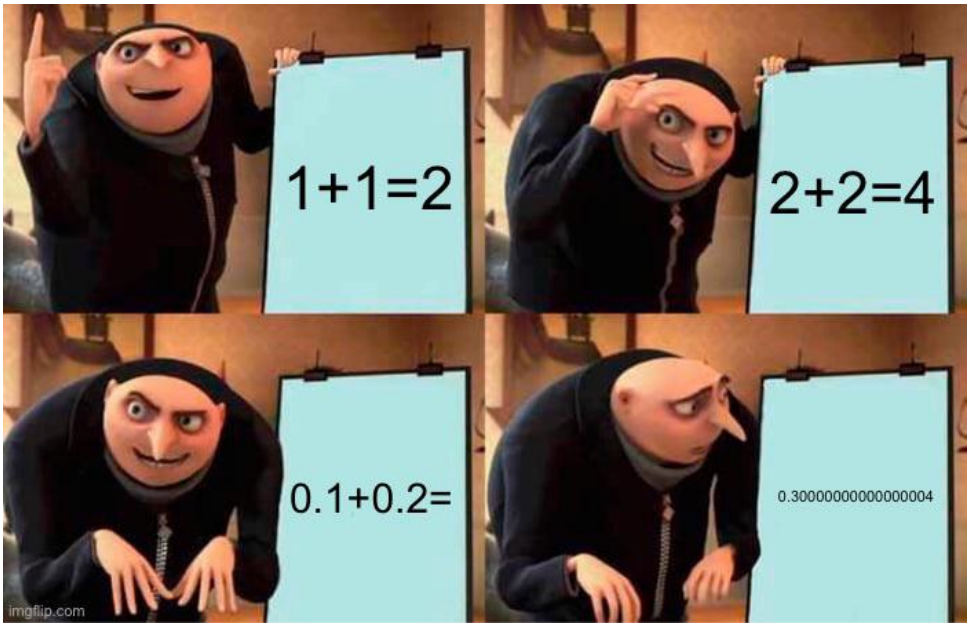
EF Informatik

Datentypen

Kantonsschule Romanshorn

Andreas Schärer



1 Integers

Die Ariane 5 Katastrophe

Am 4. Juni 1996 startete in Kourou (Französisch-Guayana) zum ersten Mal eine Rakete des Typs **Ariane 5**. Nach 37 Sekunden begann sie zu taumeln und sprengte sich selbst und damit die vier Forschungssatelliten an Bord – rund 370 Millionen Dollar waren im Eimer.

Die Ursache war weder ein Materialfehler noch ein Sturm, sondern ein Softwarefehler! Das Navigationssystem wollte einen Messwert für die horizontale Geschwindigkeit von einer 64-Bit-Gleitkommazahl in eine vorzeichenbehaftete 16-Bit-Ganzzahl umwandeln. Dazu hat man unverändert den Code der Vorgängerrakete Ariane 4 übernommen, der sich dort bewährt hatte. Die Ariane 5 war aber deutlich schneller als ihre Vorgängerin, und der Geschwindigkeitswert war schlicht zu gross, um in 16 Bit gespeichert zu werden. Man spricht in einem solchen Fall von einem **Überlauf**, englisch **Overflow**: Eine Zahl ist zu gross für den Datentyp, in dem sie gespeichert werden soll – und was dann passiert, schauen wir uns unten genauer an. Das Navigationssystem meldete einen Fehler, das Ersatzsystem (mit identischer Software) ebenfalls, und die Steuerung interpretierte die Fehlermeldung als Flugdaten. Die Triebwerke wurden fälschlicherweise schräg gestellt, worauf die Rakete auseinanderbrach und damit die Selbstzerstörung auslöste.

64-Bit-Gleitkommazahlen und 16-Bit-Ganzzahlen sind unterschiedliche **Datentypen**. Datentypen scheinen also etwas ziemlich Wichtiges zu sein und deshalb wollen wir uns jetzt mit diesen genauer auseinandersetzen. Damit stellen wir sicher, dass dir, wenn du einmal ein Rocket Scientist bist, nicht wegen eines solchen Fehlers die Millionen um die Ohren fliegen!

Datentypen in Python

Beim Programmieren mit **Python** hast du dir über Datentypen wahrscheinlich noch nie ernsthaft Gedanken machen müssen. Du schreibst `x = 42` und Python kümmert sich um den Rest. Man kann auch ohne Problem den Datentyp einer Variablen ändern. Zum Beispiel ist der folgende Code gar kein Problem:

```python
# Python
a = 42                    # Datentyp: Ganze Zahl (integer)
a = 3.14159               # Datentyp: Gleitkommazahl (float)
a = "Ich bin ein String!" # Datentyp: String
```

Man kann also jederzeit einer Variablen einen neuen Wert eines anderen Datentyps zuweisen. Daher nennt man Python eine **dynamisch typisierte** Sprache.

Datentypen in C-Sprachen

In diesem Kurs arbeiten wir aber mehrheitlich mit **C-Sprachen**, genauer C# ('C Sharp') und C++ ('C Plus Plus'), bei welchen man selbst sicherstellen muss, dass jede Variable vom richtigen Datentyp ist:

```
1 // C#
2 int a;
3 a = 42;
```

Zuerst muss die **Variable** deklariert werden `int a;`. Wir sagen dem Computer also: 'Wir möchten eine Variable mit Namen a, mit welcher man Werte vom Datentyp int (ganze Zahlen) speichern kann'. Die Zuweisung eines anderen Werts ist dann nicht gestattet:

```
1 // C#
2 int a;
3 a = 42;
4 a = 3.14159;
```

Die letzte Zeile produziert einen Fehler, da 3.14159 keine ganze Zahl (int), sondern eine Gleitkommazahl (float) ist, und damit *nicht* in der als int deklarierten Variable a gespeichert werden kann! Daher nennt man C-Sprachen **statisch typisierte** Sprachen.

Der **Compiler** ist das Programm, das deinen Quellcode in Maschinencode übersetzt; diesen Vorgang nennt man *Kompilieren*. Bei statisch typisierten Sprachen wie den C-Sprachen findet der Compiler Fehler bei den Datentypen bereits dabei – das Programm wird gar nicht erst gestartet. Bei dynamisch typisierten Sprachen fallen solche Fehler erst zur **Laufzeit** auf, also während der Ausführung.

Für ganze Zahlen gibt es dabei nicht *einen* Datentyp, sondern mehrere. Sie unterscheiden sich darin, wie viele **Bytes** sie im Arbeitsspeicher belegen und ob auch negative Zahlen erlaubt sind. Das `u` am Anfang steht für **unsigned**, also 'ohne Vorzeichen': damit lassen sich nur Werte ≥ 0 speichern, dafür reicht der Wertebereich nach oben doppelt so weit.

C#	Arduino (C++)	negativ?	Platz in C#	Platz auf Arduino
`int`	`int`	ja	4 Bytes (32 Bit)	2 Bytes (16 Bit)
`uint`	`unsigned int`	nein	4 Bytes (32 Bit)	2 Bytes (16 Bit)
`long`	`long`	ja	8 Bytes (64 Bit)	4 Bytes (32 Bit)
`ulong`	`unsigned long`	nein	8 Bytes (64 Bit)	4 Bytes (32 Bit)

Bemerkungen:

- Wie in der Tabelle zu sehen ist, kann z.B. ein `int` auf unterschiedlichen Plattformen unterschiedlich gross sein. Derselbe Code kann deshalb auf dem PC funktionieren und auf dem Arduino falsche Resultate liefern!
- Neben `(u)int` und `(u)long` gibt es noch die Datentypen `(s)byte` und `(u)short`, für den Moment ignorieren wir diese aber.

Aufgaben von Hand

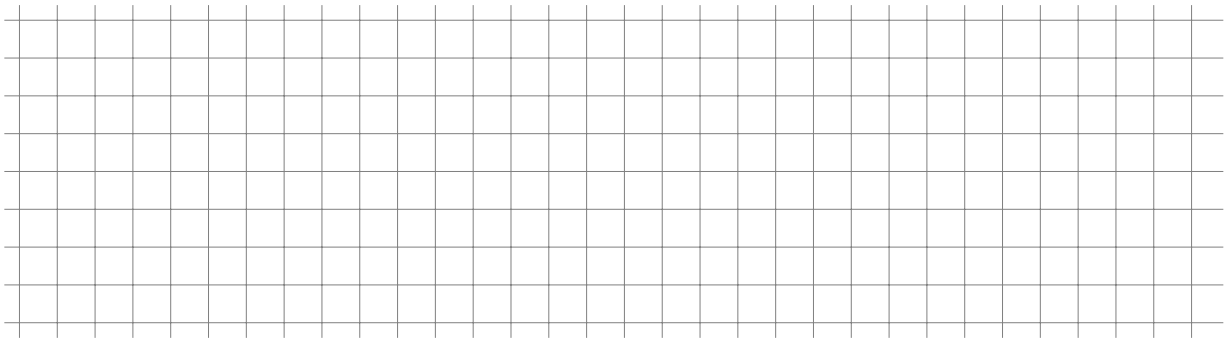
Löse die Aufgaben in diesem Abschnitt mit Stift und Taschenrechner, aber ohne Code.

Aufgabe A1

Erinnerung Binärsystem. Ein **Bit** ist die kleinste Informationseinheit: es ist entweder 0 oder 1. Acht Bits ergeben ein **Byte**, z.B. $1001\,1010_2$. Wandle die folgenden Binärzahlen *von Hand* ins Dezimalsystem um. Jeder Rechenschritt muss ersichtlich sein.

a) 1011_2

b) $1001\,1010_2$



Aufgabe A2

Für eine Variable stehen magere 4 Bit zur Verfügung. Bestimme jeweils, wie viele verschiedene Zahlen damit dargestellt werden können, sowie die kleinste und die grösste Zahl:

a) für einen `unsigned int` (nur Zahlen ≥ 0)

b) für einen `int` (auch negative Zahlen)

Aufgabe A3

Nun stehen nicht 4, sondern allgemein n Bits zur Verfügung. Stelle für `unsigned int` und `int` je eine **Formel** auf für die Anzahl darstellbarer Zahlen sowie für die kleinste und die grösste Zahl.

Aufgabe A4

Verwende deine Formeln und die Tabelle von oben:

a) Was ist die grösste Zahl, die man in C# in einem `int` speichern kann?

b) Und die grösste Zahl in einem `int` auf dem Arduino Uno?

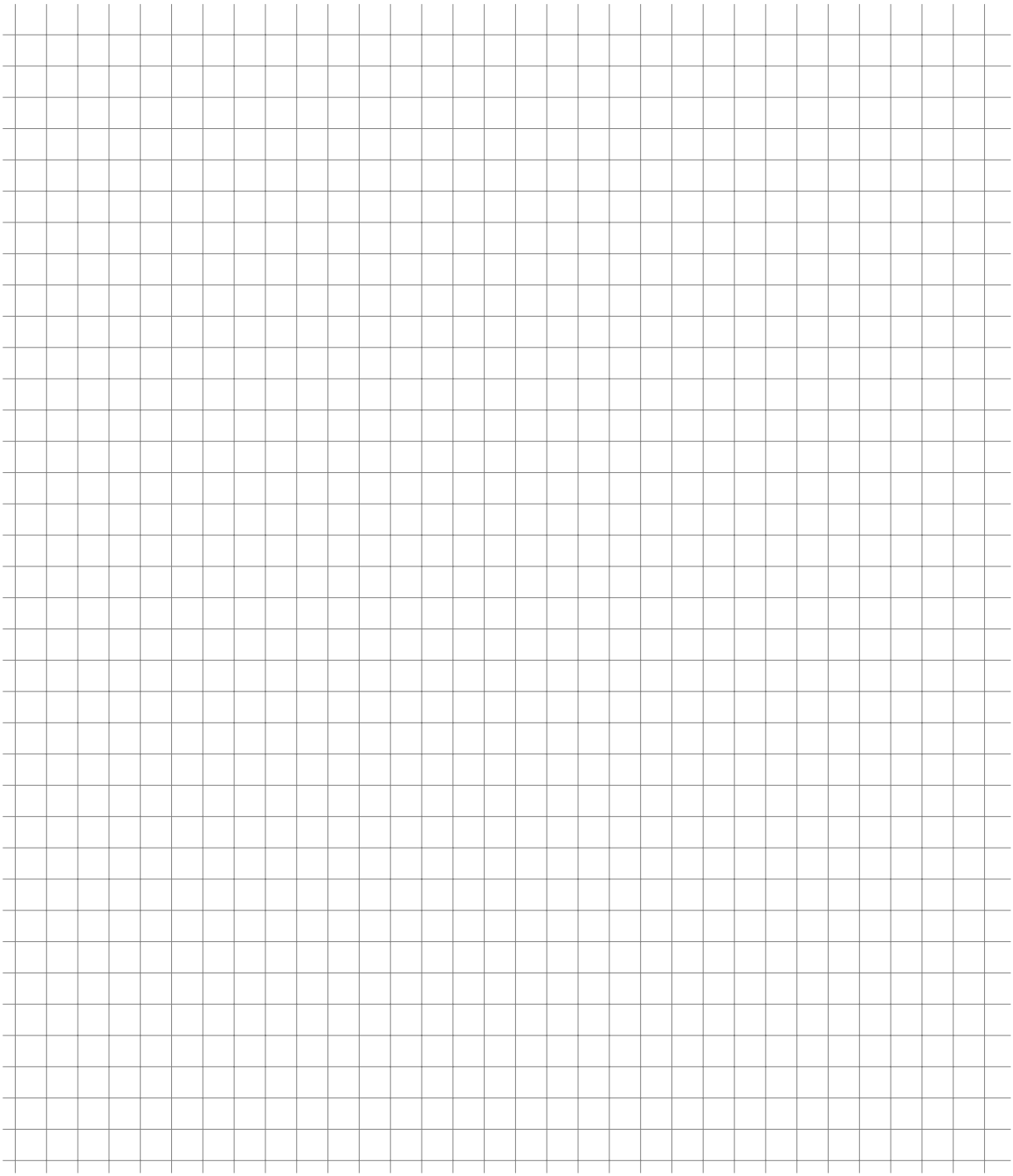
Aufgabe A5

Die Arduino-Funktion `millis()` gibt an, wie viele **Millisekunden** der Arduino bereits läuft. Du möchtest diesen Wert in einer eigenen Variablen speichern.

a) Welchen Datentyp wählst du: `unsigned int` oder `unsigned long`? Begründe.

b) Wie lange darf der Arduino laufen, bis der Wert nicht mehr in die Variable passt? Rechne beide Varianten aus und gib das Resultat in einer sinnvollen Einheit an.

c) Warum macht es hier Sinn, die 'unsigned' Variante zu wählen?



