa	ba	bab	c	c	b	d
<97>	<98>	<99>	<99>	<257>	<260>	<258>	<259>	<98>	<264>	<259>	<266>	<100>	<100>	<99>	<97>

Ursprüngliche Anzahl der Zeichen: **25**
Anzahl der Zeichen nach der Kompression: **16**

$C_R \approx 1,6$
Die Dateigröße der komprimierten Sequenz konnte um **36%** gegenüber der Originaldatei verkleinert werden.

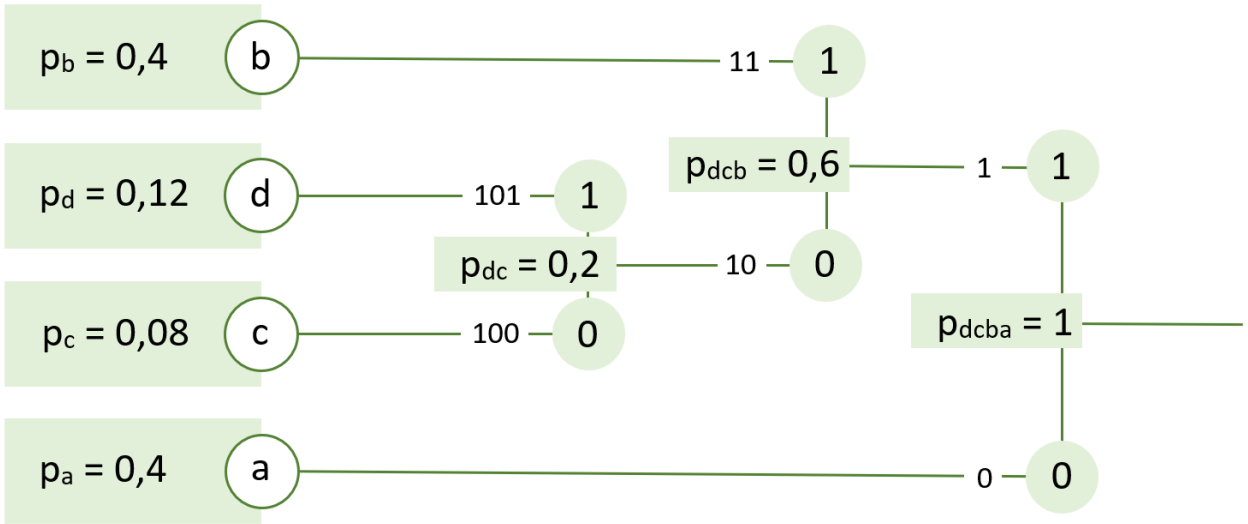
b) **Transformieren Sie die Sequenz dabbababababbbaaaababababccdd (n=4 Symbole: {a,b,c,d}) händisch mittels Huffmann-Coding in eine komprimierte Darstellung und geben Sie dabei auch den Huffmann-Baum an. Berechnen Sie die erzielbare Kompressionsrate bzw. die mittlere Codewort Länge. Bei welcher eigenständig gewählten 10-stelligen Sequenz ist die Kompressionsrate maximal bzw. bei welcher 10-stelligen Sequenz ist die Kompressionsrate minimal? Von welcher Beschaffenheit der Daten hängt die erzielbare Kompressionsrate ab?**

Bei der Huffmann Kodierung wird in das binäre Layout des Codes komprimiert. Dabei spielt die Häufigkeit eines Codewortes eine essenzielle Rolle – je öfter ein Wort vorkommt, desto kürzer ist seine zugewiesene Bit-Sequenz (z.B. 1 – für das häufigste Element, 01, 001, 000). Es macht für diese Kodierung keinen Unterschied, in welcher Reihenfolge die Elemente angeordnet sind.

Die für das Beispiel vorgegebene Sequenz besteht aus 4 Symbolen und insgesamt 25 Zeichen.

