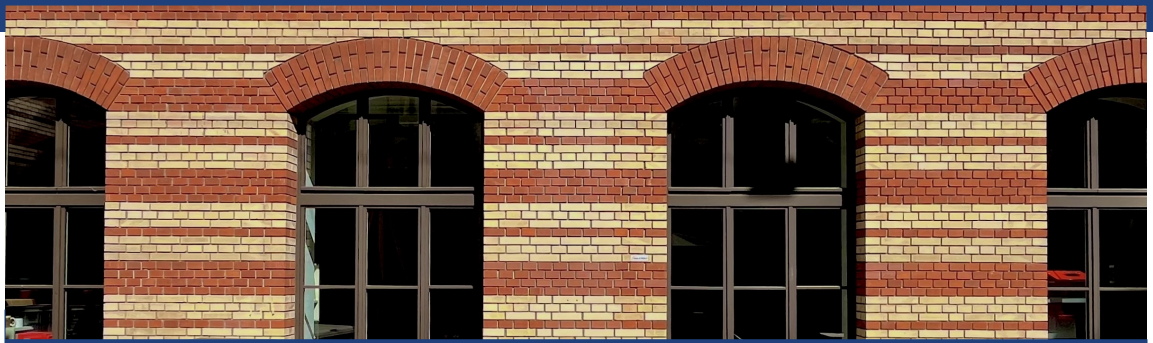


Informatik

Structs, Rekursion

Adel Gavranović

ETH Zürich — 2026

Überblick

Runden in F^*
F^* auf der Zahlengeraden
Structs
Rekursion
Extra: Rekursion in Iteration umschreiben
Extra: Wiederholung Fliesskommazahlen



adelgavranovic.ch

Webpage

Mail

1. Follow-up

Follow-up aus letzter Übungsstunde

Follow-up aus letzter Übungsstunde

1. Wie rundet man in F^*? Was wird zum Beispiel aus 1.00011 mit $p = 4$?

Follow-up aus letzter Übungsstunde

1. Wie rundet man in F^*? Was wird zum Beispiel aus 1.00011 mit $p = 4$?
2. Wie sind die Zahlen von F^* auf der Zahlengeraden verteilt?

2. Runden in F^*

Das Rezept

Das Rezept

1. **Normalisieren**: genau eine Ziffer $\neq 0$ vor dem Punkt

Das Rezept

1. **Normalisieren**: genau eine Ziffer $\neq 0$ vor dem Punkt
2. **Abschneiden**: nach p Ziffern, die Ziffer vor dem Punkt zählt mit

Das Rezept

1. **Normalisieren**: genau eine Ziffer $\neq 0$ vor dem Punkt
2. **Abschneiden**: nach p Ziffern, die Ziffer vor dem Punkt zählt mit
3. **Rest anschauen**: mit der Hälfte der letzten behaltenen Stelle vergleichen

Das Rezept

1. **Normalisieren**: genau eine Ziffer $\neq 0$ vor dem Punkt
2. **Abschneiden**: nach p Ziffern, die Ziffer vor dem Punkt zählt mit
3. **Rest anschauen**: mit der Hälfte der letzten behaltenen Stelle vergleichen
4. **Aufräumen**: Übertrag? Renormalisieren. Exponent noch im Bereich?

Das Rezept

1. **Normalisieren**: genau eine Ziffer $\neq 0$ vor dem Punkt
2. **Abschneiden**: nach p Ziffern, die Ziffer vor dem Punkt zählt mit
3. **Rest anschauen**: mit der Hälfte der letzten behaltenen Stelle vergleichen
4. **Aufräumen**: Übertrag? Renormalisieren. Exponent noch im Bereich?

Beispiel ($p = 4$): $1.0001\color{magenta}{1}$

Das Rezept

1. **Normalisieren**: genau eine Ziffer $\neq 0$ vor dem Punkt
2. **Abschneiden**: nach p Ziffern, die Ziffer vor dem Punkt zählt mit
3. **Rest anschauen**: mit der Hälfte der letzten behaltenen Stelle vergleichen
4. **Aufräumen**: Übertrag? Renormalisieren. Exponent noch im Bereich?

Beispiel ($p = 4$): 1.00011 , der Rest ist grösser als die Hälfte der letzten Stelle, also aufrunden: 1.001

Alle Fälle ($p = 4$)

Exakt	Rest im Vergleich zur Hälfte	Gerundet

Alle Fälle ($p = 4$)

Exakt	Rest im Vergleich zur Hälfte	Gerundet
1.001		

Alle Fälle ($p = 4$)

Exakt	Rest im Vergleich zur Hälfte	Gerundet
1.001	kein Rest	1.001

Alle Fälle ($p = 4$)

Exakt	Rest im Vergleich zur Hälfte	Gerundet
1.001	kein Rest	1.001
1.00001		

Alle Fälle ($p = 4$)

Exakt	Rest im Vergleich zur Hälfte	Gerundet
1.001	kein Rest	1.001
1.00001	kleiner: abrunden	1.000

Alle Fälle ($p = 4$)

Exakt	Rest im Vergleich zur Hälfte	Gerundet
1.001	kein Rest	1.001
1.00001	kleiner: abrunden	1.000
1.00011		

Alle Fälle ($p = 4$)

Exakt	Rest im Vergleich zur Hälfte	Gerundet
1.001	kein Rest	1.001
1.00001	kleiner: abrunden	1.000
1.00011	grösser: aufrunden	1.001

Alle Fälle ($p = 4$)

Exakt	Rest im Vergleich zur Hälfte	Gerundet
1.001	kein Rest	1.001
1.00001	kleiner: abrunden	1.000
1.00011	grösser: aufrunden	1.001
1.0011		

Alle Fälle ($p = 4$)

Exakt	Rest im Vergleich zur Hälfte	Gerundet
1.001	kein Rest	1.001
1.00001	kleiner: abrunden	1.000
1.00011	grösser: aufrunden	1.001
1.0011		

Bei Gleichstand gilt *round to even*: die letzte Ziffer soll gerade werden.

Alle Fälle ($p = 4$)

Exakt	Rest im Vergleich zur Hälfte	Gerundet
1.001	kein Rest	1.001
1.00001	kleiner: abrunden	1.000
1.00011	grösser: aufrunden	1.001
1.0011		

Bei Gleichstand gilt *round to even*: die letzte Ziffer soll gerade werden.

Alle Fälle ($p = 4$)

Exakt	Rest im Vergleich zur Hälfte	Gerundet
1.001	kein Rest	1.001
1.00001	kleiner: abrunden	1.000
1.00011	grösser: aufrunden	1.001
1.0011	genau gleich, letzte Ziffer ungerade: auf	1.010

Bei Gleichstand gilt *round to even*: die letzte Ziffer soll gerade werden.