Aufgabe A6

Vergleich dynamische vs. statische Typisierung. Fülle die Tabelle mit je mindestens **zwei** Stichworten pro Feld aus.

	Dynamisch (Python)	**Statisch (C#, C++)**
Vorteile		
Nachteile		

Beantworte anschliessend: Für welche Art von Projekt würdest du eher Python wählen, für welche eher C#?

Aufgaben mit Code

Löse die Aufgaben mithilfe von einfachen Codes. Falls du Python und/oder C# auf deinem Computer installiert hast, verwende diese Installationen. Ansonsten verwende einen Online-Compiler wie z.B.:

- Python: `https://webtigerpython.ethz.ch`
- C#: `https://dotnetfiddle.net`

Aufgabe A7

Deine ersten Zeilen C# (nur für SuS ohne C#-Erfahrung). Das Programmieren in C# ist dem Programmieren in Python recht ähnlich – ein paar Dinge sind aber anders: Jede Anweisung endet mit einem **Semikolon** `;`, ausgegeben wird mit `Console.WriteLine(...)` statt mit `print(...)`, und jede Variable muss mit ihrem Datentyp **deklariert** werden.

Hinweis: Je nach Online-Compiler ist bereits ein Gerüst mit `class Program` und `static void Main()` vorgegeben. Schreibe deinen Code dann einfach innerhalb der geschweiften Klammern von `Main`.

a) Tippe (nicht Copy-Paste!) dein erstes C#-Programm ab und führe es aus:

```
1 Console.WriteLine("Hallo Kanti Romanshorn!");
```

b) Lösche das Semikolon am Zeilenende und führe das Programm nochmals aus. Notiere die **exakte** Fehlermeldung. Was lernst du daraus?

c) Setze das Semikolon wieder ein. Deklariere nun zwei Variablen vom Typ `int` mit den Werten 7 und 3 und lass dir ihre **Summe** ausgeben.

d) Erweitere dein Programm so, dass es auch **Differenz**, **Produkt** und **Quotient** der beiden Zahlen ausgibt. Vergleiche das Resultat des Quotienten mit dem, was dein Taschenrechner sagt. Was fällt dir auf?

e) Sorge zum Schluss dafür, dass die Ausgabe nicht nur `10` lautet, sondern `7 + 3 = 10`. Tipp: Mit `+` kann man auch Text aneinanderhängen.

Aufgabe A8

Statische Typisierung in C#. In einer Aufgabe oben hast du bereits bestimmt, dass 2 147 483 647 die grösste Zahl ist, die in einem C#-`int` gespeichert werden kann. Diesen Maximalwert wie auch den Minimalwert kann man übrigens auch direkt abfragen:

```
1 Console.WriteLine(int.MaxValue);
2 Console.WriteLine(int.MinValue);
```

Jetzt schauen wir nach, was passiert, wenn man diese Grenze des Maximalwerts *überschreitet.*

a) Betrachte den folgenden Code:

```
1 int large;
2 large = 2147483647;
3 Console.WriteLine(large);
4 large = large + 1;
5 Console.WriteLine(large);
```

Führe den Code noch *nicht* aus. Überlege dir zuerst, was jede Zeile macht. Was erwartest du für eine Ausgabe resp. Fehler?

b) Tippe nun den Code ab (kein Copy-Paste!) und führe ihn aus. Hattest du recht?

c) Warum ist das Verhalten des Codes in diesem Fall ziemlich gefährlich und kann zu hässlichen Problemen (#Ariane5) führen?

Aufgabe A9

Datentypen in Python: Auch wenn man sich in Python im Allgemeinen nicht so viele Gedanken über Datentypen machen muss, gibt es Anwendungen, in denen dies anders ist: Ein Temperatursensor hat während einer Nacht Messwerte aufgezeichnet. Weil er manchmal gestört wurde, stehen in der Liste nicht nur Zahlen, sondern auch Fehlermeldungen:

```
1 messwerte = [12.5, 14, "Fehler", 13.2, None, 11, "n/a", 15.8]
```

Beachte, dass die Liste sowohl `int` als auch `float` enthält – beides sind gültige Messwerte. Schreibe eine Funktion `auswerten(werte)`, welche mithilfe von `isinstance` die brauchbaren Messwerte heraussucht und Folgendes ausgibt:

a) wie viele gültige Messwerte es gibt

b) den Durchschnitt dieser Messwerte

c) wie viele Einträge verworfen werden mussten

Beachte: Der Code muss auch noch funktionieren, wenn die Liste verändert wird, wenn also Messdaten entfernt oder hinzugefügt werden.

Tipp: Mit `isinstance(w, (int, float))` kannst du in einem einzigen Schritt prüfen, ob `w` ein int *oder* ein float ist.

Füge einen Screenshot deines Codes ein:

Zusatzaufgaben (Optional)

Aufgabe A10

Überläufe der Weltgeschichte. Die folgenden drei Geschichten sind alle wahr. Rechne jeweils nach, ob die Zahlen stimmen.

a) **Boeing 787.** Die US-Luftfahrtbehörde verordnete 2015, dass jede Boeing 787 spätestens alle 248 Tage komplett ausgeschaltet werden muss – sonst fällt im Flug die gesamte Stromversorgung aus. Ursache war ein **signed** 32-Bit-Zähler, der *Hundertstelsekunden* zählt. Zeige durch Rechnung, dass dieser Zähler tatsächlich nach rund 248 Tagen überläuft.

b) **Das Jahr-2038-Problem.** Sehr viele Systeme speichern die Zeit als Anzahl Sekunden seit dem 1. Januar 1970, und zwar in einem signed 32-Bit-Integer. Berechne, wie viele Jahre dieser Zähler reicht, und bestimme damit das Jahr des Überlaufs. Was zeigt ein betroffenes Gerät unmittelbar danach als Datum an?

c) **Gangnam Style.** Ende 2014 knackte dieses Musikvideo als erstes die Grenze des YouTube-Zählers, der als signed 32-Bit-Integer angelegt war. Wie viele Aufrufe waren dazu nötig? YouTube stellte danach auf 64 Bit um – vergleiche diese neue Grenze mit der Weltbevölkerung von rund 8 Milliarden Menschen.

Aufgabe A11

Den richtigen Datentyp wählen. Gib für jede Anwendung an, welchen C#-Datentyp du wählen würdest. Nimm jeweils den **kleinstmöglichen**, der sicher ausreicht, und begründe deine Wahl mit einer kurzen Abschätzung. Zur Auswahl stehen `byte`, `sbyte` (s für signed), `short`, `ushort`, `int`, `uint` und `long`.

C#	Arduino (C++)	negativ?	Platz in C#	Platz auf Arduino
`sbyte`	`char`	ja	1 Byte (8 Bit)	1 Byte (8 Bit)
`byte`	`byte`	nein	1 Byte (8 Bit)	1 Byte (8 Bit)
`short`	`short`	ja	2 Bytes (16 Bit)	2 Bytes (16 Bit)
`ushort`	`unsigned short`	nein	2 Bytes (16 Bit)	2 Bytes (16 Bit)
`int`	`int`	ja	4 Bytes (32 Bit)	2 Bytes (16 Bit)
`uint`	`unsigned int`	nein	4 Bytes (32 Bit)	2 Bytes (16 Bit)
`long`	`long`	ja	8 Bytes (64 Bit)	4 Bytes (32 Bit)
`ulong`	`unsigned long`	nein	8 Bytes (64 Bit)	4 Bytes (32 Bit)

a) Alter eines Menschen in Jahren

b) Anzahl SuS der Kanti Romanshorn

c) Helligkeit eines Farbkanals (0–255)

d) Einwohnerzahl der Schweiz

e) Weltbevölkerung

f) Temperatur in Zehntelgrad, von $-50.0\,^{\circ}\mathrm{C}$ bis $+60.0\,^{\circ}\mathrm{C}$

g) Anzahl Sekunden seit dem 1. Januar 1970

h) Kontostand eines Grosskonzerns in Rappen

2 Exkurs: Programmieren in C#

In diesem Exkurs geht es nicht mehr um Datentypen an sich, sondern um die beiden wichtigsten Bausteine jedes Programms: **Bedingungen** und **Schleifen**. Beides kennst du aus Python bereits – in C# sieht es nur etwas anders aus.

Arbeite in dem Teil, der zu dir passt: Wer noch nie in C# programmiert hat, beginnt bei den Grundlagen. Wer C# schon einigermassen beherrscht, springt direkt zu den Aufgaben für Fortgeschrittene am Ende des Kapitels.

Grundlagen

Verzweigungen mit if-else if-else

Eine Bedingung sieht in Python bekanntlich so aus:

```python
note = 5

if note >= 4:
    print("genügend")
else:
    print("ungenügend")
```

Der entsprechende Code in C# ist:

```csharp
int note = 5;

if (note >= 4)
{
    Console.WriteLine("genügend");
}
else
{
    Console.WriteLine("ungenügend");
}
```

Im Vergleich zu Python fallen drei Dinge auf:

1. Die Bedingung steht in **runden Klammern**.
2. Nach `if (...)` steht **kein Doppelpunkt**.
3. Der zugehörige Block wird durch **geschweifte Klammern `{ }`** begrenzt statt durch *Einrückung*. Die Einrückung ist für C# *völlig egal* – man macht sie trotzdem, damit der Code besser lesbar ist.

Aufgabe B1

Schreibe folgenden C#-Code:

a) Speichere in einer Variablen eine Zahl: `int zahl = 42;` Der Code entscheidet dann, ob die Zahl grösser gleich 0 oder negativ ist. Entsprechend wird ausgegeben: 'positiv oder Null' resp. 'negativ'.

b) Nun wollen wir noch unterscheiden zwischen grösser als 0 und gleich Null – dafür müssen wir also drei Fälle unterscheiden. In Python würden wir das mit if-elif-else machen, in C# machen wir das mit **if-else if-else**.

Füge einen Screenshot deines Codes ein:

Operatoren

Um zwei Werte zu vergleichen oder Bedingungen miteinander zu verknüpfen, benötigen wir diverse Operatoren – wie bereits in Python:

Bedeutung	Operator Python	Operator C#	Beispiel C#
ist gleich	`==`	`==`	`if (a == 5)`
ist ungleich	`!=`	`!=`	`if (a != 5)`
kleiner (oder gleich)	`< <=`	`< <=`	`if (a <= 5)`
grösser (oder gleich)	`> >=`	`> >=`	`if (a >= 5)`
und (beide Bedingungen)	`and`	`&&`	`if (a > 0 && a < 10)`
oder (mindestens eine)	`or`	`||`	`if (a < 0 || a > 10)`
nicht (Umkehrung)	`not`	`!`	`if (!(a == 5))`
Rest der Division	`%`	`%`	`a % 2` ist 0, falls `a` gerade ist

Die eingefärbten Zeilen sind diejenigen, in denen sich Python und C# unterscheiden: Wo Python die ausgeschriebenen Wörter `and`, `or` und `not` verwendet, kennt C# nur Zeichenkombinationen. Alles Übrige ist identisch.

