Humboldt-Universität zu Berlin
Institut für Informatik
Prof. Dr. Jens-Peter Redlich
M. Kabierski, A. Heuts, Dr. W. Müller, M. Sänger

Berlin, den 18.01.2021

Übungen zur Vorlesung
Grundlagen der Programmierung

Übungsblatt 5

Abgabe: Die Abgabe Ihrer Lösungen erfolgt über den Moodle-Kurs der Vorlesung (https://moodle.hu-berlin.de/course/view.php?id=98193). Nutzen Sie die dort im Abschnitt Übung angegebene **Aufgabe 5.** Die Abgabe ist bis zum **01.02.2021** um **09:15 Uhr** möglich. **Die Abgabe erfolgt in Gruppen.** Bitte bilden Sie im Moodle Gruppen! Bitte verwenden Sie **keine** (noch nicht in der Vorlesung eingeführten) **Java-Bibliotheken**.

Hinweise:

- Achten Sie darauf, dass Ihre Java-Programmdateien mittels des UTF-8-Formats (ohne byte order marker (BOM)) codiert sind und keinerlei Formatierungen enthalten.
- Verzichten Sie auf eigene Packages bei der Erstellung Ihrer Lösungen, d. h. **kein** `package`-Statement am Beginn Ihrer Java-Dateien.
- Quelltextkommentare können die Korrektoren beim Verständnis Ihrer Lösung (insbesondere bei inkorrekten Abgaben) unterstützen; sind aber nicht verpflichtend.
- Testen Sie Ihre Lösung bitte auf einem Rechner aus dem Informatik Computer-Pool z. B. gruenau6.informatik.hu-berlin.de (siehe Praktikumsordnung: Referenzplattform)

Aufgabe 1 (Labyrinth) — 10 Punkte

Gegeben sei ein Labyrinth der Dimension *n* x *n*, in dem mithilfe von Backtracking ein Weg von einem Start- zu einem Zielpunkt gefunden werden soll. Dabei kann das Labyrinth nach links oder nach unten durchlaufen werden. Wenn ein Zug nach links in eine Sackgasse führt, dann wird dieser zurückgenommen. Das wird solange wiederholt, bis ein Zug nach unten möglich ist. Das Beispiel in Abbildung 1.1 demonstriert das beschriebene Vorgehen.

Angefangen in der oberen rechten Ecke soll die untere linke Ecke erreicht werden. Zunächst wird das Labyrinth nach links durchlaufen, bis eine Sackgasse (S) erreicht wird. Ein Zug nach links wird zurückgenommen, bis ein Zug nach unten möglich ist. Da anschließend kein Zug nach links möglich ist, werden weitere Züge nach unten durchgeführt. Es folgt ein Zug nach links, bis das Ziel (x) erreicht wird.

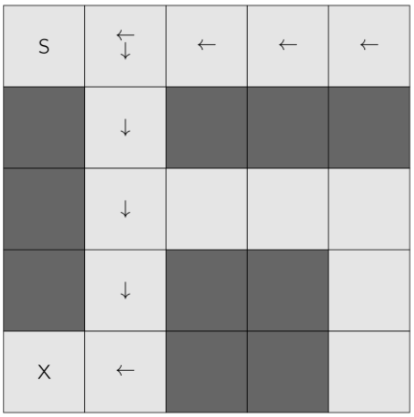


Abbildung 1.1: Beispiellabyrinth

Eine Lösung in Java kann mithilfe von Rekursion und Backtracking realisiert werden. Dazu wird jeweils ein Zug nach links bzw. nach unten vollzogen, bis

1. die Grenze des Labyrinths erreicht wurde,
2. der Zug in eine Sackgasse führt oder
3. das Ziel erreicht wurde.