Alle Fälle ($p = 4$)

Exakt	Rest im Vergleich zur Hälfte	Gerundet
1.001	kein Rest	1.001
1.00001	kleiner: abrunden	1.000
1.00011	grösser: aufrunden	1.001
1.0011	genau gleich, letzte Ziffer ungerade: auf	1.010
1.0001		

Bei Gleichstand gilt *round to even*: die letzte Ziffer soll gerade werden.

Alle Fälle ($p = 4$)

Exakt	Rest im Vergleich zur Hälfte	Gerundet
1.001	kein Rest	1.001
1.00001	kleiner: abrunden	1.000
1.00011	grösser: aufrunden	1.001
1.0011	genau gleich, letzte Ziffer ungerade: auf	1.010
1.0001	genau gleich, letzte Ziffer gerade: ab	1.000

Bei Gleichstand gilt *round to even*: die letzte Ziffer soll gerade werden.

Alle Fälle ($p = 4$)

Exakt	Rest im Vergleich zur Hälfte	Gerundet
1.001	kein Rest	1.001
1.00001	kleiner: abrunden	1.000
1.00011	grösser: aufrunden	1.001
1.0011	genau gleich, letzte Ziffer ungerade: auf	1.010
1.0001	genau gleich, letzte Ziffer gerade: ab	1.000

Bei Gleichstand gilt *round to even*: die letzte Ziffer soll gerade werden.

Binär kurz: erstes abgeschnittenes Bit 0: ab. Bit 1 und danach noch eine 1: auf. Bit 1 und sonst nur Nullen: Gleichstand. Letzte Woche: $1.010011 \cdot 2^3 \approx 1.01010 \cdot 2^3$ war genau so ein Gleichstand.

Spezialfälle

Spezialfälle

- **Übertrag** ($p = 4$): $1.1111 \cdot 2^0 \approx 10.000 \cdot 2^0$

Spezialfälle

- **Übertrag** ($p = 4$): $1.1111 \cdot 2^0 \approx 10.000 \cdot 2^0 = 1.000 \cdot 2^1$ (renormalisieren)

Spezialfälle

- **Übertrag** ($p = 4$): $1.1111 \cdot 2^0 \approx 10.000 \cdot 2^0 = 1.000 \cdot 2^1$ (renormalisieren)
- **Zu gross** in $F^*(2, 4, -2, 2)$: $1.1111 \cdot 2^2 \approx 1.000 \cdot 2^3$

Spezialfälle

- **Übertrag** ($p = 4$): $1.1111 \cdot 2^0 \approx 10.000 \cdot 2^0 = 1.000 \cdot 2^1$ (renormalisieren)
- **Zu gross** in $F^*(2, 4, -2, 2)$: $1.1111 \cdot 2^2 \approx 1.000 \cdot 2^3$, aber $3 > e_{\max}$: nicht mehr in F^* (Overflow)

Spezialfälle

- **Übertrag** ($p = 4$): $1.1111 \cdot 2^0 \approx 10.000 \cdot 2^0 = 1.000 \cdot 2^1$ (renormalisieren)
- **Zu gross** in $F^*(2, 4, -2, 2)$: $1.1111 \cdot 2^2 \approx 1.000 \cdot 2^3$, aber $3 > e_{\max}$: nicht mehr in F^* (Overflow)
- **Zu klein**: $1.000 \cdot 2^{-3}$ ist kleiner als die kleinste positive Zahl $1.000 \cdot 2^{-2}$ (Underflow)

Spezialfälle

- **Übertrag** ($p = 4$): $1.1111 \cdot 2^0 \approx 10.000 \cdot 2^0 = 1.000 \cdot 2^1$ (renormalisieren)
- **Zu gross** in $F^*(2, 4, -2, 2)$: $1.1111 \cdot 2^2 \approx 1.000 \cdot 2^3$, aber $3 > e_{\max}$: nicht mehr in F^* (Overflow)
- **Zu klein**: $1.000 \cdot 2^{-3}$ ist kleiner als die kleinste positive Zahl $1.000 \cdot 2^{-2}$ (Underflow)
- **Negative Zahlen**: den Betrag runden, das Vorzeichen bleibt

Und eine Dezimalzahl?

0.1 **in** $F^*(2, 4, -4, 4)$

Und eine Dezimalzahl?

0.1 **in** $F^*(2, 4, -4, 4)$

1. Binär: $0.1_{10} = 0.0\overline{00011}_2$

Und eine Dezimalzahl?

0.1 **in** $F^*(2, 4, -4, 4)$

1. Binär: $0.1_{10} = 0.0\overline{0011}_2$
2. Normalisieren: $1.\overline{1001} \cdot 2^{-4}$

Und eine Dezimalzahl?

0.1 **in** $F^*(2, 4, -4, 4)$

1. Binär: $0.1_{10} = 0.0\overline{00011}_2$
2. Normalisieren: $1.\overline{1001} \cdot 2^{-4}$
3. Abschneiden: $1.100\mathbf{11001}\ldots \cdot 2^{-4}$

Und eine Dezimalzahl?

0.1 **in** $F^*(2, 4, -4, 4)$

1. Binär: $0.1_{10} = 0.0\overline{0011}_2$
2. Normalisieren: $1.\overline{1001} \cdot 2^{-4}$
3. Abschneiden: $1.10011001\ldots \cdot 2^{-4}$
4. Rest grösser als die Hälfte: $1.101 \cdot 2^{-4} = 0.1015625 \neq 0.1$

Und eine Dezimalzahl?

0.1 **in** $F^*(2, 4, -4, 4)$

1. Binär: $0.1_{10} = 0.0\overline{0011}_2$
2. Normalisieren: $1.\overline{1001} \cdot 2^{-4}$
3. Abschneiden: $1.10011001\ldots \cdot 2^{-4}$
4. Rest grösser als die Hälfte: $1.101 \cdot 2^{-4} = 0.1015625 \neq 0.1$

Andere Basis, gleiche Regeln: $\beta = 10$, $p = 3$

$$1.234 \approx 1.23 \qquad 1.236 \approx 1.24 \qquad 1.235 \approx 1.24 \qquad 1.225 \approx 1.22$$

Fragen/Unklarheiten?

3. F^* auf der Zahlengeraden

$F^*(2, 3, -2, 2)$ auf der Zahlengeraden



e
Zahlen
Abstand

Nur die positive Hälfte, die negative ist spiegelbildlich. Die 0 selbst liegt nicht in F^*.

$F^*(2, 3, -2, 2)$ auf der Zahlengeraden



e	-2
Zahlen	4
Abstand	$\frac{1}{16}$

Nur die positive Hälfte, die negative ist spiegelbildlich. Die 0 selbst liegt nicht in F^*.

$F^*(2, 3, -2, 2)$ auf der Zahlengeraden



e	-2	-1
Zahlen	4	4
Abstand	$\frac{1}{16}$	$\frac{1}{8}$

Nur die positive Hälfte, die negative ist spiegelbildlich. Die 0 selbst liegt nicht in F^*.