Aufgabe B2

Speichere in einer Variablen eine ganze Zahl, z.B. `int zahl = 132;`

Schreibe einen Code, der abhängig von der Zahl ausgibt:

a) ‘Die Zahl ist 42’ oder ‘Die Zahl ist nicht 42’.

b) ‘Die Zahl ist grösser als 67’ oder ‘Die Zahl ist kleiner gleich 67’.

c) ‘Die Zahl ist gerade’ oder ‘Die Zahl ist ungerade’.

Screenshot Code:

`for`-Schleifen

In Python schreibst du `for i in range(1, 6):`. In C# musst du dieselbe Information ausführlicher hinschreiben, dafür siehst du genau, was passiert:

```
for (int i = 1; i < 6; i++)
{
    Console.WriteLine(i);
}
```

In den runden Klammern stehen durch Semikolon getrennt drei Angaben:

1. `int i = 1` – der **Startwert**. Die Zählvariable wird hier gleich deklariert; sie existiert nur innerhalb der Schleife.
2. `i < 6` – die **Bedingung**. Solange sie erfüllt ist, läuft die Schleife weiter. Die Bedingung `i <= 5` wäre hier gleichbedeutend.
3. `i++` – die **Schrittweite**. `i++` ist die Kurzschreibweise für `i = i + 1`. Mit `i = i + 2` zählst du in Zweierschritten, mit `i--` rückwärts.

Aufgabe B3

Erste Schleifen.

a) Gib alle Zahlen von 1 bis 20 aus.

b) Gib nur die **geraden** Zahlen von 1 bis 20 aus. Finde *zwei* verschiedene Lösungswege: einen über die Schrittweite und einen mit einem `if` in der Schleife.

c) Gib die Zahlen von 10 bis 1 **rückwärts** aus.

d) Berechne die Summe $1 + 2 + 3 + \ldots + 100$ und gib sie aus. Kontrolliere dein Resultat mit der Formel von Gauss: $\frac{n \cdot (n+1)}{2}$.

Screenshot Code:

Aufgabe B4

Schleife und Bedingung kombinieren.

a) Gib alle Vielfachen von 7 zwischen 1 und 100 aus.

b) Zähle, wie viele Zahlen zwischen 1 und 1000 sowohl durch 3 *als auch* durch 5 teilbar sind. Gib am Schluss nur diese Anzahl aus.

c) Überlege dir *vor* dem Programmieren, wie das Resultat von b) auch ohne Computer bestimmt werden kann.

Screenshot Code:

Für Fortgeschrittene

Die folgenden Aufgaben sind für alle gedacht, die schon C# programmieren können.

Aufgabe B5

Wie schnell ist man am Ende? Schreibe ein Programm, das mit einer Variablen `int x = 1;` startet und diese in einer Schleife immer wieder **verdoppelt**. Gib bei jedem Durchgang die Nummer des Durchgangs und den aktuellen Wert aus, insgesamt etwa 35 Mal.

a) Bei welchem Durchgang wird der Wert zum ersten Mal negativ? Erkläre den Wert, der dort ausgegeben wird.

b) Was passiert in allen folgenden Durchgängen? Begründe das Ergebnis auf Bit-Ebene.

c) Ändere den Datentyp auf `long`. Bei welchem Durchgang tritt das Problem nun auf? Sage die Antwort *voraus*, bevor du das Programm laufen lässt.

Screenshot Code:

Aufgabe B6

Restwert-Algorithmus in C#. Schreibe ein Programm, das eine positive ganze Zahl (z.B. 42) mit dem **Restwert-Algorithmus** in ihre Binärdarstellung umwandelt und diese ausgibt. Verwende dazu *nicht* die fertige Funktion `Convert.ToString(zahl, 2)`, sondern implementiere den Algorithmus selbst.

Tipp: In C# kannst du mit `+` auch Strings aneinanderhängen, z.B. `bin = rest % 2 + bin;`

Prüfe dein Ergebnis anschliessend mit `Convert.ToString(zahl, 2)`.

Screenshot Code:

Aufgabe B7

Das Zweierkomplement sichtbar machen. Mit `Convert.ToString(x, 2).PadLeft(32, '0')` lässt sich das gespeicherte Bitmuster einer `int`-Variablen als 32-stellige Binärzahl ausgeben.

a) Lass dir die Bitmuster von 5, -5, `int.MaxValue` und `int.MinValue` ausgeben.

b) Wie kommt man vom Bitmuster von 5 zu jenem von -5? Formuliere eine Regel und überprüfe sie an einem weiteren Zahlenpaar, z.B. 12 und -12.

c) Erkläre mithilfe der Bitmuster, warum `int.MinValue` betragsmässig um 1 grösser ist als `int.MaxValue`.

Screenshot Code:

Aufgabe B8

Die Ariane 5 im Kleinen. Zum Schluss stellen wir den Fehler nach, mit dem dieses Dossier begonnen hat. Das Navigationssystem misst die horizontale Geschwindigkeit als `int` (in cm/s) und speichert sie anschliessend in einem `short` ab – so, wie es sich bei der Ariane 4 bewährt hatte.

Beachte: Für diese Aufgaben benötigen wir **Funktionen** und **Arrays**.

a) Lege ein Array mit den Messwerten `{ 12000, 25000, 32000, 32768, 40000, 120000 }` an. Gehe es mit einer Schleife durch, wandle jeden Wert mit `(short)` um und gib beide Werte nebeneinander aus. Ab welchem Messwert stimmt das Gespeicherte nicht mehr mit dem Gemessenen überein?

b) Aus 120 000 wird −11 072. Erkläre diesen Wert auf Bit-Ebene. *Tipp:* `Convert.ToString(120000, 2)` hilft weiter – und ein `short` hat nur 16 Bit Platz.

c) Baue nun die Rettung ein: Schreibe eine Funktion `bool PasstInShort(int wert)`, die *vor* der Umwandlung prüft, ob der Wert überhaupt hineinpasst. Passt er nicht, soll das Programm eine Warnung ausgeben statt eines falschen Werts. Verwende dazu `short.MinValue` und `short.MaxValue`.

d) **Zum Nachdenken.** Eine Alternative zu c) wäre, die Umwandlung mit `checked { ... }` zu umschliessen: Dann bricht das Programm bei einem Überlauf mit einer `OverflowException` ab – lieber laut abstürzen als leise falsch weiterrechnen. Bei der echten Ariane 5 wurde der Überlauf tatsächlich bemerkt und ein Fehler gemeldet – die Rakete stürzte trotzdem ab. Was hätte es also zusätzlich gebraucht?

Screenshot Code:

3 Gleitkommazahlen

Der Patriot-Fehler von Dhahran

Am 25. Februar 1991 schlug im Zweiten Golfkrieg eine irakische Scud-Rakete in eine Kaserne der US-Armee in Dhahran (Saudi-Arabien) ein. 28 Soldatinnen und Soldaten starben, rund 100 wurden verletzt. Dabei war die Kaserne durch eine Patriot-Batterie geschützt – ein Raketenabwehrsystem, das anfliegende Scuds am Himmel abfangen sollte. In dieser Nacht feuerte die Batterie keinen einzigen Schuss ab.

Die Untersuchung ergab: Die Patriot-Software zählte die Zeit in **Zehntelsekunden**. Um daraus Sekunden zu machen, multiplizierte sie den Zähler mit 0.1 – und genau hier lag das Problem. Die Zahl 0.1 lässt sich im Binärsystem **nicht exakt** darstellen, ähnlich wie sich $\frac{1}{3}$ im Dezimalsystem nicht exakt hinschreiben lässt ($0.333\ldots$). Der Computer rechnete deshalb mit einem winzigen Fehler von rund 0.000 000 095 Sekunden pro Zehntelsekunde – das ist rund ein Zehntausendstel eines Prozents. Völlig belanglos, sollte man meinen.

Die Batterie in Dhahran lief zu diesem Zeitpunkt allerdings bereits seit rund 100 Stunden ohne Neustart. In dieser Zeit hatte sich der winzige Fehler auf ein Drittel einer Sekunde aufsummiert. Eine Scud fliegt mit etwa 1676 m/s – in einer Drittelsekunde legt sie über einen halben Kilometer zurück. Das Radar suchte die anfliegende Rakete also an einer Stelle, an der sie längst nicht mehr war, fand dort nichts und stufte den Alarm als Fehlalarm ein. Zwei Wochen zuvor war das Problem bereits bekannt geworden; die korrigierte Software traf einen Tag nach dem Einschlag in Dhahran ein.

Beim Ariane-Fehler zu Beginn dieses Dossiers war eine Zahl *zu gross* für ihren Datentyp. Hier ist es subtiler: Die Zahlen sind alle brav im Wertebereich – sie sind bloss ein ganz kleines bisschen *falsch*. Genau darum geht es in diesem Kapitel.

Wie speichert man Zahlen mit Komma?

Bisher haben wir nur ganze Zahlen gespeichert. Für 3.14, -0.5 oder 6.022×10^{23} brauchen wir etwas Neues. Es gibt dafür zwei grundsätzlich verschiedene Ideen.

Idee 1: Festkomma

Man legt *fest*, wie viele Nachkommastellen es gibt, und speichert dann ganz normal eine ganze Zahl. Zwei Beispiele sind dir schon begegnet: Geldbeträge in **Rappen** statt in Franken, oder Temperaturen in **Zehntelgrad** statt in Grad. Aus 19.85 Franken wird die ganze Zahl 1985, aus $-12.3\,^\circ\text{C}$ wird -123. Das Komma existiert nur im Kopf der Programmiererin und wird erst bei der Ausgabe wieder eingefügt.

Festkomma ist **exakt** und schnell – aber unflexibel: Der Abstand zwischen zwei benachbarten darstellbaren Zahlen ist überall gleich gross. Mit Rappen kann man niemals genauer als auf 0.01 Franken rechnen, egal ob es um einen Kaugummi oder um das Bundesbudget geht. Für eine Physiksimulation, in der sowohl der Durchmesser eines Atoms ($10^{-10}\,\text{m}$) als auch die Distanz zur Sonne ($10^{11}\,\text{m}$) vorkommt, ist das völlig unbrauchbar.

Idee 2: Gleitkomma

Hier *gleitet* das Komma – daher der Name. Das Prinzip kennst du aus der Physik als **wissenschaftliche Schreibweise**:

$$299\,792\,458 = 2.99792458 \cdot 10^8 \qquad 0.000\,000\,000\,053 = 5.3 \cdot 10^{-11}$$

Vor dem Komma darf nur eine Ziffer stehen – und zwar *nicht* 0.

Gleichermassen schreibt man Binärzahlen in der wissenschaftlichen Schreibweise:

$$6.5 = 110.1_2 = \mathbf{1}.101_2 \cdot 2^2 \qquad 0.75 = 0.11_2 = \mathbf{1}.1_2 \cdot 2^{-1}$$

Da man im Binärsystem nur die beiden Ziffern 0 und 1 hat, muss vor dem Komma zwingend eine 1 stehen!

Um eine Zahl, die in der wissenschaftlichen Schreibweise ausgedrückt ist, zu speichern, muss man nicht die vielen Nullen speichern, sondern nur drei Dinge:

- Das **Vorzeichen**
- Die **Mantisse**, also die Ziffernfolge.
- Der **Exponent**, also die Angabe, wo das Komma hingehört.

Aufgabe C1

Wissenschaftliche Schreibweise von Dezimalzahlen:

Schreibe in wissenschaftlicher Schreibweise: $45\,000$, 0.00072, $-6\,371\,000$. Markiere dabei mit drei verschiedenen Farben: Vorzeichen, Mantisse, Exponent.

Aufgabe C2

Wissenschaftliche Schreibweise von Binärzahlen:

a) Wandle um in (binäre) wissenschaftliche Schreibweise. Markiere wieder: Vorzeichen, Mantisse, Exponent.

$$1011.01_2 \qquad 0.0011_2 \qquad -10\,1000_2$$

b) Wandle um ins Dezimalsystem:

$$1.101_2 \cdot 2^3 \qquad 1.0011_2 \cdot 2^{-2} \qquad -1.11_2 \cdot 2^4$$

Aufgabe C3

Gleitkommazahlen ins Binärsystem umwandeln. Wandeln wir als Beispiel 6.375 ins Binärsystem um. Dazu separieren wir den ganzzahligen Vorkommateil vom Nachkommateil: $6.375 = 6 + 0.375$

- **Vorkommateil:** Umwandeln mit Restwert-Algorithmus ($\rightarrow$ Grundlagenfach): Durch 2 dividieren, die Reste sind die Ziffern der Binärzahl:

$$\begin{aligned} 6 : 2 = 3 \text{ Rest: } 0 \\ 3 : 2 = 1 \text{ Rest: } 1 \\ 1 : 2 = 0 \text{ Rest: } 1 \\ \rightarrow 6 = 110_2 \end{aligned}$$

- **Nachkommateil:** Der Nachkommateil wird wiederholt **mit** 2 **multipliziert**. Die Ziffer, die dabei vor dem Komma erscheint, ist das nächste Bit; sie wird notiert und anschliessend gestrichen. Man hört auf, wenn der Rest 0 ist – oder wenn sich die Sache zu wiederholen beginnt.