Im 1. und 2. Fall wird (sofern möglich) ein Zug nach unten vollzogen. Ansonsten wird der Zug „zurückgenommen“ und stattdessen (sofern möglich) ein Zug nach unten gemacht. Andernfalls wird ein weiterer Zug zurückgenommen. Im 3. Fall konnte ein Weg gefunden werden und das Labyrinth ist somit lösbar.

Ergänzen Sie in der vorgegebenen Java-Datei `MazeSolver.java`, die fehlenden Implementierungen für die Methoden `solve` und `draw`, welche das Lösen des Labyrinths (gemäß der oben genannten Regeln) und das Anzeigen der in `int[][] maze` repräsentierten Konfiguration, umsetzen sollen. Als Repräsentation des Labyrinths dient ein mehrdimensionales Array, wobei die erste Dimension die Zeilen und die zweite Dimension die Spalten des Labyrinths repräsentieren. Dabei gibt der Wert `2` den Start- und der Wert `3` den Zielpunkt an. Der Wert `1` repräsentiert ein begehbares Feld im Labyrinth und der Wert `0` steht für ein Hindernis. Das Labyrinth aus Abbildung 1.1 würde bspw. wie folgt repräsentiert:

```
int[][] maze = {
      {1, 1, 1, 1, 2},
      {0, 1, 0, 0, 0},
      {0, 1, 1, 1, 1},
      {0, 1, 0, 0, 1},
      {3, 1, 0, 0, 1}
};
```

Implementieren Sie die in Ihrer Klasse `MazeSolver` die von außen nutzbare Methode `public static boolean solve(int[][] maze, int row, int col)`. Die beiden Parameter vom Typ `int` repräsentieren dabei jeweils den aktuellen Index der Zeile bzw. Spalte. Der Rückgabewert vom Typ `boolean` gibt an, ob das Labyrinth lösbar ist, d. h. ob ausgehend vom Startpunkt der Zielpunkt erreicht werden kann. Implementieren Sie die Methode `solve` so, dass der schlussendlich gewählte Pfad durch das Labyrinth im Array visuali-

siert wird. Dazu sollen die Werte an den Stellen der jeweils gewählten Einträge durch den Wert 2 ersetzt werden.

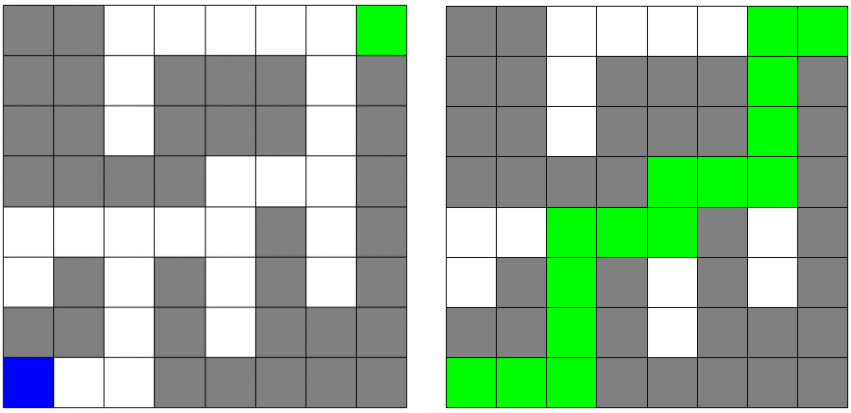


Abbildung 1.2: Labyrinth vor und nach Ausführung der Methode `solve()`

Bei der Implementierung der Methode `public static void draw(int[][] maze)` zum Anzeigen beachten Sie bitte folgende Konventionen:

- Passen Sie Größe Zeichenfeldes mithilfe der beiden Methoden `setXscale(double min, double max)` und `setYscale(double min, double max)` der Bibliothek `StdDraw` an.
- Der Startpunkt ist oben rechts, der Zielpunkt unten links.
- Jedes in `maze` enthaltene Zahl wird als Quadrat in der folgenden Farbe (`0` → grau, `1` → weiß, `2` → grün, `3` → blau) dargestellt und in mit einem schwarzen Rahmen umgeben.