$F^*(2, 3, -2, 2)$ auf der Zahlengeraden



e	-2	-1	0
Zahlen	4	4	4
Abstand	$\frac{1}{16}$	$\frac{1}{8}$	$\frac{1}{4}$

Nur die positive Hälfte, die negative ist spiegelbildlich. Die 0 selbst liegt nicht in F^*.

$F^*(2, 3, -2, 2)$ auf der Zahlengeraden



e	-2	-1	0	1
Zahlen	4	4	4	4
Abstand	$\frac{1}{16}$	$\frac{1}{8}$	$\frac{1}{4}$	$\frac{1}{2}$

Nur die positive Hälfte, die negative ist spiegelbildlich. Die 0 selbst liegt nicht in F^*.

$F^*(2, 3, -2, 2)$ auf der Zahlengeraden



e	-2	-1	0	1	2
Zahlen	4	4	4	4	4
Abstand	$\frac{1}{16}$	$\frac{1}{8}$	$\frac{1}{4}$	$\frac{1}{2}$	1

Nur die positive Hälfte, die negative ist spiegelbildlich. Die 0 selbst liegt nicht in F^*.

Was sieht man?

Was sieht man?

- Jeder Exponent liefert gleich viele, gleichmässig verteilte Zahlen: $(\beta - 1) \cdot \beta^{p-1}$

Was sieht man?

- Jeder Exponent liefert gleich viele, gleichmässig verteilte Zahlen: $(\beta - 1) \cdot \beta^{p-1}$
- Von Exponent zu Exponent wird der Abstand β-mal grösser: β^{e-p+1}

Was sieht man?

- Jeder Exponent liefert gleich viele, gleichmässig verteilte Zahlen: $(\beta - 1) \cdot \beta^{p-1}$
- Von Exponent zu Exponent wird der Abstand β-mal grösser: β^{e-p+1}
- Bei der 0 ist es dicht, weit weg dünn

Was sieht man?

- Jeder Exponent liefert gleich viele, gleichmässig verteilte Zahlen: $(\beta - 1) \cdot \beta^{p-1}$
- Von Exponent zu Exponent wird der Abstand β-mal grösser: β^{e-p+1}
- Bei der 0 ist es dicht, weit weg dünn
- Um die 0 herum gibt es eine Lücke, oberhalb von 7 gar nichts mehr

Was sieht man?

- Jeder Exponent liefert gleich viele, gleichmässig verteilte Zahlen: $(\beta - 1) \cdot \beta^{p-1}$
- Von Exponent zu Exponent wird der Abstand β-mal grösser: β^{e-p+1}
- Bei der 0 ist es dicht, weit weg dünn
- Um die 0 herum gibt es eine Lücke, oberhalb von 7 gar nichts mehr
- Darum Guideline 2: keine Zahlen von extrem verschiedener Grösse addieren. Bei $p = 3$ ist der Abstand um 10 schon 2, also verschwindet die 0.5 in $10 + 0.5$ beim Runden

Selber ausprobieren

Fragen/Unklarheiten?

4. Feedback zu **code** expert

Allgemeines bezüglich **code** expert

Allgemeines bezüglich **code** expert

- Woche 9, “Set Product”, “All permutations” und “Dungeon”: nur mit Rekursion. Wer die Aufgabe ohne Rekursion trivialisiert, behält die XP, bekommt aber keine TA-Punkte

Allgemeines bezüglich **code** expert

- Woche 9, “Set Product”, “All permutations” und “Dungeon”: nur mit Rekursion. Wer die Aufgabe ohne Rekursion trivialisiert, behält die XP, bekommt aber keine TA-Punkte
- An der Prüfung genauso: “mit Rekursion” lösen heisst mit Rekursion, sonst gibt es nicht die volle Punktzahl

Allgemeines bezüglich **code** expert

- Woche 9, “Set Product”, “All permutations” und “Dungeon”: nur mit Rekursion. Wer die Aufgabe ohne Rekursion trivialisiert, behält die XP, bekommt aber keine TA-Punkte
- An der Prüfung genauso: “mit Rekursion” lösen heisst mit Rekursion, sonst gibt es nicht die volle Punktzahl
- Abgabe Woche 9: Montag, 16. November, 18:00

Allgemeines bezüglich **code** expert

- Woche 9, “Set Product”, “All permutations” und “Dungeon”: nur mit Rekursion. Wer die Aufgabe ohne Rekursion trivialisiert, behält die XP, bekommt aber keine TA-Punkte
- An der Prüfung genauso: “mit Rekursion” lösen heisst mit Rekursion, sonst gibt es nicht die volle Punktzahl
- Abgabe Woche 9: Montag, 16. November, 18:00
- Bei einer Aufgabe steckengeblieben? Kommt ins Study Center (jeden Mittwoch, Infos auf der Study-Center-Webseite)! Study Center

Fragen zu den Übungen der letzten Woche?

5. Lernziele

Ziele

Ziele

- ☐ Fortgeschrittenere Rekursionsaufgaben lösen können
- ☐ C++-Structs definieren und verwenden können

6. Zusammenfassung

Getting on the same page

Getting on the same page

- Was wurde diese Woche behandelt?

7. Structs

Ein eigener Datentyp: `struct`

```
struct strange {
  int n;
  bool b;
  std::vector<int> a = std::vector<int>(0);
};

int main() {
  strange x = {1, true, {1,2,3}};
  strange y = x;  // all members are copied
  std::cout << y.n << "  " << y.a[2] << "\n";
  return 0;
}
```

Ein eigener Datentyp: `struct`

```
struct strange {
  int n;
  bool b;
  std::vector<int> a = std::vector<int>(0);
};

int main() {
  strange x = {1, true, {1,2,3}};
  strange y = x;  // all members are copied
  std::cout << y.n << "  " << y.a[2] << "\n";
  return 0;
}
```

- Member dürfen beliebige Typen haben, auch Vektoren

Ein eigener Datentyp: `struct`

```
struct strange {
  int n;
  bool b;
  std::vector<int> a = std::vector<int>(0);
};

int main() {
  strange x = {1, true, {1,2,3}};
  strange y = x;  // all members are copied
  std::cout << y.n << "  " << y.a[2] << "\n";
  return 0;
}
```

- Member dürfen beliebige Typen haben, auch Vektoren
- Nach der schliessenden Klammer kommt ein ;

Ein eigener Datentyp: `struct`

```
struct strange {
  int n;
  bool b;
  std::vector<int> a = std::vector<int>(0);
};

int main() {
  strange x = {1, true, {1,2,3}};
  strange y = x;  // all members are copied
  std::cout << y.n << "  " << y.a[2] << "\n";
  return 0;
}
```

- Member dürfen beliebige Typen haben, auch Vektoren
- Nach der schliessenden Klammer kommt ein `;`
- Ausgabe: `1  3`