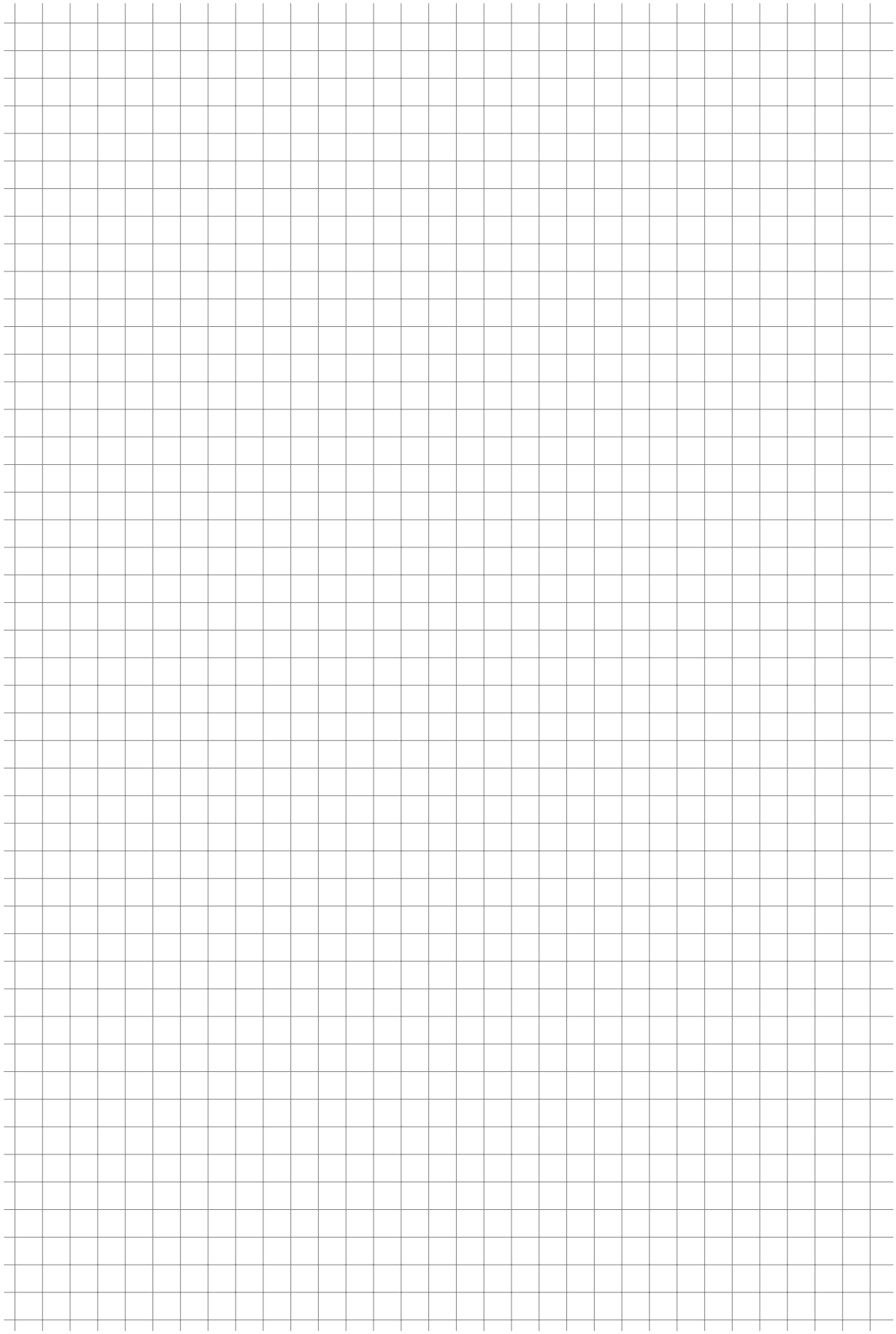
$$\begin{aligned} 0.375 \cdot 2 = 0.75 \text{ vor Komma: } 0 \\ 0.75 \cdot 2 = 1.5 \text{ vor Komma: } 1 \\ 0.5 \cdot 2 = 1.0 \text{ vor Komma: } 1 \\ \text{Rest ist } 0 \\ \rightarrow 0.375 = 0.011_2 \end{aligned}$$

Damit gilt also:

$$6.375 = 110.011_2 = 1.10011_2 \cdot 2^2$$

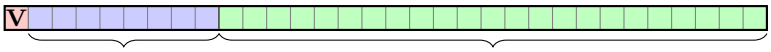
Wandle um ins Binärsystem und notiere dort in wissenschaftlicher Schreibweise:

a) 5.25
b) 0.625
c) 19.5
d) 0.1



Gleitkommazahlen in Programmierung

Seit 1985 sind Gleitkommazahlen im Standard **IEEE 754** weltweit einheitlich geregelt (gesprochen 'I triple E'). Jede Gleitkommazahl besteht aus drei Teilen, die hintereinander in den verfügbaren Bits abgelegt werden:



Exponent (8 Bit) Mantisse (23 Bit)

V = Vorzeichenbit (0 = positiv, 1 = negativ)

- Das erste Bit steht für das **Vorzeichen**: 0 positiv, 1 negativ
- Dann kommt eine fixe Anzahl Bits für den **Exponenten**. Dieser muss auch *negativ* sein können. Statt des Zweierkomplements verwendet man hier den **Bias** ('Verschiebung'): Gespeichert wird nicht der Exponent e selbst, sondern die Zahl $E = e + \text{Bias}$, die dadurch immer ≥ 0 ist. Bei 8 Exponentenbits ist der Bias $2^{8-1} - 1 = 127$; aus dem Exponenten $e = 2$ wird also der gespeicherte Wert $E = 129$. Der Vorteil: Zwei Gleitkommazahlen lassen sich so einfach Bit für Bit von links nach rechts vergleichen, wie Wörter im Lexikon.
- Zuletzt kommt eine fixe Anzahl Bits für die ***normalisierte*** **Mantisse**: Die Mantisse in binärer Schreibweise beginnt immer mit einer 1. Etwas, das immer gleich ist, muss man nicht speichern! Diese **implizite Eins** (auch 'hidden bit') wird weggelassen und beim Rechnen wieder ergänzt. Man bekommt so ein Bit Genauigkeit geschenkt.

Insgesamt gilt für eine normalisierte Gleitkommazahl:

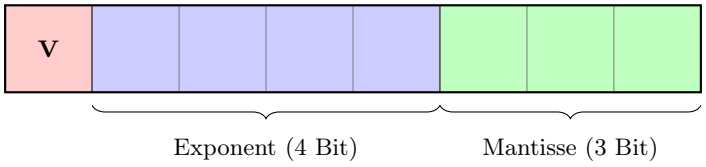
$$\text{Wert} = (-1)^{V} \cdot \underbrace{1.M}_{\text{Mantisse}} \cdot 2^{E-\text{Bias}}$$

Wie viele Bits genau für Exponent und Mantisse zur Verfügung stehen, hängt vom genauen Datentyp ab. Die beiden wichtigsten Formate sind:

	Total	**V**	**Exponent**	**Mantisse**	**Bias**
`float` (einfache Genauigkeit)	32 Bit	1	8 Bit	23 Bit	127
`double` (doppelte Genauigkeit)	64 Bit	1	11 Bit	52 Bit	1023

Das Mini-Float-Format: Gleitkommazahlen zum Anfassen

Mit 32 oder gar 64 Bit von Hand zu rechnen ist eher mühsam. Wir basteln uns deshalb ein Spielzeugformat, das **Mini-Float**, das nach exakt denselben Regeln funktioniert, aber nur 8 Bit braucht:



V = Vorzeichenbit (0 = positiv, 1 = negativ)

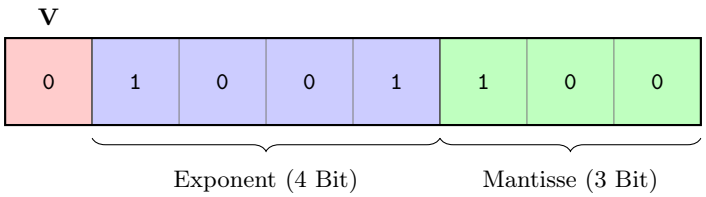
Für den Exponenten haben wir folgende Zahlen zur Verfügung: $0001_2 = 1$ bis und mit $1110_2 = 14$ – $0000_2 = 0$ und $1111_2 = 15$ sind für Spezialfälle reserviert. Wir haben also insgesamt $2^4 - 2 = 14$ Möglichkeiten. Wir verteilen diese fast fair zwischen den negativen und positiven Zahlen, und zwar von -6 bis und mit 7. Der Bias muss also $2^{4-1} - 1$ sein. Es gilt:

Tatsächlicher Exponent	Exponent (inkl. Bias)
7	$1110_2 = 14 = 7 + 7$
6	$1101_2 = 13 = 6 + 7$
1	$1000_2 = 8 = 1 + 7$
0	$0111_2 = 7 = 0 + 7$
-1	$0110_2 = 6 = -1 + 7$
-5	$0010_2 = 2 = -5 + 7$
-6	$0001_2 = 1 = -6 + 7$

Als **Beispiel** wollen wir die Zahl $6.375 = 1.10011_2 \cdot 2^2$ im Mini-Float-Format ausdrücken:

- **Vorzeichen:** Positiv, Vorzeichenbit also 0
- **Exponent mit Bias:** $2 + 7 = 9 = 1001_2$
- **Normalisierte Mantisse:** 110011 $\rightarrow$ ~~1~~100~~11~~. Beachte: Die beiden Bits rechts gehen verloren, da das Mini-Float-Format eine zu tiefe Genauigkeit besitzt.

Damit haben wir also:



V = Vorzeichenbit (0 = positiv, 1 = negativ)

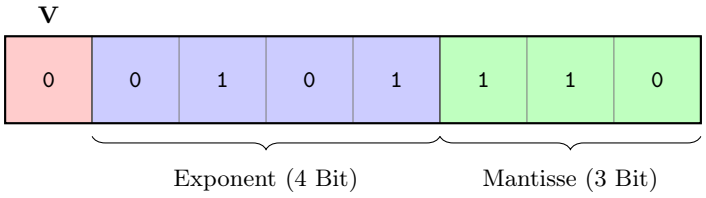
Aufgabe C4

Schreibe als Mini-Float:

a) -2.5
b) 0.875

Aufgabe C5

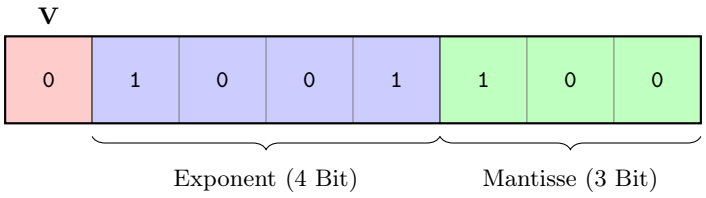
Welche Dezimalzahl steckt hinter der folgenden Zahl?



V = Vorzeichenbit (0 = positiv, 1 = negativ)

Aufgabe C6

Im Beispiel oben haben wir gesehen, dass die Zahl 6.375 im Mini-Float-Format wie folgt aussieht:



V = Vorzeichenbit (0 = positiv, 1 = negativ)

Wandle die Mini-Float Zahl um in eine Dezimalzahl. Was fällt dir auf? Erkläre!

Aufgabe C7

Betrachte folgenden Python-Code:

```
1 print(1 + 2 == 3)
2 print(0.1 + 0.2 == 0.3)
```

a) Welche Ausgabe erwartest du?

b) Führe den Code nun aus. Hast du richtig gelegen? Erkläre, was hier passiert!

Aufgabe C8

Wie weit reicht das Mini-Float? Erinnerung: Der gespeicherte Exponent E läuft von 1 bis 14, die Mantisse hat 3 Bit.

a) Bestimme die **grösste** darstellbare (normale) Zahl. Welches Bitmuster hat sie?

b) Bestimme die **kleinste positive** darstellbare Zahl.

c) Wie viele verschiedene *positive* Zahlen kann das Mini-Float darstellen? Vergleiche mit einem `byte`, das mit denselben 8 Bit $2^8 = 256$ Werte kennt. Was lernst du daraus?

Aufgabe C9

Wie gross sind die Lücken? Bei einem `int` ist der Abstand zwischen zwei benachbarten darstellbaren Zahlen immer 1. Bei Gleitkommazahlen ist das nicht so.

a) In der letzten Aufgabe hast du bereits die kleinste positive, sowie die grösste Zahl berechnet, die man mit Mini-Float darstellen kann. Bestimme nun die zweitkleinste positive, sowie die zweitgrösste Zahl.

b) Bestimme den Abstand zwischen der kleinsten und zweitkleinsten, sowie der grössten und zweitgrössten Zahl.

c) Was fällt dir auf?

Aufgabe C10

Welche Ausgabe erwartest du? Probiere den Code nachher aus und erkläre, was hier passiert!

```
1 print(1e16 + 1 == 1e16)
```

Probleme mit Gleitkommazahlen

Wir haben bisher einige Probleme angetroffen. Fassen wir sie zusammen!