Hinweise zur Bearbeitung:

- Sie können davon ausgehen, dass nur quadratische Labyrinthe übergeben werden.
- Bei der Entwicklung Ihrer Lösung kann es ggf. hilfreich sein den Zustand des Labyrinths (in der Entwicklungsphase) häufiger darzustellen, um so die Vorgehensweise Ihrer Lösung grafisch nachzuvollziehen.
- Testen Sie Ihre Lösungen mithilfe der Klasse `MazeTester`, welche Testfälle für sechs verschiedene lösbare oder unlösbare Labyrinthe beinhaltet.
- Zum Testen liegen `MazeSolver` und `MazeTester` im gleichen Verzeichnis und gdp.stdlib.StdDraw wird an diesem Platz erwartet.
- Laden Sie bitte genau die von Ihnen mit den Implementierungen ergänzte Datei `MazeSolver.java` in Moodle hoch.

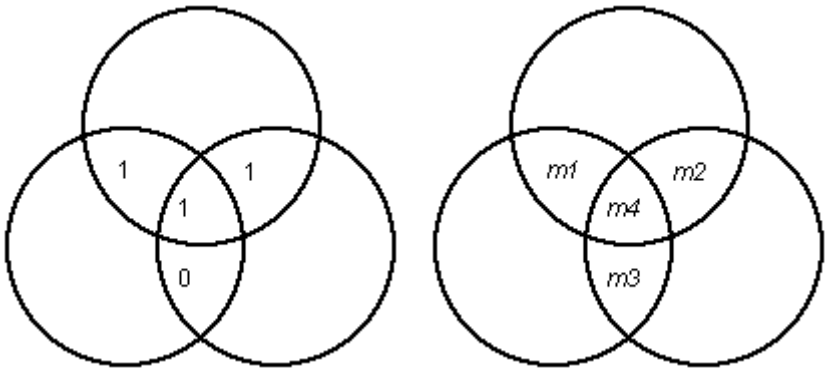
Aufgabe 2 (Hamming Codes in TOY) 10 + 2 Punkte

Schreiben Sie ein TOY-Programm, um Daten mittels Hamming-Codes zu kodieren. Schreiben Sie außerdem ein TOY-Programm, welches codierte Daten, die beschädigt wurden, korrigiert.

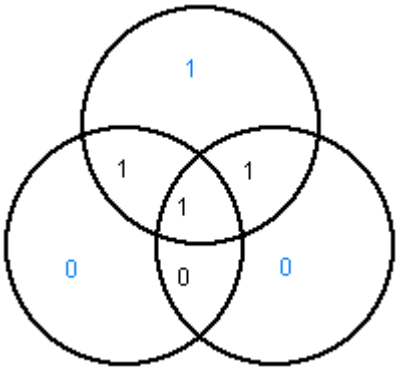
Hintergrund: Fehlerkorrekturcodes ermöglichen es Daten, die über einen gestörten Kommunikationskanal übertragen werden, fehlerfrei zu empfangen. Um dies zu erreichen, fügt der Absender redundante Informationen zu der Nachricht hinzu, sodass, selbst wenn ein Teil der Ausgangsdaten bei der Übertragung beschädigt wird, der Empfänger immer noch die ursprüngliche Nachricht empfangen kann. Übertragungsfehler sind häufig und können z. B. durch Kratzer auf einer CD entstehen oder bei der drahtlosen Kommunikation durch atmosphärische Störung auftreten. In einer gestörten Umgebung können Fehlerkorrekturcodes den Durchsatz einer Kommunikationsverbindung erhöhen, da keine Notwendigkeit besteht, die Nachricht erneut zu übertragen, wenn sie während der Übertragung teilweise beschädigt wird. Aus diesem Grund werden Fehlerkorrekturcodes in vielen gängigen Systemen verwendet: Speichermedien (CD, DVD, DRAM), Mobilfunk (Mobiltelefone, drahtlose Verbindungen), Digital-TV und High-Speed- Modem (ADSL, xDSL).