Ein eigener Datentyp: `struct`

```
struct strange {
  int n;
  bool b;
  std::vector<int> a = std::vector<int>(0);
};

int main() {
  strange x = {1, true, {1,2,3}};
  strange y = x;  // all members are copied
  std::cout << y.n << "  " << y.a[2] << "\n";
  return 0;
}
```

- Member dürfen beliebige Typen haben, auch Vektoren
- Nach der schliessenden Klammer kommt ein `;`
- Ausgabe: `1  3`
- `y` ist eine echte Kopie, auch der Vektor

Was gibt es gratis?

```
struct vec { double x; double y; double z; };

vec a = {1.0, 2.0, 3.0};  // initialize the members in order
vec b = a;                // copy, member by member
vec c;                    // reading c.x now is undefined behaviour

a == b;                   // error: no operator== for vec
a + b;                    // error: no operator+ for vec
std::cout << a;           // error: no operator<< for vec
```

Was gibt es gratis?

```
struct vec { double x; double y; double z; };

vec a = {1.0, 2.0, 3.0};  // initialize the members in order
vec b = a;                // copy, member by member
vec c;                    // reading c.x now is undefined behaviour

a == b;                   // error: no operator== for vec
a + b;                    // error: no operator+ for vec
std::cout << a;           // error: no operator<< for vec
```

Nur = entsteht automatisch. Alles andere überladen wir selbst.

Aufgabe “Geometry Exercise”

Öffnet “Geometry Exercise”
auf **code** expert

```
struct vec {
  double x;
  double y;
  double z;
};
```

Aufgabe “Geometry Exercise”

Öffnet “Geometry Exercise” auf **code** expert

```
struct vec {
  double x;
  double y;
  double z;
};
```

1. Summe von zwei Vektoren
2. `struct line` für Geraden in 3D
3. Gerade um einen Vektor verschieben
4. `operator+` für Vektoren
5. `operator+` für Gerade plus Vektor
6. (optional) `operator<<` für beide

Alle zusammen, Schritt für Schritt, 30 Minuten.

Teil 1: Summe

Aufgabe

Implementiert eine Funktion, die die Summe von zwei Vektoren berechnet.

Teil 1: Summe

Aufgabe

Implementiert eine Funktion, die die Summe von zwei Vektoren berechnet.

```
// POST: returns the sum of a and b
vec sum(const vec& a, const vec& b) {
  return {a.x + b.x, a.y + b.y, a.z + b.z};
}
```

```
vec sum(const vec& a, const vec& b) {
  vec tmp;
  tmp.x = a.x + b.x;
  tmp.y = a.y + b.y;
  tmp.z = a.z + b.z;
  return tmp;
}
```

Beide sind korrekt. Wir nehmen ab jetzt die kurze Version.

Teil 2: Gerade

Aufgabe

Implementiert ein `struct line` für Geraden in 3D. Jedes Objekt muss genau eine Gerade beschreiben. Eine Gerade darf aber mehrere Darstellungen haben.

Wie würdet ihr eine Gerade darstellen?

Zu zweit, 1 Minute.

Teil 2: Zwei Möglichkeiten

Zwei Punkte

```
struct line {
  vec point1;
  vec point2;    // INV: point1 != point2
};
```

Punkt und Richtung

```
struct line {
  vec point;
  vec dir;       // INV: dir != (0,0,0)
};
```

Teil 2: Zwei Möglichkeiten

Zwei Punkte

```
struct line {
  vec point1;
  vec point2;    // INV: point1 != point2
};
```

Punkt und Richtung

```
struct line {
  vec point;
  vec dir;       // INV: dir != (0,0,0)
};
```

- Beide sind richtig, aber die nächsten Teile hängen von der Wahl ab

Teil 2: Zwei Möglichkeiten

Zwei Punkte

```
struct line {
  vec point1;
  vec point2;    // INV: point1 != point2
};
```

Punkt und Richtung

```
struct line {
  vec point;
  vec dir;       // INV: dir != (0,0,0)
};
```

- Beide sind richtig, aber die nächsten Teile hängen von der Wahl ab
- Wir nehmen zwei Punkte, wie die Musterlösung auf **code** expert

Teil 2: Zwei Möglichkeiten

Zwei Punkte

```
struct line {
  vec point1;
  vec point2;    // INV: point1 != point2
};
```

Punkt und Richtung

```
struct line {
  vec point;
  vec dir;       // INV: dir != (0,0,0)
};
```

- Beide sind richtig, aber die nächsten Teile hängen von der Wahl ab
- Wir nehmen zwei Punkte, wie die Musterlösung auf **code** expert
- Wozu die Invariante? Durch einen einzigen Punkt gehen unendlich viele Geraden

Teil 3: Gerade verschieben

Aufgabe

Implementiert eine Funktion, die eine Gerade um einen Vektor verschiebt.

Teil 3: Gerade verschieben

Aufgabe

Implementiert eine Funktion, die eine Gerade um einen Vektor verschiebt.

```
// POST: returns a new line obtained by shifting l by v
line shift_line(const line& l, const vec& v) {
  return {sum(l.point1, v), sum(l.point2, v)};
}
```

Beide Punkte verschieben, `sum` gibt es schon.

Teile 4 und 5: `operator+`

```
vec operator+(const vec& a, const vec& b) {
  return sum(a, b);
}

line operator+(const line& l, const vec& v) {
  return shift_line(l, v);
  // or: return {l.point1 + v, l.point2 + v};
}
```

Teile 4 und 5: `operator+`

```
vec operator+(const vec& a, const vec& b) {
  return sum(a, b);
}

line operator+(const line& l, const vec& v) {
  return shift_line(l, v);
  // or: return {l.point1 + v, l.point2 + v};
}
```

- Ein Operator ist eine Funktion mit besonderem Namen

Teile 4 und 5: `operator+`

```
vec operator+(const vec& a, const vec& b) {
  return sum(a, b);
}

line operator+(const line& l, const vec& v) {
  return shift_line(l, v);
  // or: return {l.point1 + v, l.point2 + v};
}
```

- Ein Operator ist eine Funktion mit besonderem Namen
- Geht auch `v + l`?

Teile 4 und 5: `operator+`

```
vec operator+(const vec& a, const vec& b) {
  return sum(a, b);
}

line operator+(const line& l, const vec& v) {
  return shift_line(l, v);
  // or: return {l.point1 + v, l.point2 + v};
}
```

- Ein Operator ist eine Funktion mit besonderem Namen
- Geht auch `v + l`? Nein, wir haben nur `line + vec` definiert

Teil 6 (optional): operator<<

```
std::ostream& operator<<(std::ostream& os, const vec& v) {
  os << "(" << v.x << "," << v.y << "," << v.z << ")";
  return os;
}

std::ostream& operator<<(std::ostream& os, const line& l) {
  os << l.point1 << " <-> " << l.point2;
  return os;
}
```

Teil 6 (optional): operator<<

```
std::ostream& operator<<(std::ostream& os, const vec& v) {
  os << "(" << v.x << "," << v.y << "," << v.z << ")";
  return os;
}

std::ostream& operator<<(std::ostream& os, const line& l) {
  os << l.point1 << " <-> " << l.point2;
  return os;
}
```

- Warum `return os`?