Problem 1: Nicht genau darstellbare Zahlen

Im Dezimalsystem lassen sich genau jene Brüche exakt hinschreiben, deren Nenner nur die Primfaktoren 2 und 5 enthält – $\frac{1}{4} = 0.25$ geht, $\frac{1}{3} = 0.333\ldots$ nicht. Im Binärsystem ist die Bedingung strenger: Es geht nur, wenn der Nenner eine **Zweierpotenz** ist. Deshalb sind 0.5, 0.25 und 0.375 exakt darstellbar – 0.1, 0.2 und 0.3 aber *nicht*, denn deren Nenner enthält den Faktor 5:

$$0.1 = 0.0\overline{0011}_2 = 0.0001100110011001100\ldots_2$$

Der Computer muss diese unendliche Folge abbrechen und speichert daher einen gerundeten Wert. Daher gilt zum Beispiel:

```
0.1 + 0.2 == 0.3 ⟶ False
```

Merke: Vergleiche Gleitkommazahlen **nie** mit `==`. Prüfe stattdessen, ob die Differenz klein genug ist: `if (Math.Abs(a - b) < 1e-9)`

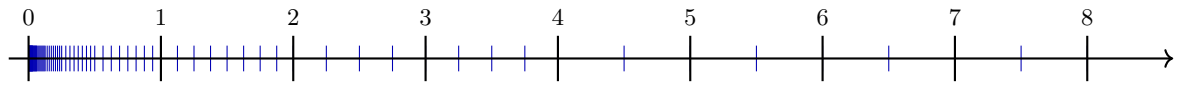
Problem 2: Fehler summieren sich auf

Ein einzelner solcher Rundungsfehler ist harmlos. Tut man aber in einer Schleife millionenfach dasselbe, addieren sich die Fehler – genau das ist der Patriot-Batterie zum Verhängnis geworden.

Merke: Verwende wenn möglich `int` oder `long`. Speichere die Zeit also nicht als `double` in Sekunden mit Nachkommastellen, sondern als `long` in Millisekunden.

Problem 3: Die Zahlen liegen ungleichmässig verteilt

Bei einem `int` ist der Abstand zwischen zwei benachbarten darstellbaren Zahlen immer 1. Bei Gleitkommazahlen ist er das nicht: Die Mantisse hat stets gleich viele Bits, aber der Exponent *streckt sie*. Zwischen 1 und 2 liegen also gleich viele darstellbare Zahlen wie zwischen 1024 und 2048 – die Lücken sind dort aber 1024 Mal grösser. Für unser Mini-Format sieht das so aus:



Alle darstellbaren Mini-Float-Zahlen von 0 bis 8. Weiter rechts geht es mit Schrittweite 1, dann 2, 4, … bis zur grössten Zahl 240.

Daraus folgt etwas Wichtiges: Mit n Bit kann man *immer* höchstens 2^n verschiedene Werte unterscheiden – auch mit Gleitkomma. Das Format erschafft keine zusätzlichen Zahlen, es **verteilt sie nur anders**: dicht dort, wo die Zahlen klein sind, und immer grosszügiger, je weiter man nach aussen kommt. Man tauscht Genauigkeit gegen Reichweite.

Problem 4: Absorption – kleine Summanden verschwinden

Ist eine Zahl sehr gross, so ist die Lücke zur nächsten darstellbaren Zahl grösser als der Summand, den man addieren will. Das Ergebnis wird auf den ursprünglichen Wert zurückgerundet, die Addition verpufft also spurlos. Bei einem `double` passiert das zum Beispiel hier:

```
>>> 1e16 + 1 == 1e16
True
```

Bei einem `float` ist der Effekt viel früher da: Schon ab $2^{24} = 16\,777\,216$ liegen die darstellbaren Zahlen zwei auseinander, ganze Zahlen darüber lassen sich also nicht mehr alle speichern.

Merke: Alle diese Probleme sind viel gravierender für `float` als für `double`. Verwende daher **im Zweifel** `double` statt `float`. Auf einem PC ist `double` kaum langsamer; `float` nimmt man dort, wo es wirklich auf Speicherplatz und Tempo ankommt.

Sonderfälle

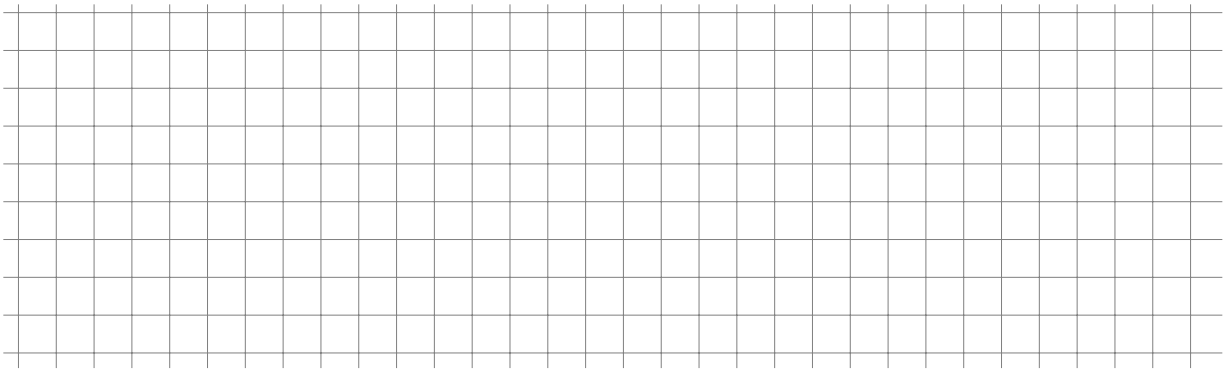
Mit der bisher betrachteten Form von Bitmustern lässt sich noch nicht alles darstellen – schon die Zahl 0 nicht, denn wegen der impliziten Eins ist die Mantisse ja *immer* mindestens 1. Deshalb sind zwei Exponenten-Bitmuster **reserviert**: dasjenige aus lauter Nullen und dasjenige aus lauter Einsen. Was die Zahl bedeutet, hängt dann zusätzlich davon ab, ob die Mantisse 0 ist oder nicht:

Exponent	Mantisse	Bedeutung	Erklärung
nur Nullen	$= 0$	± 0	Die Null. Es gibt sie zweimal: $+0$ und -0 (das Vorzeichenbit entscheidet). Beim Vergleich gelten sie trotzdem als gleich.
nur Nullen	$\neq 0$	subnormale Zahl	Füllt die Lücke zwischen 0 und der kleinsten normalen Zahl. Hier gilt *keine* implizite Eins, der Wert ist $0.M \cdot 2^{1-\text{Bias}}$.
gemischt	beliebig	normale Zahl	Der Normalfall, den wir oben behandelt haben: $\pm 1.M \cdot 2^{E-\text{Bias}}$
nur Einsen	$= 0$	$\pm\infty$	Entsteht bei Überlauf und bei einer Division wie `1.0 / 0.0`. In C# ausgegeben als ∞.
nur Einsen	$\neq 0$	`NaN`	'Not a Number', entsteht bei undefinierten Operationen wie $0/0$, $\sqrt{-1}$ oder $\infty - \infty$.

Zwei Dinge sind an den Sonderfällen bemerkenswert. Erstens löst nichts davon eine Fehlermeldung aus – das Programm rechnet mit ∞ und `NaN` einfach weiter. Zweitens ist `NaN` **ansteckend**: Jede Rechnung mit `NaN` ergibt wieder `NaN`. Ein einziger kaputter Messwert macht so eine ganze Auswertung unbrauchbar. Und `NaN` ist mit *nichts* gleich, nicht einmal mit sich selbst – `double.NaN == double.NaN` ergibt `False`. Prüfen muss man deshalb mit `double.IsNaN(x)`.

Aufgabe C11

Wir haben schon gesehen, dass die kleinste *normale* positive Zahl im Mini-Float-Format die Zahl 0.015625 (Bitmuster `0 0001 000`) ist. Verwendet man **subnormale Zahlen**, so geht es noch kleiner. Was ist also tatsächlich die kleinste mögliche positive Zahl?



Programmieraufgaben

Bemerkungen zum Programmieren mit Gleitkommazahlen:

- In C# schreibt man `double x = 0.1;` oder `float y = 0.1f;` – das `f` ist Pflicht, sonst reklamiert der Compiler mit `error CS0664`. Ohne Suffix ist jede Kommazahl in C# ein `double`.
- In **Python** gibt es nur einen einzigen Typ für Kommazahlen, und der heisst `float` – ist aber in Wahrheit ein 64-Bit-`double`. Was du in Python siehst, gilt also für C#-`double` genauso.
- Auf dem **Arduino Uno** ist `double` *dasselbe* wie `float`, nämlich nur 32 Bit. Wer dort doppelte Genauigkeit erwartet, bekommt sie nicht.
- Zwei Bitmuster des Exponenten sind reserviert: Mit $E = 0$ werden die Null und sehr kleine Zahlen dargestellt, mit lauter Einsen die Spezialwerte **Unendlich** und **NaN**.

Erinnerung `while`-Schleife. Bei einer `for`-Schleife weiss man vorher, wie viele Durchgänge es gibt. In diesem Kapitel wollen wir aber oft zählen, *bis* etwas eintritt – und wie lange das dauert, ist gerade die Frage. Dafür gibt es die `while`-Schleife: Sie läuft, solange ihre Bedingung erfüllt ist.

```
int n = 0;
while (n * n < 1000)      // laeuft, solange die Bedingung wahr ist
{
    n++;
}
Console.WriteLine(n);     // 32, denn 32*32 = 1024 > 1000
```

Alternative: Programmiere eine ‘Endlosschleife’ und breche dann mit `break` aus:

```
while (true)
{
    ...
    if (...){ break;}
}
```

Aufgabe C12

Das Sparschwein. Du legst jeden Tag 10 Rappen zur Seite. Ein Programm soll den Kontostand führen – einmal richtig und einmal falsch.

a) Schreibe eine Schleife über 1 000 000 Buchungen (natürlich nicht für eine einzelne Person), die parallel drei Variablen führt: einen `float` und einen `double`, zu denen jeweils 0.1 addiert wird, sowie einen `long`, zu dem 10 (Rappen) addiert wird. Gib am Schluss alle drei Kontostände in Franken aus.

b) Berechne für `float` und `double` die Abweichung vom exakten Betrag. Wie viele Franken fehlen jeweils?

c) Schreibe nun eine `while`-Schleife, die zählt, nach wie vielen Tagen der `float`-Kontostand zum ersten Mal um mehr als einen Rappen daneben liegt. Rechne das Ergebnis in Jahre um.

d) Welchen Datentyp nimmst du für eine Bank? Begründe mit deinen Zahlen.

Screenshot Code:

Aufgabe C13

Die Patriot-Batterie, zu Ende gerechnet. Zum Schluss rechnen wir die Katastrophe vom Kapitelanfang selbst nach. Die interne Uhr zählte **Zehntelsekunden** als ganze Zahl. Um daraus Sekunden zu machen, multiplizierte die Software mit einer gespeicherten Konstante – und die lautete wegen der begrenzten Bitzahl nicht 0.1, sondern

```
0.099999904632568
```

Schreibe ein Programm, das aus einer Anzahl **Betriebsstunden** die Anzahl Zehntelsekunden berechnet und daraus zwei Zeiten: die exakte (`ticks / 10.0`) und die fehlerhafte (`ticks * konstante`). Gib die Differenz in Sekunden aus und, mit der Geschwindigkeit multipliziert, in Metern. Teste mit 100 Stunden und vergleiche mit den Zahlen aus der Einleitung. Erinnerung: Eine Scud fliegt mit 1676 m/s.

Screenshot Code:

Aufgabe C14

Ultimativer IEEE 754 Umrechner. Schreibe mit Python oder C# ein Programm, dem man eine Dezimalzahl übergeben kann sowie die Anzahl Bits für Exponent und Mantisse. Der Code bestimmt dann die entsprechende Bitfolge.

Unter anderem muss man: Dezimalzahl in Binärzahl umwandeln, Bias bestimmen, ...

Screenshot:

Lösungen

Lösung Aufgabe A1

a) $1011_2 = 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 8 + 2 + 1 = 11$

b) $1001\,1010_2 = 2^7 + 2^4 + 2^3 + 2^1 = 128 + 16 + 8 + 2 = 154$

Lösung Aufgabe A2

In beiden Fällen können gleich viele Zahlen dargestellt werden, nämlich $2^4 = \mathbf{16}$: Jedes der 4 Bits hat zwei Möglichkeiten, also $2 \cdot 2 \cdot 2 \cdot 2$. Unterschiedlich ist nur, *welche* 16 Zahlen das sind.

a) `unsigned int`: kleinste Zahl **0**, grösste Zahl **15**. Das -1 bei $2^4 - 1 = 15$ kommt daher, dass man bei 0 zu zählen beginnt.

b) `int`: kleinste Zahl $-\mathbf{8}$, grösste Zahl **7**. Die 16 Möglichkeiten werden hälftig aufgeteilt: 8 negative Zahlen und 8 Zahlen ≥ 0. Weil die 0 zur positiven Hälfte gezählt wird, reicht diese nur bis 7.