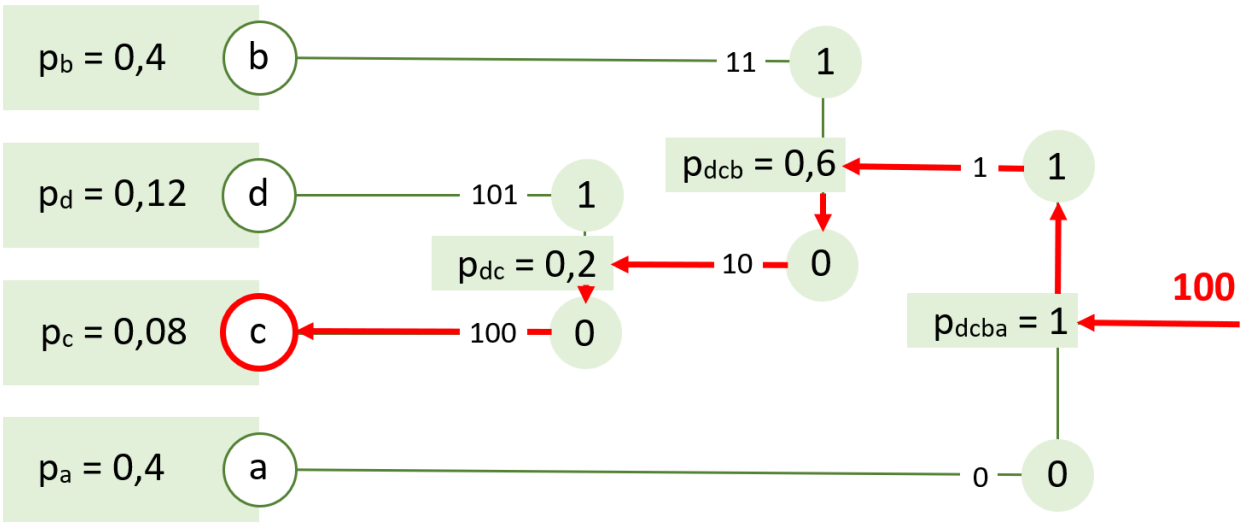
Vorkommen	sortiert nach Häufigkeit	Verteilung der Symbole als Wahrscheinlichkeit	Bit-Sequenz der einzelnen Symbole im Huffmann-Baum
d: 3	a: 10	pa = 0,4	a = 0
a: 10	b: 10	pb = 0,4	b = 11
b: 10	d: 3	pd = 0,12	d = 101
c: 2	c: 2	pc = 0,08	c = 100

Huffmann-Baum



Die **Dekodierung** funktioniert in die andere Richtung: von rechts nach links. Dort wird die Bit-Sequenz eingegeben: wenn der erste Wert eine 0 ist, dann wird bei der Richtungsänderung der untere Zweig gewählt, wenn der Wert eine 1 ist, dann wird der obere Weg gewählt. Es werden so lange die Abzweigungen genommen, bis der Weg zum Ergebnis führt (veranschaulicht am Beispiel der Bit-Sequenz 100 [c]).

Huffmann-Baum – Beispiel Dekodierung mit 100 [c]



Sequenz:	dabbababababbbaaaababababccdd
Kodierbar mit 2 bit	a = 00 | b = 01 | c = 10 | d = 11
Größe	2 * 25 = **50 bit**
Binärcode:	101 0 11 11 0 11 0 11 0 11 11 11 0 0 0 0 11 0 11 0 11 100 100 101 101
Binärlayout:	10 * 1 + 10 * 2 + 3 * 3 + 2 * 3 = 10 + 20 + 9 + 6 = **45 bit**

mittlere Codewortlänge	1 * 0,4 + 2 * 0,4 + 3 * 0,12 + 3 * 0,08 = 0,4 + 0,8 + 0,36 + 0,24 = **1,8 bit**

Kompressionsrate	50 / 45 = **1,1111**
Platzersparnis	**10%**

Besonders effektiv ist diese Art der Kodierung bei überproportional dominanten Codewörtern: *niedrige Entropie.* Der schlimmste Fall für die Huffmann Kodierung ist es, wenn alle Symbole gleich häufig vorkommen.

Das untenstehende Beispiel sollte den besten Fall zeigen – es gibt nur ein (oder wenige) stark dominierende Symbole.

Kompression max.	aaaaaaaaaa
Binärcode:	1111111111
Binärlayout:	10 * 1 = **10 bit**

mittlere Codewortlänge	1 * 1 = **1 bit**

Jedoch muss man bedenken, dass diese Sequenz, die nur aus einem Symbol besteht, ohnehin mit 1 bit kodierbar ist: a = 0. Es gibt daher durch die Huffmann Kodierung keine Verbesserung bei dieser Sequenz. Entgegen der ursprünglichen Annahme stellt sich diese Sequenz als die am schlechtesten für die Huffmann Kodierung geeignetste Sequenz heraus, die mit einer Kompressionsrate von 1 die **minimale Kompression** liefert, dies entspricht einer Platzersparnis von 0%.

Kompression max.	aaaaaaaaaa
Kodierbar mit 1 bit	a = 0
Größe	1 * 10 = **10 bit**
Binärcode:	1111111111
Binärlayout:	10 * 1 = **10 bit**

Kompressionsrate	10 / 10 = **1**
Platzersparnis	**0%**

Schlecht wäre es theoretisch, wenn es mehrere Symbole gibt, die aber gleich oft in der Sequenz vorkommen. Das untenstehende Beispiel zeigt 5 Symbole, die jeweils 2-mal vorkommen, wodurch sich eine Kompressionsrate von 1,25 und eine Platzersparnis von 20% ergibt.