Hamming-Codes: Ein Hamming-Code ist eine bestimmte Art von Fehlerkorrekturcode, der die Erkennung und Korrektur von Einzel-Bit-Übertragungsfehlern ermöglicht. Hamming-Codes werden in vielen Anwendungen benutzt, bei denen solche Fehler üblich sind, einschließlich DRAM-Speicherchips und Satellitenkommunikationshardware. Hamming-Codes arbeiten, indem sie immer vier Nachrichten-Bits lesen (`m1`, `m2`, `m3` und `m4`) und dann drei Parity-Bits (`p1`, `p2` und `p3`) einfügen. Wenn eines dieser sieben Bits während der Übertragung beschädigt wird, kann der Empfänger den Fehler erkennen und die ursprünglichen vier Nachrichtenbits wieder rekonstruieren. Dies nennt sich Einbitfehlerkorrektur, da höchstens ein Bit pro gesendeter Dateneinheit korrigiert werden kann. Der Aufwand für diese Methode ist eine 1,75-fache Erhöhung der Menge an Bandbreite, da wir drei zusätzliche Paritätsbits für alle vier Nachrichten-Bits benötigen. Im Vergleich mit naiven Ansätzen, wie z.B. das Senden von drei Kopien von jedem Bit, ist dies jedoch effizienter.

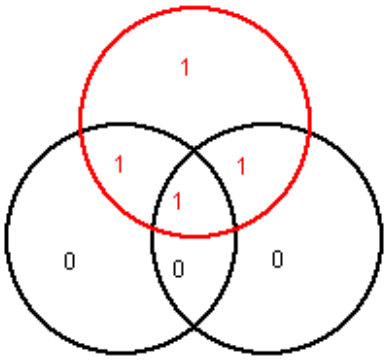
Bevor wir die Algebra der Hamming-Codes erklären, visualisieren wir zunächst die Berechnung dieser Paritätsbits mit Venn-Diagrammen. Als Beispiel nehmen wir die Nachricht `1101`. Jedes der vier Nachrichten-Bits schreiben wir in einen bestimmten der vier Kreuzungsbereiche dreier paarweise überlappender Kreise, wie unten dargestellt:



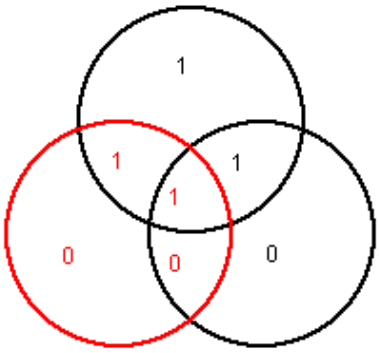
Der Hamming-Code fügt nun drei Parity-Bits hinzu, sodass jeder Kreis gerade Parität besitzt.



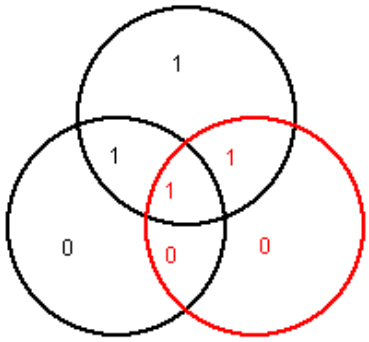
Das heißt, die Summe der vier Bits in jedem Kreis ist gerade:



1 + 1 + 1 + 1 = 0



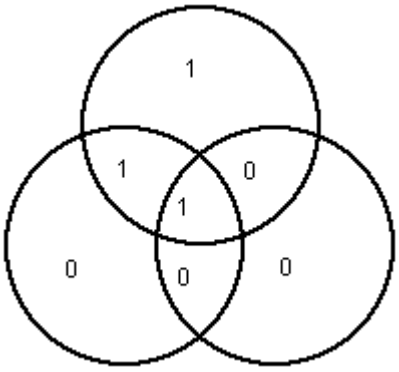
0 + 1 + 0 + 1 = 0



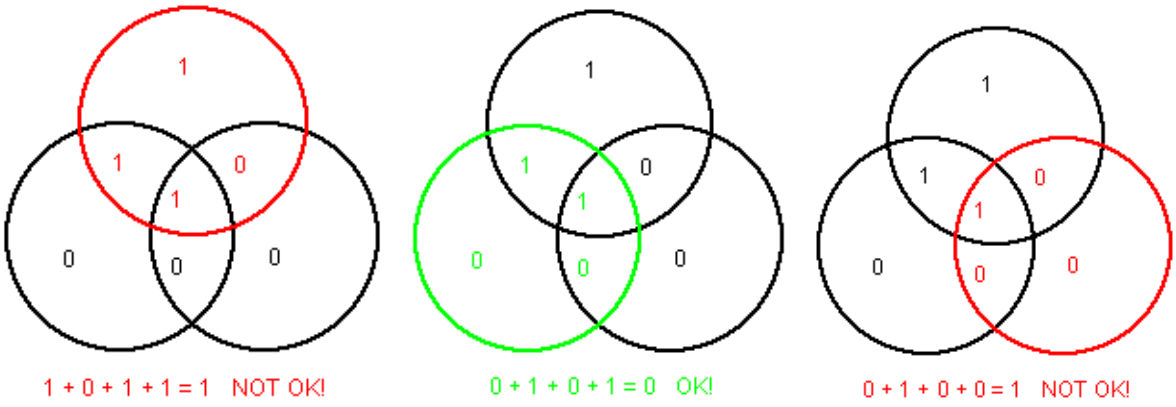
0 + 1 + 0 + 1 = 0

In diesem Fall senden wir `1101100` da die drei Paritätsbits `1` (oben), `0` (links), und `0` (rechts) sind.

Nun stellen Sie sich vor, dieses Bild wird per Modem über einen verrauschten Kommunikationskanal übertragen und dass dabei ein Bit beschädigt wird, sodass folgendes Bild bei der Empfangsstation ankommt (entsprechend `1001100`):



Der Empfänger erkennt durch Prüfen der Parität, dass ein Fehler in einem der drei Kreisen aufgetreten ist. Darüber hinaus kann der Empfänger bestimmen, wo der Fehler aufgetreten ist (das zweite Bit), den Fehler korrigieren und so die vier ursprünglichen Nachricht-Bits empfangen!



Da die Paritätsprüfung für den ersten Kreis und letzten Kreis gescheitert ist, aber der mittlere Kreis in Ordnung war, gibt es nur ein Bit, das verantwortlich sein kann und das ist `m2`. Wenn das mittlere Bit `m4` beschädigt wird, dann schlagen alle drei Paritätsprüfungen fehl. Wenn ein Paritäts-Bit selbst beschädigt ist, dann wird nur eine Paritätsprüfung fehlschlagen. Wenn die Datenverbindung so gestört ist, dass zwei oder mehr Bits gleichzeitig beschädigt werden, wird unsere Codierung nicht funktionieren. Anspruchsvollere Arten von Fehlerkorrekturcodes können und behandeln auch solche Situationen.

Natürlich werden in der Praxis nur die 7 Bits übertragen und nicht die Kreisdiagramme.

Aufgabe

Teil 1 [5 Punkte]:
Schreiben Sie ein TOY-Programm `encode.toy` das eine binäre Nachricht mit dem oben beschriebenen Schema verschlüsselt. Ihr Programm sollte wiederholend vier Bits `m1`, `m2`, `m3` und `m4` von der TOY-Standardeingabe lesen und die sieben Bits `m1`, `m2`, `m3`, `m4`, `p1`, `p2`, `p3` auf der TOY-Standardausgabe ausgeben, wobei:

- `p1 = m1 ^ m2 ^ m4`
- `p2 = m1 ^ m3 ^ m4`
- `p3 = m2 ^ m3 ^ m4`

Hinweis: `^` ist der XOR-Operator in Java und TOY. Dies entspricht dem oben beschriebenen Paritäts-Begriff.