Lösung Aufgabe A3

Anzahl Zahlen in beiden Fällen: $\mathbf{2^n}$

	kleinste Zahl	**grösste Zahl**
`unsigned int`	0	$2^n - 1$
`int`	-2^{n-1}	$2^{n-1} - 1$

Kontrolle mit $n = 4$: $2^4 = 16$ Zahlen, `unsigned int` von 0 bis $2^4 - 1 = 15$, `int` von $-2^3 = -8$ bis $2^3 - 1 = 7$. Stimmt mit der letzten Aufgabe überein.

Lösung Aufgabe A4

a) In C# belegt ein `int` 4 Bytes, also $n = 32$ Bit:

$$2^{32-1} - 1 = 2^{31} - 1 = 2\,147\,483\,647 \approx 2.1 \text{ Milliarden}$$

(In C# kann man sich diesen Wert mit `Console.WriteLine(int.MaxValue);` auch ausgeben lassen.)

b) Auf dem Arduino Uno belegt ein `int` nur 2 Bytes, also $n = 16$ Bit:

$$2^{16-1} - 1 = 2^{15} - 1 = 32\,767$$

Das ist rund 65 000 Mal weniger! Genau diese Zahl 32 767 war übrigens auch der Ariane 5 zum Verhängnis geworden.

Lösung Aufgabe A5

a) `unsigned long`, denn ein `unsigned int` ist auf dem Arduino nur 2 Bytes gross und damit viel zu klein (siehe b). Genau deshalb gibt `millis()` selbst ebenfalls einen `unsigned long` zurück.

b) `unsigned int` (16 Bit): grösster Wert $2^{16} - 1 = 65\,535$ ms, also nur gut **65 Sekunden**. `unsigned long` (32 Bit): grösster Wert $2^{32} - 1 = 4\,294\,967\,295$ ms. Umgerechnet:

$$\frac{4\,294\,967\,295}{1000 \cdot 60 \cdot 60 \cdot 24} \approx \mathbf{49.7\ Tage}$$

Auch das ist nicht unendlich: Ein Arduino, der länger als 49.7 Tage durchläuft, beginnt bei `millis()` wieder bei 0. Für ein Dauerprojekt (z.B. eine Uhr) muss man das im Code berücksichtigen.

c) Weil eine Laufzeit nie negativ sein kann. Verzichtet man auf die negative Hälfte, verdoppelt sich der nutzbare Bereich nach oben – mit einem `long` statt `unsigned long` käme man nur auf knapp 25 Tage.

Lösung Aufgabe A6

Dynamisch – Vorteile: schnell zu schreiben, wenig "Bürokratie"; ideal für kleine Skripte, Prototypen, Datenanalyse; dieselbe Funktion funktioniert für verschiedene Typen.

Dynamisch – Nachteile: Typfehler erst zur Laufzeit (evtl. nach Stunden oder beim Kunden); Editor kann weniger helfen; langsamer, weil der Typ ständig neu geprüft werden muss; höherer Speicherverbrauch.

Statisch – Vorteile: Fehler werden beim Kompilieren gefunden; deutlich schnellerer und speichersparender Maschinencode (der Compiler weiss genau, wie viele Bytes wo liegen); Autovervollständigung im Editor funktioniert gut.

Statisch – Nachteile: mehr Schreibarbeit; unflexibler; man muss sich *vorher* Gedanken über Wertebereiche machen.

Wahl: Python für Auswertungen, Prototypen, "mal schnell etwas ausprobieren". C#/C++ für grosse Programme im Team, für Software, die sehr zuverlässig sein muss (Medizin, Luftfahrt), und überall dort, wo Speicher und Rechenzeit knapp sind – also insbesondere auf einem Mikrocontroller wie dem Arduino. Und natürlich dort, wo die Risiken gross sind (finanziell, Leben).

Lösung Aufgabe A7

Zu b): Die Fehlermeldung lautet `error CS1002: ; expected`. Der Compiler weigert sich also bereits vor dem Ausführen – typisch für eine **statisch typisierte**, kompilierte Sprache. In Python würde man einen solchen Fehler erst bemerken, wenn die Zeile tatsächlich ausgeführt wird.

Tipp: Lies Fehlermeldungen immer genau – oft sagen sie dir ziemlich exakt, wo das Problem liegt und worin es besteht!

Zu c), d) und e):

```
int a = 7;
int b = 3;

Console.WriteLine(a + b);   // 10
Console.WriteLine(a - b);   // 4
Console.WriteLine(a * b);   // 21
Console.WriteLine(a / b);   // 2   <-- nicht 2.333!

Console.WriteLine("7 + 3 = " + (a + b));   // 7 + 3 = 10
```

Zu d): Der Taschenrechner sagt $7 : 3 = 2.\overline{3}$, C# gibt aber 2 aus. Der Grund: Sind *beide* Zahlen vom Typ `int`, so führt C# eine **Ganzzahldivision** durch – das Ergebnis ist wieder ein `int`, der Rest wird weggeworfen. Damit gebrochen gerechnet wird, muss mindestens eine der beiden Zahlen eine Gleitkommazahl sein, z.B. `Console.WriteLine(a / 3.0);` ergibt 2.3333333333333335.

Zu e): Die Klammern um `(a + b)` sind wichtig! Ohne sie rechnet C# von links nach rechts, hängt also zuerst die 7 und dann die 3 an den Text an – die Ausgabe wäre `7 + 3 = 73`.

Lösung Aufgabe A8

a) Zeile für Zeile: `int large;` deklariert eine Variable für ganze Zahlen, noch ohne ihr einen Wert zuzuweisen. `large = 2147483647;` weist ihr den grösstmöglichen `int`-Wert zu, die erste Ausgabe lautet also 2 147 483 647. Danach wird 1 dazugezählt – und dieses Resultat passt nicht mehr in 32 Bit. Erfahrungsgemäss erwartet man hier entweder eine Fehlermeldung, einen Programmabsturz oder die Ausgabe 2 147 483 648. Alle drei Vermutungen sind falsch.

b) Die tatsächliche Ausgabe lautet:

```
2147483647
-2147483648
```

Die Zahl kippt also von der grösstmöglichen auf die kleinstmögliche. Man nennt das einen **Überlauf** (englisch *overflow*). Auf Bit-Ebene ist gut zu sehen, warum:

$$\underbrace{0111\,1111\,\ldots\,1111\,1111_2}_{2\,147\,483\,647} + 1_2 = \underbrace{1000\,0000\,\ldots\,0000\,0000_2}_{-2\,147\,483\,648}$$

Durch den Übertrag springt das vorderste Bit von 0 auf 1 – und genau dieses Bit entscheidet im Zweierkomplement über das Vorzeichen. Aus der grössten positiven Zahl wird so die kleinste negative, ähnlich wie ein Kilometerzähler von 999 999 auf 000 000 springt.

c) Gefährlich ist vor allem, **dass gar nichts passiert**: kein Absturz, keine Fehlermeldung, nicht einmal eine Warnung. Das Programm rechnet einfach mit einer falschen Zahl weiter. Dazu kommt der Vorzeichenwechsel – aus einer sehr grossen positiven Zahl wird eine sehr grosse negative. Plötzlich ist eine zurückgelegte Distanz negativ, aus einem Guthaben wird eine Schuld oder eine Uhr läuft rückwärts. Weil der Fehler still geschieht, fällt er meist erst viel später auf, weit weg von seiner Ursache. Genau so war es bei der **Ariane 5**: Ein Wert passte nicht in eine 16-Bit-Ganzzahl, das Navigationssystem meldete daraufhin einen Fehler, und die Steuerung hielt diese Fehlermeldung für Flugdaten.

Nachtrag: In C# kann man sich davor schützen. Umschliesst man die Rechnung mit `checked { ... }`, so wird bei einem Überlauf eine `OverflowException` geworfen und das Programm bricht ab. Für sicherheitskritische Software ist das die bessere Variante: lieber laut abstürzen als leise falsch weiterrechnen.

Lösung Aufgabe A9

```python
messwerte = [12.5, 14, "Fehler", 13.2, None, 11, "n/a", 15.8]

def auswerten(werte):
    summe = 0
    counter = 0

    for mw in werte:
        if isinstance(mw,int) or isinstance(mw,float):
            summe += mw
            counter += 1

    print(f"Anzahl gültige Messwerte: {counter}")
    print(f"Mittelwert Messwerte:     {summe/counter}")
    print(f"Anzahl ungültige Werte:   {len(werte) - counter}")

auswerten(messwerte)
```

Die Ausgabe lautet: 5 gültige Messwerte, Mittelwert 13.3, 3 verworfene Einträge.

Wichtig für die Forderung, dass der Code auch bei veränderter Liste funktioniert: Innerhalb der Funktion wird ausschliesslich mit dem Parameter `werte` gearbeitet, nie mit `messwerte`. Und die Anzahl der verworfenen Einträge wird mit `len(werte) - counter` *berechnet* statt fest hingeschrieben.

Ohne die Prüfung mit `isinstance` würde bereits `summe += mw` beim Eintrag `"Fehler"` mit einem `TypeError` abstürzen, weil man einen String nicht zu einer Zahl addieren kann. Genau das ist die Kehrseite der dynamischen Typisierung: Python merkt den Fehler erst, wenn die betroffene Zeile tatsächlich ausgeführt wird – C# hätte sich schon vor dem Start geweigert.

Lösung Aufgabe A10

a) Der grösste Wert ist $2^{31} - 1 = 2\,147\,483\,647$ Hundertstelsekunden:

$$\frac{2\,147\,483\,647}{100 \cdot 60 \cdot 60 \cdot 24} = \frac{21\,474\,836.47}{86\,400} \approx \mathbf{248.6\ Tage}$$

Die Behörde schrieb 248 Tage vor, also mit etwas Sicherheitsmarge.

b)

$$\frac{2^{31} - 1}{60 \cdot 60 \cdot 24 \cdot 365.25} = \frac{2\,147\,483\,647}{31\,557\,600} \approx \mathbf{68.1\ Jahre}$$

Wegen $1970 + 68 = \mathbf{2038}$ passiert der Überlauf im Januar 2038 (genau am 19. Januar um 03:14:07 Uhr UTC). Eine Sekunde später springt der Zähler auf -2^{31} – und betroffene Geräte zeigen dann den 13. Dezember **1901** an, also 68 Jahre *vor* 1970.

c) Nötig waren $2^{31} - 1 = 2\,147\,483\,647$ Aufrufe, also rund 2.1 Milliarden. Mit 64 Bit liegt die Grenze bei $2^{63} - 1 \approx 9.2 \cdot 10^{18}$. Pro Mensch auf der Erde wären das über eine Milliarde Aufrufe – das Problem ist damit endgültig erledigt.

Lösung Aufgabe A11

a) `byte` $(0 \ldots 255)$ reicht bequem.

b) `ushort` – rund 600 SuS. Ein `byte` reicht **nicht**, weil es nur bis 255 zählt.

c) `byte` – genau dafür gemacht, $0 \ldots 255$ sind exakt 8 Bit.

d) `int` – rund 9 Millionen. In 16 Bit passen nur 65 535, also zu wenig; 32 Bit reichen locker.

e) `long` – die Falle! Rund $8 \cdot 10^9$ Menschen passen weder in `int` $(2.1 \cdot 10^9)$ noch in `uint` $(4.3 \cdot 10^9)$.

f) `short` – der Bereich ist $-500 \ldots 600$ in Zehntelgrad. Der Trick: Man versteckt die Nachkommastelle in der Einheit und braucht dadurch *keine* Gleitkommazahl. Genau so machen es viele Sensoren.

g) `int` würde bis $2^{31} - 1$ Sekunden reichen, also bis ins Jahr 2038 (siehe Aufgabe oben). Wer länger plant, nimmt `long`.

h) `long`. In Rappen statt in Franken zu rechnen, ist die Standardlösung, um Rundungsfehler zu vermeiden – warum, sehen wir im Kapitel über Gleitkommazahlen.

Lösung Aufgabe B1

```
int zahl = 42;

// a) zwei Faelle
if (zahl >= 0)
{
    Console.WriteLine("positiv oder Null");
}
else
{
    Console.WriteLine("negativ");
}

// b) drei Faelle
if (zahl > 0)
{
    Console.WriteLine("positiv");
}
else if (zahl == 0)
{
    Console.WriteLine("Null");
}
else
{
    Console.WriteLine("negativ");
}
```

Teste dein Programm unbedingt mit allen drei Sorten von Zahlen, also z.B. mit 42, mit -7 und mit 0 – sonst merkst du nicht, ob wirklich alle Fälle richtig behandelt werden.