Sequenz	aabbccddee
Kodierbar mit 3 bit	a = 000 | b = 001 | c = 010 | d = 011 | e = 100
Größe	3 * 10 = **30 bit**
Binärcode:	10 10 01 01 00 00 111 111 110 110
Binärlayout:	2 * 2 + 2 * 2 + 2 * 2 + 2 * 3 + 2 * 3 = **24 bit**

mittlere Codewortlänge	2 * 0,2 + 2 * 0,2 + 2 * 0,2 + 3 * 0,2 + 3 * 0,2 = **2,4 bit**

Kompressionsrate	30 / 24 = **1,25**
Platzersparnis	**20%**

Da diese Sequenz jedoch nur mit 3 bit kodierbar ist, erweist sich hier die Huffmann Kompression als sinnvoll, da sie immerhin für eine Platzersparnis von 20% sorgt.

Der schlimmste Fall ist, wenn die Sequenz aus 10 unterschiedlichen Symbolen besteht. Dadurch werden die Verzweigungen des Huffman-Baumes so häufig, dass jede Sequenz aus 3 oder 4 Zeichen besteht.

Sequenz	abcdefghij
Kodierbar mit 4 bit	a = 0000 | b = 0001 | c = 0010 | d = 0011 | e = 0100 | f = 0101 | g = 0110 | h = 0111 | i = 1000 | j = 1001
Größe	4 * 10 = **40 bit**
Binärcode:	101 100 1111 1110 1101 1100 011 010 001 000
Binärlayout:	1 * 3 + 1 * 3 + 1 * 4 + 1 * 4 + 1 * 4 + 1 * 4 + 1 * 3 + 1 * 3 + 1 * 3 + 1 * 3 = **34 bit**

mittlere Codewortlänge	3 * 0,1 + 3 * 0,1 + 4 * 0,1 + 4 * 0,1 + 4 * 0,1 + 4 * 0,1 + 3 * 0,1 + 3 * 0,1 + 3 * 0,1 + 3 * 0,1 = **3.4 bit**

Kompressionsrate	40 / 34 = **1,1765**
Platzersparnis	**15%**

Entgegen der Annahme stellt sich hier jedoch heraus, dass die Platzersparnis nicht so groß ist, wie bei der darüberliegenden Variante.

Folgende Sequenz hat sich als die mit der **maximalen Kompression** herausgestellt:

Sequenz	aaaaaaaabc
Kodierbar mit 2 bit	a = 00 | b = 01 | c = 10
Größe	2 * 10 = **20 bit**
Binärcode:	1 1 1 1 1 1 1 1 01 00
Binärlayout:	8 * 1 + 1 * 2 + 1 * 2 = **12 bit**

mittlere Codewortlänge	1 * 0,8 + 2 * 0,1 + 2 * 0,1 = **1,2 bit**

Kompressionsrate	20 / 12 = **1,7**
Platzersparnis	**40%**

Es wurde gezielt diese Anzahl an Zeichen gewählt, da ab einer Anzahl von 3 Zeichen für eine Kodierung 2 bit benötigt werden. Die Kompressionsrate mit 1,7 führt zu einer Platzersparnis von 40%.

c) **Komprimieren Sie händisch die Sequenz 11110101011111000001111000101O (30 Stellen, n=2 Symbole: {0,1}) mittels Runlength Coding. Berechnen Sie die erzielbare Kompressionsrate. Wie kann das Runlength Coding auf eine Symbolmenge n>2 erweitert werden und was ist dabei zu beachten? Demonstrieren Sie Ihre Erweiterung an einem selbst gewählten BSP und berechnen Sie die erzielbare Kompressionsrate. Von welcher Beschaffenheit der Daten hängt die erzielbare Kompressionsrate ab?**

Beim Runlength Coding sind die Symbole stark von davorliegenden Symbolwert abhängig. Bei hoher Homogenität ist diese Art der Komprimierung daher besonders sinnvoll. Diese Art der Kompression spart Speicherplatz, indem die Anzahl der aufeinanderfolgenden Symbole gespeichert wird. Bei binären Sequenzen funktioniert dies einfacher, da das jeweilige Symbol nicht zusätzlich gespeichert werden muss, da es nur zwei Symbole gibt: 0 und 1. Um eine einheitliche Kompression sicherzustellen, sollte das Zählen immer mit dem Wert 0 begonnen werden.

unkomprimiert	**komprimiert**
11110101011111000001111000101O	041111155431111
30 Zeichen	15 Zeichen

Kompressionsrate: (30 / 15) : (15 / 15) = **2 : 1**
Die Anzahl der Zeichen der komprimierten Sequenz konnte um **50%** gegenüber der Originalanzahl verkleinert werden.