Teil 6 (optional): operator<<

```
std::ostream& operator<<(std::ostream& os, const vec& v) {
  os << "(" << v.x << "," << v.y << "," << v.z << ")";
  return os;
}

std::ostream& operator<<(std::ostream& os, const line& l) {
  os << l.point1 << " <-> " << l.point2;
  return os;
}
```

- Warum `return os`? `std::cout << a << b` ist `(std::cout << a) << b`: das nächste << braucht wieder den Stream

Teil 6 (optional): operator<<

```
std::ostream& operator<<(std::ostream& os, const vec& v) {
  os << "(" << v.x << "," << v.y << "," << v.z << ")";
  return os;
}

std::ostream& operator<<(std::ostream& os, const line& l) {
  os << l.point1 << " <-> " << l.point2;
  return os;
}
```

- Warum `return os`? `std::cout << a << b` ist `(std::cout << a) << b`: das nächste << braucht wieder den Stream
- Mit `vec a = {1, 2, 3}, b = {4, -2, 1}; line l = {a, b}; std::cout << l + (a + b);` gibt `(6,2,7) <-> (9,-2,5)` aus

Fragen/Unklarheiten?

8. Rekursion

Das kommt in den Übungen

Das kommt in den Übungen

- Die Rekursion wird jetzt anspruchsvoller

Das kommt in den Übungen

- Die Rekursion wird jetzt anspruchsvoller
- Das hier bereitet euch auf die Rekursionsaufgaben dieser Woche vor

Das kommt in den Übungen

- Die Rekursion wird jetzt anspruchsvoller
- Das hier bereitet euch auf die Rekursionsaufgaben dieser Woche vor
- Öffnet “Power Set” auf **code** expert und schaut in `main.cpp`

`CharSet`: eine Menge von Zeichen

```
CharSet empty = CharSet();                        // {}
CharSet t     = CharSet('a');                     // {a}
CharSet s     = CharSet({'x', 'a', 'y', 'b'});    // {a,b,x,y}
```

`CharSet`: eine Menge von Zeichen

```
CharSet empty = CharSet();                        // {}
CharSet t     = CharSet('a');                     // {a}
CharSet s     = CharSet({'x', 'a', 'y', 'b'});    // {a,b,x,y}
```

- Benutzt es wie ein **`struct`**: wie es gebaut ist, müsst ihr nicht verstehen

`CharSet`: eine Menge von Zeichen

```
CharSet empty = CharSet();                        // {}
CharSet t     = CharSet('a');                     // {a}
CharSet s     = CharSet({'x', 'a', 'y', 'b'});    // {a,b,x,y}
```

- Benutzt es wie ein **`struct`**: wie es gebaut ist, müsst ihr nicht verstehen
- Die Elemente sind automatisch sortiert

`CharSet`: eine Menge von Zeichen

```
CharSet empty = CharSet();                      // {}
CharSet t     = CharSet('a');                   // {a}
CharSet s     = CharSet({'x', 'a', 'y', 'b'});  // {a,b,x,y}
```

- Benutzt es wie ein **`struct`**: wie es gebaut ist, müsst ihr nicht verstehen
- Die Elemente sind automatisch sortiert
- `SetOfCharSets` funktioniert genauso, die Elemente sind dort `CharSets`

`CharSet`: was man damit machen kann

```
t.insert('c');   // t is now {a,c}
t.remove('z');   // not in t: nothing happens

s + t            // {a,b,c,x,y}    union
s - t            // {b,x,y}        difference
s[0]             // 'a'            read only
s.size()         // 4
t = s;           // t is now a copy of s
```

Alle Beispiele stehen in `set_demonstration()` in `main.cpp`.

Aufgabe “Power Set”

Aufgabe

Implementiert `power_set`. Die Potenzmenge von S ist die Menge aller Teilmengen von S:

$$2^S := \{X \mid X \subseteq S\}$$

Beispiel: $2^{\{a,b,c\}} = \{\{\}, \{a\}, \{b\}, \{c\}, \{a,b\}, \{a,c\}, \{b,c\}, \{a,b,c\}\}$

Alleine, mit Stift und Papier, 10 Minuten: Wie würdet ihr das lösen?

Welche Fälle kennt ihr sofort? Was ändert sich, wenn ein Element dazukommt?

Klein anfangen

Wie bei einem Induktionsbeweis: zuerst der einfachste Fall, dann Schritt für Schritt grösser.

Klein anfangen

Wie bei einem Induktionsbeweis: zuerst der einfachste Fall, dann Schritt für Schritt grösser.

$$2^{\{\}} =$$

Klein anfangen

Wie bei einem Induktionsbeweis: zuerst der einfachste Fall, dann Schritt für Schritt grösser.

$$2^{\{\}} = \{\{\}\}$$

Klein anfangen

Wie bei einem Induktionsbeweis: zuerst der einfachste Fall, dann Schritt für Schritt grösser.

$$
\begin{aligned}
2^{\{\}} &= \{\{\}\} \\
2^{\{a\}} &=
\end{aligned}
$$

Klein anfangen

Wie bei einem Induktionsbeweis: zuerst der einfachste Fall, dann Schritt für Schritt grösser.

$$
\begin{aligned}
2^{\{\}} &= \{\{\}\} \\
2^{\{a\}} &= \{\{\}, \{a\}\}
\end{aligned}
$$

Klein anfangen

Wie bei einem Induktionsbeweis: zuerst der einfachste Fall, dann Schritt für Schritt grösser.

$$
\begin{aligned}
2^{\{\}} &= \{\{\}\} \\
2^{\{a\}} &= \{\{\}, \{a\}\} = \{\{\}\} \cup \{\{a\}\}
\end{aligned}
$$

Klein anfangen

Wie bei einem Induktionsbeweis: zuerst der einfachste Fall, dann Schritt für Schritt grösser.

$$
\begin{aligned}
2^{\{\}} &= \{\{\}\} \\
2^{\{a\}} &= \{\{\}, \{a\}\} = \{\{\}\} \cup \{\{a\}\} \\
2^{\{a,b\}} &=
\end{aligned}
$$

Klein anfangen

Wie bei einem Induktionsbeweis: zuerst der einfachste Fall, dann Schritt für Schritt grösser.

$$
\begin{aligned}
2^{\{\}} &= \{\{\}\} \\
2^{\{a\}} &= \{\{\}, \{a\}\} = \{\{\}\} \cup \{\{a\}\} \\
2^{\{a,b\}} &= \{\{\}, \{a\}, \{b\}, \{a, b\}\}
\end{aligned}
$$