Zwei Dinge sind wichtig: Erstens wird beim Vergleichen `==` geschrieben (doppeltes Gleichheitszeichen), denn ein einfaches `=` *weist zu* und vergleicht nicht. Zweitens wird `else if` nur dann überhaupt geprüft, wenn die Bedingung davor *falsch* war. Deshalb genügt in b) die Bedingung `zahl == 0`: Dass die Zahl nicht positiv ist, wurde ja bereits ausgeschlossen. Das abschliessende `else` braucht gar keine Bedingung mehr – es fängt schlicht alles auf, was übrig bleibt.

Lösung Aufgabe B2

```
int zahl = 132;

// a)
if (zahl == 42) {
    Console.WriteLine("Die Zahl ist 42");
}
else {
    Console.WriteLine("Die Zahl ist nicht 42");
}

// b)
if (zahl > 67) {
    Console.WriteLine("Die Zahl ist grösser als 67");
}
else {
    Console.WriteLine("Die Zahl ist kleiner gleich 67");
}

// c)
if (zahl % 2 == 0) {
    Console.WriteLine("Die Zahl ist gerade");
}
else {
    Console.WriteLine("Die Zahl ist ungerade");
}
```

Mit `zahl = 132` lautet die Ausgabe: *Die Zahl ist nicht 42*, *Die Zahl ist grösser als 67*, *Die Zahl ist gerade*.

Hier ist ein neuer Schreibstil verwendet worden: Besteht ein Block nur aus einer einzigen Anweisung, darf man die geschweiften Klammern auch *auf derselben Zeile* schreiben. Das ist reine Kosmetik – C# ist die Einrückung ja egal –, macht kurze Verzweigungen aber deutlich übersichtlicher.

Zu c): Der Trick mit `%` ist es wert, gemerkt zu werden. `zahl % 2` liefert den **Rest** der Division durch 2, und dieser Rest ist genau dann 0, wenn die Zahl gerade ist. Nach demselben Muster prüft man mit `zahl % 3 == 0` die Teilbarkeit durch 3 und so weiter.

Lösung Aufgabe B3

```
// a)
for (int i = 1; i <= 20; i++) {
    Console.WriteLine(i);
}

// b) Variante 1: Schrittweite 2
for (int i = 2; i <= 20; i = i + 2) {
    Console.WriteLine(i);
}

// b) Variante 2: alle durchgehen, ungerade ueberspringen
for (int i = 1; i <= 20; i++)
{
    if (i % 2 == 0) {
        Console.WriteLine(i);
    }
}

// c)
for (int i = 10; i >= 1; i--) {
    Console.WriteLine(i);
}

// d)
int summe = 0;
for (int i = 1; i <= 100; i++) {
    summe += i;
}
Console.WriteLine(summe);          // 5050
```

Kontrolle zu d): $\frac{100 \cdot 101}{2} = 5050$. Passt.

Variante 1 in b) ist effizienter, weil die Schleife nur halb so oft durchlaufen wird. Variante 2 ist dafür flexibler – so kann man beliebige Bedingungen prüfen, siehe nächste Aufgabe.

Lösung Aufgabe B4

```
// a)
for (int i = 1; i <= 100; i++)
{
    if (i % 7 == 0) {
        Console.WriteLine(i);
    }
}

// b)
int anzahl = 0;
for (int i = 1; i <= 1000; i++)
{
    if (i % 3 == 0 && i % 5 == 0) {
        anzahl++;
```

```
    }
}
Console.WriteLine(anzahl);          // 66
```

Zu c): Eine Zahl ist genau dann durch 3 *und* durch 5 teilbar, wenn sie durch 15 teilbar ist. Also $1000 : 15 = 66.\overline{6}$, und weil nur ganze Vielfache zählen, sind es **66** Zahlen.

Lösung Aufgabe B5

```
int x = 1;
for (int i = 0; i <= 35; i++)
{
    Console.WriteLine(i + ": " + x);
    x = x * 2;
}
```

a) Bei Durchgang 31. Dort steht $x = 2^{31}$, was nicht mehr in einen `int` passt. Ausgegeben wird $-2\,147\,483\,648$, also -2^{31}: Das einzige gesetzte Bit ist genau das Vorzeichenbit.

b) Ab Durchgang 32 wird nur noch **0** ausgegeben. Auf binärer Ebene ist das Verdoppeln einfach ein Verschieben der Eins um eine Stelle nach links:

$$\begin{aligned} 1 &= 1_2 \\ 2 &= 10_2 \\ 4 &= 100_2 \\ 8 &= 1000_2 \\ \ldots &= \ldots \end{aligned}$$

Weil nur noch das vorderste Bit gesetzt war, fällt dieses beim nächsten Verdoppeln aus dem 32-Bit-Fenster heraus – übrig bleiben lauter Nullen.

c) Ein `long` hat 64 Bit, das Problem tritt also erst bei Durchgang 63 auf. Man gewinnt durch die Verdoppelung der Bitzahl nicht doppelt so viele Durchgänge, sondern eben genau 32 zusätzliche – der Wertebereich wächst dafür um den Faktor $2^{32} \approx 4$ Milliarden.

Lösung Aufgabe B6

```
int zahl = 42;
string bin = "";

for (int rest = zahl; rest > 0; rest = rest / 2)
{
    bin = (rest % 2) + bin;   // neue Ziffer VORNE anhaengen
}

Console.WriteLine(zahl + " = " + bin);                      // 42 = 101010
Console.WriteLine(zahl + " = " + Convert.ToString(zahl, 2));// 42 = 101010
```

Wichtig ist, dass die neue Ziffer *vorne* angefügt wird – der Algorithmus liefert die Bits ja von hinten nach vorne. Zu beachten: Für `zahl = 0` gibt dieser Code einen leeren String zurück, dieser Sonderfall müsste zusätzlich abgefangen werden.

Lösung Aufgabe B7

```
Console.WriteLine(Convert.ToString(5, 2).PadLeft(32, '0'));
Console.WriteLine(Convert.ToString(-5, 2).PadLeft(32, '0'));
Console.WriteLine(Convert.ToString(int.MaxValue, 2).PadLeft(32, '0'));
Console.WriteLine(Convert.ToString(int.MinValue, 2).PadLeft(32, '0'));
```

Ausgabe:

```
00000000000000000000000000000101     5
11111111111111111111111111111011    -5
01111111111111111111111111111111     int.MaxValue
10000000000000000000000000000000     int.MinValue
```

Zu b): Regel des **Zweierkomplements** – Grüsse ans Grundlagenfach: Alle Bits umkehren ($0 \leftrightarrow 1$) und danach 1 addieren. Probe mit 5: umgekehrt ergibt …11111010, plus 1 ergibt …11111011 – genau das Bitmuster von -5. Dasselbe funktioniert mit 12 und -12.

Zu c): Das vorderste Bit ist das Vorzeichenbit. Bei den positiven Zahlen ist es 0, es bleiben also nur 31 Bit für den Wert – der grösste ist $2^{31} - 1$. Bei den negativen Zahlen ist es 1, und das Muster mit lauter Nullen dahinter ist *nicht* für -0 vergeben, denn eine negative Null gibt es nicht. Dieses Muster steht deshalb für -2^{31}. Auf der negativen Seite ist also ein Wert mehr Platz.

Lösung Aufgabe B8

```
int[] messwerte = { 12000, 25000, 32000, 32768, 40000, 120000 };

foreach (int v in messwerte)
{
    if (PasstInShort(v))
    {
        short gespeichert = (short)v;
        Console.WriteLine(v + " cm/s  ->  gespeichert: " + gespeichert);
    }
    else
    {
        Console.WriteLine(v + " cm/s  ->  WARNUNG: passt nicht in einen short!");
    }
}

static bool PasstInShort(int wert)
{
```

```
18      return wert >= short.MinValue && wert <= short.MaxValue;
19  }
```

Hinweis zum Tippen: Die Funktion `PasstInShort` darf direkt unterhalb der Schleife stehen – C# findet sie auch dann, wenn sie erst nach ihrem Aufruf notiert ist. Wichtig ist nur das Schlüsselwort `static`.

Zu a): Ohne die Prüfung (also mit `short gespeichert = (short)v;` für alle Werte) lautet die Ausgabe:

```
1  12000 ->   12000        32768 ->  -32768
2  25000 ->   25000        40000 ->  -25536
3  32000 ->   32000       120000 ->  -11072
```

Bis 32 000 geht alles gut, ab 32 768 – also ab 2^{15} – kippt es. Und das Entscheidende: Es gibt *keine* Fehlermeldung. Das Programm rechnet seelenruhig mit einer Geschwindigkeit von $-11\,072$ cm/s weiter, obwohl die Rakete in Wahrheit vorwärts fliegt.

Zu b): Es ist $120\,000 = 1\,1101\,0100\,1100\,0000_2$, also eine 17-stellige Binärzahl. In einen `short` passen aber nur die hintersten 16 Bit, die vorderste 1 (mit dem Stellenwert 2^{16}) fällt weg. Übrig bleibt $1101\,0100\,1100\,0000_2 = 54\,464$ – und weil davon nun das vorderste Bit das **Vorzeichenbit** ist, wird der Wert als negativ gelesen: $54\,464 - 65\,536 = -11\,072$. Dasselbe Prinzip wie beim Verdoppeln weiter oben, nur fällt hier nicht ein einzelnes Bit heraus, sondern alles, was jenseits des 16-Bit-Fensters liegt.

Zu d): Ein Fehler zu *melden* genügt nicht – jemand muss ihn auch sinnvoll *behandeln.* Bei der Ariane 5 gab es keinen Notfallplan für diesen Fall: Das Ersatzsystem lief mit derselben Software und scheiterte an derselben Zeile, und die Steuerung interpretierte die Fehlermeldung anschliessend als Flugdaten. Nötig gewesen wäre ein Datentyp, der zum tatsächlichen Wertebereich der neuen Rakete passt (hier: `int` statt `short`), eine Prüfung wie in c) – und ein Ersatzsystem, das den Fehler wirklich auffängt. Merke: Der richtige Datentyp ist die erste Verteidigungslinie, nicht die letzte.

Lösung Aufgabe C1

Mit Vorzeichen, Mantisse und Exponent:

$$45\,000 = 4.5 \cdot 10^{4} \qquad 0.00072 = 7.2 \cdot 10^{-4} \qquad -6\,371\,000 = -6.371 \cdot 10^{6}$$

Die letzte Zahl ist der Erdradius in Metern. Beachte: Das Vorzeichen zählt nicht als Ziffer – vor dem Komma steht bei allen drei Zahlen genau eine Ziffer, und diese ist nicht 0. Die ersten beiden Zahlen sind positiv; ihr Vorzeichen ist ein unsichtbares +, das man nicht hinschreibt. Genau dieses eine Zeichen wird der Computer später in *einem einzigen Bit* unterbringen.

Lösung Aufgabe C2

a) Das Komma wird jeweils so weit verschoben, bis genau eine 1 davor steht. Nach links verschieben ergibt einen positiven Exponenten, nach rechts einen negativen:

$$1011.01_2 = 1.01101_2 \cdot 2^3 \qquad 0.0011_2 = 1.1_2 \cdot 2^{-3} \qquad -10\,1000_2 = -1.01_2 \cdot 2^5$$

Der Exponent ist nichts anderes als die Anzahl Stellen, um die das Komma gewandert ist. Und anders als im Dezimalsystem muss man sich nicht fragen, *welche* Ziffer vor dem Komma landet – es ist immer eine 1.

b) Zuerst die Mantisse in eine Dezimalzahl umwandeln, dann mit der Zweierpotenz multiplizieren:

$$\begin{aligned} 1.101_2 \cdot 2^3 &= \left(1 + \tfrac{1}{2} + \tfrac{1}{8}\right) \cdot 8 = 1.625 \cdot 8 = \mathbf{13} \\ 1.0011_2 \cdot 2^{-2} &= \left(1 + \tfrac{1}{8} + \tfrac{1}{16}\right) : 4 = 1.1875 : 4 = \mathbf{0.296875} \\ -1.11_2 \cdot 2^4 &= -\left(1 + \tfrac{1}{2} + \tfrac{1}{4}\right) \cdot 16 = -1.75 \cdot 16 = \mathbf{-28} \end{aligned}$$

Lösung Aufgabe C3

a) $5 = 101_2,\ 0.25 = 0.01_2 \ \Rightarrow\ 5.25 = 101.01_2 = \mathbf{1.0101_2 \cdot 2^2}$

b) $0.625 \cdot 2 = \underline{1}.25,\ 0.25 \cdot 2 = \underline{0}.5,\ 0.5 \cdot 2 = \underline{1}.0 \ \Rightarrow\ 0.625 = 0.101_2 = \mathbf{1.01_2 \cdot 2^{-1}}$

c) $19 = 1\,0011_2,\ 0.5 = 0.1_2 \ \Rightarrow\ 19.5 = 1\,0011.1_2 = \mathbf{1.00111_2 \cdot 2^4}$

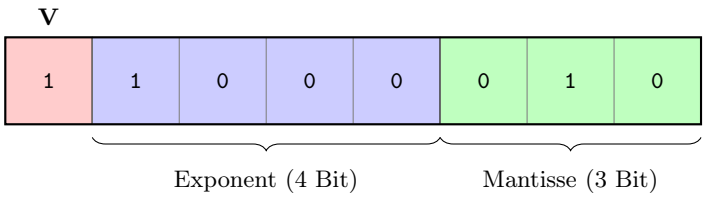
d) $0.1 \cdot 2 = \underline{0}.2,\ 0.2 \cdot 2 = \underline{0}.4,\ 0.4 \cdot 2 = \underline{0}.8,\ 0.8 \cdot 2 = \underline{1}.6,\ 0.6 \cdot 2 = \underline{1}.2,\ 0.2 \cdot 2 = \ldots$ – ab hier wiederholt sich alles:

$$0.1 = 0.0\overline{0011}_2 = \mathbf{1.\overline{1001}_2 \cdot 2^{-4}}$$

Bei d) bricht das Verfahren *nie* ab – die Binärdarstellung ist periodisch. Genau daran ist die Patriot-Batterie gescheitert.