Bei binären Signalen (wie dem obigen Beispiel) kann die Anzahl der Werte 0 und 1 in abwechselnder Reihenfolge abgebildet werden. Wenn es sich jedoch um mehrere verschiedene Symbole handelt, muss nicht nur die Anzahl der hintereinander befindlichen Symbole, sondern auch deren ID bekanntgegeben werden.

unkomprimiert	komprimiert
aaabbbbbbccaaaacdddd	3a6b2c4a1c4d
20 Zeichen	12 Zeichen

Kompressionsrate: (20 / 12) : (12 / 12) = **1,7 : 1**
Die Anzahl der Zeichen der komprimierten Sequenz konnte um **40%** gegenüber der Anzahl im originalen String verkleinert werden.

Diese Art der Kompression funktioniert nur dann gut, wenn es in der Sequenz lange Ketten von gleichen Symbolen gibt. Dabei spielt es keine Rolle, wie oft diese Symbole zusammengerechnet vorkommen. Bei einer Sequenz mit ständig wechselnden Zeichen, würde diese Kompression daher nicht funktionieren und könnte sogar das Ergebnis verschlechtern:

unkomprimiert	komprimiert
abaacdabbcdaddccabad	1a1b2a1c1d1a2b1c1d1a2d2c1a1b1a1d
20 Zeichen	32 Zeichen

Kompressionsrate: (20 / 32) : (32 / 32) = **0,6 : 1**
Die Anzahl der Zeichen der komprimierten Sequenz wurden um **60%** gegenüber der Anzahl im originalen String **vergrößert**.

d) **Berechnen Sie die Entropie der 30-stelligen Sequenz 221111226611122333345645112111 (n=6 Symbole: {1,2,3,4,5,6}). Bei welcher 10-stelligen Sequenz ist die Entropie maximal bzw. bei welcher 10-stelligen Sequenz ist die Entropie minimal? Welche Auswirkungen hat die Entropie in Bezug auf die erzielbare Kompression? Ist für die erzielbare Kompressionsrate dabei immer lediglich die Auftrittswahrscheinlichkeit entscheidend?**

Bei der Entropie spielt die Sortierung der Zeichen in einer Sequenz keine Rolle, es geht nur darum, wie oft die Zeichen vorkommen. Bei Daten mit niedriger Entropie kann eine verlustfreie Kompression stattfinden. Bei einer hohen Entropie kommen alle Symbole gleich oft in einer Sequenz vor – dies ist der schlimmste Fall.

Eintritts-Wahrscheinlichkeiten	
p(2) = 7/30	0,2
p(1) = 12/30	0,4
p(6) = 3/30	0,1
p(3) = 4/30	0,1
p(4) = 2/30	0,1
p(5) = 2/30	0,1

$H = -[p(2) * \log_2(p(2)) + p(1) * \log_2(p(1)) + p(6) * \log_2(p(6)) + p(3) * \log_2(p(3)) + p(4) * \log_2(p(4)) + p(5) * \log_2(p(5))]$
$H = -[0,2 * \log_2(0,2) + 0,4 * \log_2(0,4) + 0,1 * \log_2(0,1) + 0,1 * \log_2(0,1) + 0,1 * \log_2(0,1) + 0,1 * \log_2(0,1)]$
$H = -[-2,321928] =$ **2,3**

Um eine minimale Entropie zu erhalten, darf die Sequenz keine Unordnung enthalten. Die Entropie ist am geringsten, wenn also alle Werte einer Sequenz gleich sind. Nach der Berechnung ergibt sich für die Entropie der Wert 0. Eine maximal große Entropie zu erhalten, müssen alle Zeichen in einer Sequenz gleich oft vorkommen, um eine maximale Unordnung zu erreichen werden also nur unterschiedliche Zeichen gewählt. Die Entropie beträgt in diesem Fall 3,3.

min. Entropie	**max. Entropie**
1111111111	0123456789
$H = -[1 * \log_2(1)]$	$H = -[(0{,}1 * \log_2(0{,}1)) * 10]$
H = **0**	H = **3,3**

Das Zusammenspiel zwischen Entropie und Homogenität (Ähnlichkeit eines Symbols zur Nachbarschaft) ist für die erzielbare Kompressionsrate entscheidend – es hängt jedoch immer von der Art der Kompression ab, ob das erzielte Ergebnis eine Verbesserung liefert. Während beim Runlength Coding, eine hohe Homogenität entscheidend für ein gutes Resultat ist, so kommt es beim Huffmann Coding auf eine niedrige Entropie an. Grundsätzlich hängen Entropie und Homogenität aber zusammen: wenn die Homogenität hoch ist, dann ist die Entropie eher niedrig.