Teil 2 [5 Punkte]:
Schreiben Sie ein TOY-Programm `decode.toy`, welches eine Hamming- codierte Nachricht korrigiert. Ihr Programm sollte wiederholt sieben Bits `m1`, `m2`, `m3`, `m4`, `p1`, `p2`, `p3` von der TOY-Standard-Eingabe lesen und die vier Nachrichten-Bits auf der TOY-Standardausgabe ausgeben.

Zur Erinnerung: Um zu bestimmen, ob und welche, der Nachrichten-Bits überhaupt beschädigt sind, führen Sie die folgenden drei Paritätsprüfungen durch:

(1) `p1 = m1 ^ m2 ^ m4`
(2) `p2 = m1 ^ m3 ^ m4`
(3) `p3 = m2 ^ m3 ^ m4`

Hier ist eine Zusammenfassung dessen, was die Ergebnisse bedeuten:

- Wenn genau eine oder keine der Paritätsprüfungen versagen, dann sind alle vier Nachrichten-Bits korrekt.
- Wenn Kontrollen 1 und 2 fehlschlagen (aber nicht 3), so ist das Bit `m1` falsch.
- Wenn Kontrollen 1 und 3 fehlschlagen (aber nicht 2), so ist das Bit `m2` falsch.
- Wenn Prüfungen 2 und 3 fehlschlagen (aber nicht 1), so ist das Bit `m3` falsch.
- Wenn alle drei Prüfungen fehlschlagen, so ist das Bit `m4` falsch.

Wenn nötig, korrigieren (d.h. invertieren) Sie die beschädigten Nachrichten-Bits `m1`, `m2`, `m3` und `m4`.

Eingabeformat: Der Einfachheit halber werden wir jedes Bit einzeln (`0000` oder `0001`) statt 16 Bits gepackt pro TOY-Wort (wie es bei einer echten Anwendung durchgeführt werden würde) übertragen. Ihr Programm sollte das Einlesen wiederholen, bis `FFFF` auf TOY Standardeingabe gelesen wird. Sie können auch davon ausgehen, dass die Anzahl der übertragenen Bits (ohne das abschließende Wort) ein Vielfaches von vier bei der Codierung ist, und ein Vielfaches von sieben bei der Korrektur.

Hier sind Beispiele für Eingabedateien für `encode.toy` und `decode.toy`:

```
encode.toy input file
---------------------
0001 0001 0000 0001
0001 0001 0001 0000
0001 0001 0001 0001
FFFF
```

```
decode.toy input file
---------------------------------
0001 0000 0000 0001 0001 0000 0000
0000 0001 0001 0000 0000 0000 0000
0001 0001 0001 0001 0001 0001 0000
FFFF
```

Hinweise zur Bearbeitung:

- Zur Vereinfachung schauen Sie sich `hamming.zip` an. Dies ist eine Implementierung in Java und enthält die Eingabedateien `input[47].txt` und `all[47].txt`.
- Hinweise zur Verwendung der grafischen Toy-Maschine und der Kommandozeilenversion finden Sie im Moodle-Kurs unter den Punkten:
  „FAQ / Tutorials“ → „Toy Machine“
- **Abgabe über Moodle:** Laden Sie die Dateien: `encode.toy` und `decode.toy` über die Moodle-Webseite hoch.
- **Zusatzpunkte [2 Punkte]:** Schreiben Sie ein TOY-Programm für Teil 2, welches weniger als 40 Worte im TOY-Hauptspeicher verwendet. Sie sollten jede (nicht leere) Zeile Ihres Programms zählen. (Die Aufgabe ist auch mit weniger als 35 Zeilen lösbar.)