Lösung Aufgabe C4

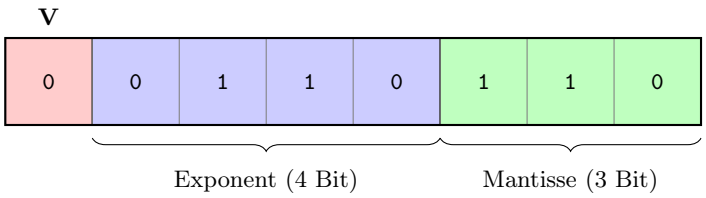
a) **Vorzeichen:** negativ, also Vorzeichenbit 1. **Betrag binär:** $2.5 = 10.1_2$. **Normalisieren:** $10.1_2 = 1.01_2 \cdot 2^1$, also $e = 1$. **Exponent mit Bias:** $E = 1 + 7 = 8 = 1000_2$. **Mantisse:** Die führende 1 wird gestrichen, es bleibt 01 – auf 3 Bit aufgefüllt also 010. Zusammengesetzt:



V = Vorzeichenbit (0 = positiv, 1 = negativ)

Hier geht nichts verloren: Die Mantisse braucht nur zwei Bits, das dritte ist eine angehängte 0. Die Zahl -2.5 ist im Mini-Float also **exakt** gespeichert.

b)



V = Vorzeichenbit (0 = positiv, 1 = negativ)

Lösung Aufgabe C5

Vorzeichen: Das erste Bit ist 0, die Zahl ist also positiv. **Exponent:** $0101_2 = 5$, und mit dem Bias abgezogen $e = 5 - 7 = -2$. **Mantisse:** Vor die gespeicherten Bits 110 kommt die implizite Eins und ein Komma: $1.110_2 = 1 + \frac{1}{2} + \frac{1}{4} = 1.75$. Damit ist

$$\text{Wert} = +1.75 \cdot 2^{-2} = \frac{1.75}{4} = \mathbf{0.4375}$$

Lösung Aufgabe C6

Zurückgerechnet ergibt sich: Vorzeichenbit 0, also positiv. Exponent $1001_2 = 9$, mit Bias abgezogen $e = 9 - 7 = 2$. Mantisse 100, mit der impliziten Eins davor $1.100_2 = 1.5$. Damit:

$$\text{Wert} = 1.5 \cdot 2^2 = \mathbf{6}$$

Das ist nicht die Zahl, mit der wir gestartet sind! Hineingesteckt haben wir 6.375, heraus kommt 6 – ein Fehler von 0.375 oder rund 6 %. Verantwortlich sind die beiden Mantissenbits, die beim Speichern abgeschnitten wurden: Sie hatten den Stellenwert $\frac{1}{16}$ und $\frac{1}{32}$ der Mantisse, was nach der Multiplikation mit 2^2 genau die fehlenden $0.25 + 0.125 = 0.375$ ergibt.

Der Grund dahinter ist grundsätzlicher Natur: Mit 3 Mantissenbits gibt es zwischen 4 und 8 überhaupt nur die acht Zahlen 4, 4.5, 5, 5.5, 6, 6.5, 7, 7.5. Die Zahl 6.375 ist schlicht nicht dabei – sie fällt in die Lücke zwischen 6 und 6.5. Der Computer *muss* sich also für einen Nachbarn entscheiden.

Und hier lohnt sich der genaue Blick: Wir haben oben einfach **abgeschnitten** und damit 6 erhalten. Der *nähere* Nachbar wäre aber 6.5 gewesen (0.125 statt 0.375 daneben). Echte Gleitkommazahlen nach IEEE 754 schneiden deshalb nicht ab, sondern **runden** – das Bitmuster wäre dort `0 1001 101`. Abschneiden ist nicht nur ungenauer, es ist auch *systematisch* ungenauer: Der Fehler geht immer in dieselbe Richtung (nach unten) und summiert sich deshalb auf, statt sich im Mittel auszugleichen.

Lösung Aufgabe C7

Erwartet hat man wohl zweimal `True`. Tatsächlich kommt aber `True` und `False` heraus.

Der Grund ist genau der aus der letzten Aufgabe: 0.1, 0.2 und 0.3 sind binär periodisch und werden deshalb alle drei *gerundet* gespeichert. Die gespeicherten Werte sind minim zu gross bzw. zu klein, und die Summe der ersten beiden trifft den dritten damit nicht exakt. Lässt man sich das Resultat direkt ausgeben, sieht man es:

```
1 print(0.1 + 0.2)     # 0.30000000000000004
```

Bei $1 + 2 = 3$ passiert das nicht, denn ganze Zahlen sind exakt darstellbar. Merke: Gleitkommazahlen **nie** mit `==` vergleichen, sondern prüfen, ob die Differenz klein genug ist:

```
1 print(abs((0.1 + 0.2) - 0.3) < 1e-9)     # True
```

Lösung Aufgabe C8

a) Grösstes E ist 14, also $e = 14 - 7 = 7$; grösste Mantisse ist $1.111_2 = 1.875$. Damit $1.875 \cdot 2^7 = \mathbf{240}$, Bitmuster `0 1110 111`.

b) Kleinstes E ist 1, also $e = 1 - 7 = -6$; kleinste Mantisse ist $1.000_2 = 1$. Damit $1 \cdot 2^{-6} = \frac{1}{64} = \mathbf{0.015625}$, Bitmuster `0 0001 000`.

c) 14 mögliche Exponenten × 8 mögliche Mantissen = **112** positive Zahlen (dazu kommen die 0 und 112 negative Zahlen, macht 225 verschiedene Werte). Ein `byte` kennt mit denselben 8 Bit 256 Werte – also sogar *mehr*, weil beim Mini-Float zwei Exponenten-Bitmuster für Spezialfälle reserviert sind. **Die Erkenntnis:** Gleitkomma zaubert keine zusätzlichen Zahlen herbei. Mit n Bit sind es immer höchstens 2^n Stück. Man verteilt sie bloss anders – statt von 0 bis 255 mit lauter gleichen Schritten von 1 nun von 0.015625 bis 240 mit immer grösseren Lücken. Man tauscht also *Genauigkeit gegen Reichweite*.

Lösung Aufgabe C9

a) Man muss nicht rechnen, sondern nur *weiterzählen*: Die nächste Zahl bekommt man, indem man das Bitmuster um 1 erhöht bzw. verringert – die Mantisse springt also zum nächsten Wert.

	Bitmuster	**Wert**
kleinste	`0 0001 000`	$1.000_2 \cdot 2^{-6} = \frac{1}{64} = 0.015625$
zweitkleinste	`0 0001 001`	$1.001_2 \cdot 2^{-6} = \frac{1.125}{64} = \mathbf{0.017578125}$
zweitgrösste	`0 1110 110`	$1.110_2 \cdot 2^{7} = 1.75 \cdot 128 = \mathbf{224}$
grösste	`0 1110 111`	$1.111_2 \cdot 2^{7} = 1.875 \cdot 128 = 240$

b) Unten: $0.017578125 - 0.015625 = \mathbf{0.001953125}$, also $\frac{1}{512} = 2^{-9}$.
Oben: $240 - 224 = \mathbf{16}$, also 2^4.

c) Die beiden Abstände sind völlig verschieden – der obere ist 8192 Mal grösser als der untere (und $8192 = 2^{13}$ ist genau der Unterschied der beiden Exponenten, von -6 bis 7). Bei einem `int` wäre der Abstand dagegen *überall* 1. Gleitkommazahlen liegen also **ungleichmässig** auf dem Zahlenstrahl: ganz dicht bei 0 und immer weiter auseinander, je grösser die Zahl wird. Der Grund: Die Mantisse hat immer gleich viele Bits, aber ihr kleinster Schritt wird mit 2^e multipliziert – der Abstand verdoppelt sich also bei jedem Schritt des Exponenten.
Rechnet man den Abstand *im Verhältnis* zur Zahl, sieht die Sache anders aus: unten $\frac{0.001953125}{0.015625} = 12.5\,\%$, oben $\frac{16}{240} \approx 6.7\,\%$. Über den ganzen Wertebereich bleibt dieser Anteil zwischen rund 6.7 % und 12.5 %. Merke: Die **absolute** Genauigkeit einer Gleitkommazahl schwankt enorm, die **relative** bleibt praktisch konstant. Genau deshalb eignet sich das Format für Grössen, die über viele Zehnerpotenzen gehen – und genau deshalb ist es für Geldbeträge ungeeignet, wo es auf den absoluten Rappen ankommt.
Ein konkretes Beispiel für die Folgen: Bei 128 beträgt der Abstand bereits 16. Die Zahl 130 lässt sich im Mini-Float also gar nicht darstellen, obwohl sie weit unterhalb der Obergrenze 240 liegt.

Lösung Aufgabe C10

Erwartet hätte man `False` – eine Zahl plus 1 ist ja nicht mehr dieselbe Zahl. Ausgegeben wird aber `True`.

Der Grund ist genau der Lückeneffekt aus der Aufgabe zuvor, nur beim echten `double`: Bei 10^{16} beträgt der Abstand zur nächsten darstellbaren Zahl bereits 2. Die Zahl $10^{16} + 1$ liegt genau dazwischen und wird auf 10^{16} zurückgerundet – die Addition **verpufft spurlos**. Man nennt das **Absorption**: Ein Summand, der kleiner ist als die Lücke, geht komplett verloren.

Lösung Aufgabe C11

Bitmuster: `0 0000 001`, also die Binärzahl:

$$0.001_2 \cdot 2^{1-7} = 2^{-3} \cdot 2^{-6} = 2^{-9} = 0.001953125$$

Lösung Aufgabe C12

```
float frankenFloat = 0.0f;
double frankenDouble = 0.0;
long rappen = 0;

for (int i=0;i < 1000000;i++){
    frankenFloat += 0.1f;
    frankenDouble += 0.1;
    rappen += 10;
}
Console.WriteLine("float:  " + frankenFloat);   // 100958.34
Console.WriteLine("double: " + frankenDouble);  // 100000.00000133288
Console.WriteLine("long:   " + rappen / 100.0); // 100000
```

Zu a) und b): Der exakte Betrag ist 100 000 Franken. Der `float` liegt bei 100 958.34, also um **958.34 Franken zu hoch** – fast ein Prozent Geld aus dem Nichts. Der `double` weicht dagegen nur um 0.0000013 Franken ab, und die Rappen-Variante ist **exakt**.

Zu c):

```
float frankenFloat = 0.0f;
long rappen = 0;
int tag = 0;

while (Math.Abs(frankenFloat - rappen / 100.0) < 0.01) {
    frankenFloat += 0.1f;
    rappen += 10;
    tag++;
}
Console.WriteLine("Tag: " + tag);        // -> 3149
```

Nach **3149** Tagen, also nach gut **8.6 Jahren**, ist der `float`-Kontostand um mehr als einen Rappen daneben. Klingt lange – aber eine Bank verbucht nicht eine Zahlung pro Tag, sondern Millionen. Dann ist man in Sekunden dort, wie a) zeigt.

Zu d): Ganzzahlig in **Rappen** (`long`). Das Resultat ist exakt, schnell und sparsam. `double` wäre für ein einzelnes Konto brauchbar, sammelt über Millionen von Buchungen aber ebenfalls Fehler an – und bei Geld ist "fast richtig" keine akzeptable Antwort. `float` ist schlicht unbrauchbar. In C# gibt es zusätzlich den Datentyp `decimal`, der intern zur Basis 10 rechnet und 0.1 deshalb exakt darstellt; er ist dafür langsamer und braucht 128 Bit.

Lösung Aufgabe C13

```
double konstante = 0.099999904632568;
long ticks = 100 * 3600 * 10;                   // 100 Betriebsstunden

double diff = ticks / 10.0 - ticks * konstante;
Console.WriteLine(diff);                        // 0.3433 s
```

```
6 Console.WriteLine(diff * 1676);                // 575.4 m
```

0.34 s bzw. **575 m** – genau die Zahlen aus der Einleitung.

Lösung Aufgabe C14

Keine Lösung.