Klein anfangen

Wie bei einem Induktionsbeweis: zuerst der einfachste Fall, dann Schritt für Schritt grösser.

$$
\begin{aligned}
2^{\{\}} &= \{\{\}\} \\
2^{\{a\}} &= \{\{\}, \{a\}\} = \{\{\}\} \cup \{\{a\}\} \\
2^{\{a,b\}} &= \{\{\}, \{a\}, \{b\}, \{a,b\}\} = \{\{\}, \{a\}\} \cup \{\{b\}, \{a,b\}\}
\end{aligned}
$$

Klein anfangen

Wie bei einem Induktionsbeweis: zuerst der einfachste Fall, dann Schritt für Schritt grösser.

$$
\begin{aligned}
2^{\{\}} &= \{\{\}\} \\
2^{\{a\}} &= \{\{\}, \{a\}\} = \{\{\}\} \cup \{\{a\}\} \\
2^{\{a,b\}} &= \{\{\}, \{a\}, \{b\}, \{a,b\}\} = \{\{\}, \{a\}\} \cup \{\{b\}, \{a,b\}\}
\end{aligned}
$$

Es fällt auf: $2^{\{\}} \subset 2^{\{a\}} \subset 2^{\{a,b\}}$

Klein anfangen

Wie bei einem Induktionsbeweis: zuerst der einfachste Fall, dann Schritt für Schritt grösser.

$$
\begin{aligned}
2^{\{\}} &= \{\{\}\} \\
2^{\{a\}} &= \{\{\}, \{a\}\} = \{\{\}\} \cup \{\{a\}\} \\
2^{\{a,b\}} &= \{\{\}, \{a\}, \{b\}, \{a,b\}\} = \{\{\}, \{a\}\} \cup \{\{b\}, \{a,b\}\}
\end{aligned}
$$

Es fällt auf: $2^{\{\}} \subset 2^{\{a\}} \subset 2^{\{a,b\}}$

Allgemein: $S_1 \subset S_2 \Rightarrow 2^{S_1} \subset 2^{S_2}$ (jede Teilmenge von S_1 ist auch Teilmenge von S_2)

Das Muster

$\{\}$ $\{a\}$ $\{b\}$ $\{a, b\}$ $\{c\}$ $\{a, c\}$ $\{b, c\}$ $\{a, b, c\}$

Das Muster

$$\underbrace{\{\}}_{2^{\{\}}} \quad \{a\} \quad \{b\} \quad \{a,b\} \quad \{c\} \quad \{a,c\} \quad \{b,c\} \quad \{a,b,c\}$$

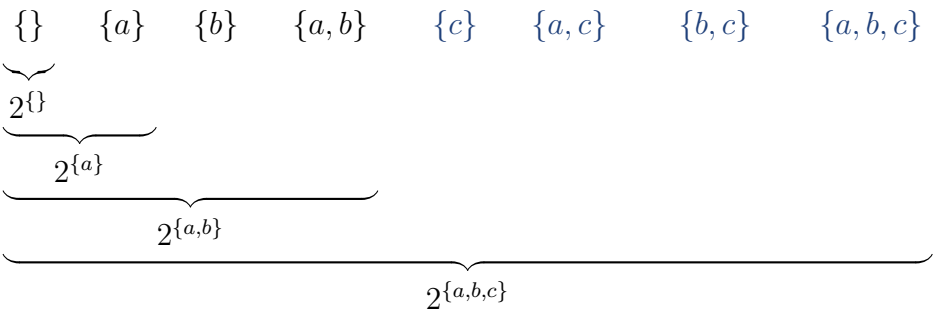
Das Muster

$$\underbrace{\underbrace{\{\}}_{2^{\{\}}} \quad \{a\}}_{2^{\{a\}}} \quad \{b\} \quad \{a,b\} \quad \{c\} \quad \{a,c\} \quad \{b,c\} \quad \{a,b,c\}$$

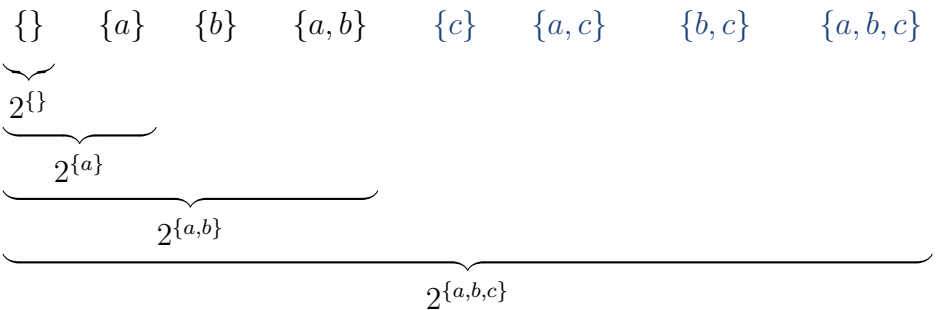
Das Muster

$$\underbrace{\{\}}_{2^{\{\}}} \quad \underbrace{\{a\}}_{2^{\{a\}}} \quad \underbrace{\{b\} \quad \{a,b\}}_{2^{\{a,b\}}} \quad \{c\} \quad \{a,c\} \quad \{b,c\} \quad \{a,b,c\}$$

Das Muster

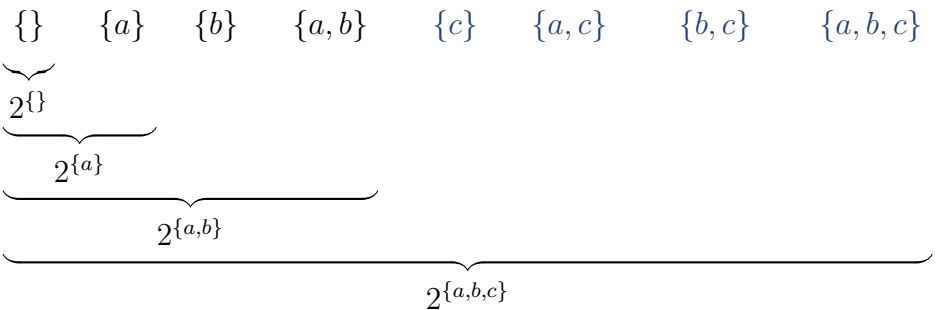


Das Muster



Mit jedem neuen Element verdoppelt sich die Anzahl der Teilmengen:

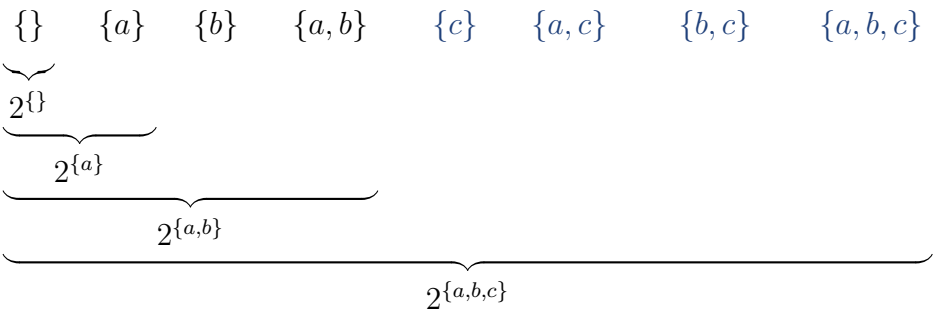
Das Muster



Mit jedem neuen Element verdoppelt sich die Anzahl der Teilmengen:

- Die Hälfte ist alt: die Potenzmenge ohne das neue Element

Das Muster



Mit jedem neuen Element verdoppelt sich die Anzahl der Teilmengen:

- Die Hälfte ist alt: die Potenzmenge ohne das neue Element
- Die andere Hälfte ist neu: jede alte Teilmenge plus das neue Element

Mathematisch aufgeschrieben

$$2^{\{a,b,c\}} = \{\{\}, \{a\}, \{b\}, \{c\}, \{a,b\}, \{a,c\}, \{b,c\}, \{a,b,c\}\}$$

Mathematisch aufgeschrieben

$$
\begin{aligned}
2^{\{a,b,c\}} &= \{\{\}, \{a\}, \{b\}, \{c\}, \{a,b\}, \{a,c\}, \{b,c\}, \{a,b,c\}\} \\
&= \{\{\}, \{a\}, \{b\}, \{a,b\}\} \cup \{\{c\}, \{a,c\}, \{b,c\}, \{a,b,c\}\}
\end{aligned}
$$

Mathematisch aufgeschrieben

$$
\begin{aligned}
2^{\{a,b,c\}} &= \{\{\}, \{a\}, \{b\}, \{c\}, \{a,b\}, \{a,c\}, \{b,c\}, \{a,b,c\}\} \\
&= \{\{\}, \{a\}, \{b\}, \{a,b\}\} \cup \{\{c\}, \{a,c\}, \{b,c\}, \{a,b,c\}\} \\
&= \{\{\}, \{a\}, \{b\}, \{a,b\}\} \cup \{\, Y \cup \{c\} \mid Y \in \{\{\}, \{a\}, \{b\}, \{a,b\}\}\,\}
\end{aligned}
$$

Mathematisch aufgeschrieben

$$
\begin{aligned}
2^{\{a,b,c\}} &= \{\{\}, \{a\}, \{b\}, \{c\}, \{a,b\}, \{a,c\}, \{b,c\}, \{a,b,c\}\} \\
&= \{\{\}, \{a\}, \{b\}, \{a,b\}\} \cup \{\{c\}, \{a,c\}, \{b,c\}, \{a,b,c\}\} \\
&= \{\{\}, \{a\}, \{b\}, \{a,b\}\} \cup \{\, Y \cup \{c\} \mid Y \in \{\{\}, \{a\}, \{b\}, \{a,b\}\}\,\} \\
&= 2^{\{a,b\}} \cup \{\, Y \cup \{c\} \mid Y \in 2^{\{a,b\}}\,\}
\end{aligned}
$$

Erster Entwurf

1. Wähle ein Zeichen $x \in S$
2. $S' := S \setminus \{x\}$
3. Berechne $2^{S'}$
4. Gib zurück: $2^S = 2^{S'} \cup \{ Y \cup \{x\} \mid Y \in 2^{S'} \}$

Erster Entwurf

1. Wähle ein Zeichen $x \in S$
2. $S' := S \setminus \{x\}$
3. Berechne $2^{S'}$
4. Gib zurück: $2^S = 2^{S'} \cup \{ Y \cup \{x\} \mid Y \in 2^{S'} \}$

- Wie berechnen wir $2^{S'}$?

Erster Entwurf

1. Wähle ein Zeichen $x \in S$
2. $S' := S \setminus \{x\}$
3. Berechne $2^{S'}$
4. Gib zurück: $2^S = 2^{S'} \cup \{ Y \cup \{x\} \mid Y \in 2^{S'} \}$

- Wie berechnen wir $2^{S'}$? Mit genau diesem Algorithmus: rekursiv!

Erster Entwurf

1. Wähle ein Zeichen $x \in S$
2. $S' := S \setminus \{x\}$
3. Berechne $2^{S'}$
4. Gib zurück: $2^S = 2^{S'} \cup \{ Y \cup \{x\} \mid Y \in 2^{S'} \}$

- Wie berechnen wir $2^{S'}$? Mit genau diesem Algorithmus: rekursiv!
- Funktioniert das schon?

Erster Entwurf

1. Wähle ein Zeichen $x \in S$
2. $S' := S \setminus \{x\}$
3. Berechne $2^{S'}$
4. Gib zurück: $2^S = 2^{S'} \cup \{ Y \cup \{x\} \mid Y \in 2^{S'} \}$

- Wie berechnen wir $2^{S'}$? Mit genau diesem Algorithmus: rekursiv!
- Funktioniert das schon? Nein: für $S = \{\}$ gibt es in Schritt 1 kein Zeichen. Wir brauchen einen Basisfall

Erster Entwurf

1. Wähle ein Zeichen $x \in S$
2. $S' := S \setminus \{x\}$
3. Berechne $2^{S'}$
4. Gib zurück: $2^S = 2^{S'} \cup \{ Y \cup \{x\} \mid Y \in 2^{S'} \}$

- Wie berechnen wir $2^{S'}$? Mit genau diesem Algorithmus: rekursiv!
- Funktioniert das schon? Nein: für $S = \{\}$ gibt es in Schritt 1 kein Zeichen. Wir brauchen einen Basisfall
- Terminiert es dann?

Erster Entwurf

1. Wähle ein Zeichen $x \in S$
2. $S' := S \setminus \{x\}$
3. Berechne $2^{S'}$
4. Gib zurück: $2^S = 2^{S'} \cup \{ Y \cup \{x\} \mid Y \in 2^{S'} \}$

- Wie berechnen wir $2^{S'}$? Mit genau diesem Algorithmus: rekursiv!
- Funktioniert das schon? Nein: für $S = \{\}$ gibt es in Schritt 1 kein Zeichen. Wir brauchen einen Basisfall
- Terminiert es dann? Ja: S wird bei jedem Aufruf um ein Element kleiner, also erreichen wir $\{\}$

Der ganze Algorithmus

1. Falls $S = \{\}$ (Basisfall):
   a) Gib $\{\{\}\}$ zurück
2. Sonst:
   a) Wähle ein Zeichen $x \in S$
   b) $S' := S \setminus \{x\}$
   c) Berechne $2^{S'}$ rekursiv
   d) Gib zurück: $2^S = 2^{S'} \cup \{\, Y \cup \{x\} \mid Y \in 2^{S'} \,\}$

Aufgabe “Power Set”: jetzt programmieren

Besprecht eure Ideen und schreibt das Programm.

Zum Debuggen: Ausgaben am Anfang und vor dem `return`

```
std::cerr << "CharSet start: " << set << std::endl;
std::cerr << "SetOfCharSets end: " << result << std::endl;
```

Eingabe: `3 a b c`

Zu zweit, 10 Minuten.

Mögliche Lösung (1/2)

```
SetOfCharSets power_set(const CharSet& set) {
  // base case: empty set
  if (std::ssize(set) == 0) {
    return SetOfCharSets(CharSet());
  }

  // split set into its first element and the rest
  CharSet first_element_subset = CharSet(set[0]);
  CharSet remaining_subset = set - first_element_subset;
```

Auf der Tafel nahmen wir das letzte Element, hier ist das erste einfacher zu programmieren.

Mögliche Lösung (2/2)

```
  // get power set for remaining subset
  SetOfCharSets remaining_subset_power_set = power_set(remaining_subset);

  // init result with power set of remaining subset
  SetOfCharSets result = remaining_subset_power_set;

  // add first element to every set in the powerset
  for (int i = 0; i < remaining_subset_power_set.size(); ++i) {
    result.insert(first_element_subset + remaining_subset_power_set[i]);
  }

  return result;
}
```

“Power Set” — Aufrufe für $\{a, b, c\}$

Aufruf	passiert

“Power Set” — Aufrufe für $\{a, b, c\}$

Aufruf	passiert
`power_set(`$\{a, b, c\}$`)`	ruft `power_set(`$\{b, c\}$`)`

“Power Set” — Aufrufe für $\{a, b, c\}$

Aufruf	passiert
`power_set(`$\{a, b, c\}$`)`	ruft `power_set(`$\{b, c\}$`)`
`power_set(`$\{b, c\}$`)`	ruft `power_set(`$\{c\}$`)`

“Power Set” — Aufrufe für $\{a, b, c\}$

Aufruf	passiert
`power_set(`$\{a, b, c\}$`)`	ruft `power_set(`$\{b, c\}$`)`
`power_set(`$\{b, c\}$`)`	ruft `power_set(`$\{c\}$`)`
`power_set(`$\{c\}$`)`	ruft `power_set({})`

“Power Set” — Aufrufe für $\{a, b, c\}$

Aufruf	passiert
`power_set(`$\{a, b, c\}$`)`	ruft `power_set(`$\{b, c\}$`)`
`power_set(`$\{b, c\}$`)`	ruft `power_set(`$\{c\}$`)`
`power_set(`$\{c\}$`)`	ruft `power_set({})`
`power_set({})`	Basisfall: $\{\{\}\}$

“Power Set” — Aufrufe für $\{a, b, c\}$

Aufruf	passiert
`power_set(`$\{a, b, c\}$`)`	ruft `power_set(`$\{b, c\}$`)`
`power_set(`$\{b, c\}$`)`	ruft `power_set(`$\{c\}$`)`
`power_set(`$\{c\}$`)`	ruft `power_set({})`
`power_set({})`	Basisfall: $\{\{\}\}$
`power_set(`$\{c\}$`)`	$\{\{\}\} \cup \{\{c\}\}$

“Power Set” — Aufrufe für $\{a, b, c\}$

Aufruf	passiert
`power_set(`$\{a, b, c\}$`)`	ruft `power_set(`$\{b, c\}$`)`
`power_set(`$\{b, c\}$`)`	ruft `power_set(`$\{c\}$`)`
`power_set(`$\{c\}$`)`	ruft `power_set({})`
`power_set({})`	Basisfall: $\{\{\}\}$
`power_set(`$\{c\}$`)`	$\{\{\}\} \cup \{\{c\}\}$
`power_set(`$\{b, c\}$`)`	$\{\{\}, \{c\}\} \cup \{\{b\}, \{b, c\}\}$

“Power Set” — Aufrufe für $\{a, b, c\}$

Aufruf	passiert
`power_set(`$\{a, b, c\}$`)`	ruft `power_set(`$\{b, c\}$`)`
`power_set(`$\{b, c\}$`)`	ruft `power_set(`$\{c\}$`)`
`power_set(`$\{c\}$`)`	ruft `power_set(`$\{\}$`)`
`power_set(`$\{\}$`)`	Basisfall: $\{\{\}\}$
`power_set(`$\{c\}$`)`	$\{\{\}\} \cup \{\{c\}\}$
`power_set(`$\{b, c\}$`)`	$\{\{\}, \{c\}\} \cup \{\{b\}, \{b, c\}\}$
`power_set(`$\{a, b, c\}$`)`	$\{\{\}, \{b\}, \{c\}, \{b, c\}\} \cup \{\{a\}, \{a, b\}, \{a, c\}, \{a, b, c\}\}$

“Power Set” — Aufrufe für $\{a, b, c\}$

Aufruf	passiert
`power_set(`$\{a, b, c\}$`)`	ruft `power_set(`$\{b, c\}$`)`
`power_set(`$\{b, c\}$`)`	ruft `power_set(`$\{c\}$`)`
`power_set(`$\{c\}$`)`	ruft `power_set({})`
`power_set({})`	Basisfall: $\{\{\}\}$
`power_set(`$\{c\}$`)`	$\{\{\}\} \cup \{\{c\}\}$
`power_set(`$\{b, c\}$`)`	$\{\{\}, \{c\}\} \cup \{\{b\}, \{b, c\}\}$
`power_set(`$\{a, b, c\}$`)`	$\{\{\}, \{b\}, \{c\}, \{b, c\}\} \cup \{\{a\}, \{a, b\}, \{a, c\}, \{a, b, c\}\}$

Beim Schreiben müsst ihr nicht so weit denken. Nehmt an, dass der kleinere Aufruf schon stimmt (*Recursive Leap of Faith*).

Fragen/Unklarheiten?

9. Extra: Rekursion in Iteration umschreiben

Aufgabe “Recursion to Iteration 1”

Aufgabe

Schreibt die folgende rekursive Funktion in iterativer Form.

```
// PRE: n >= 0
int f(const int n) {
  if (n <= 2) return 1;
  return f(n-1) + 2 * f(n-3);
}
```

Alleine, mit Stift und Papier, 10 Minuten. $f(0), \ldots, f(7) = 1, 1, 1, 3, 5, 7, 13, 23$

“Recursion to Iteration 1” — Lösung

```cpp
// PRE: n >= 0
int f_it(const int n) {
  if (n <= 2) return 1;
  int a = 1;  // f(0)
  int b = 1;  // f(1)
  int c = 1;  // f(2)
  for (int i = 3; i < n; ++i) {
    const int a_prev = a;  // f(i-3)
    a = b;                 // f(i-2)
    b = c;                 // f(i-1)
    c = b + 2 * a_prev;    // f(i)
  }
  return c + 2 * a;        // f(n-1) + 2 * f(n-3)
}
```

“Recursion to Iteration 1” — Lösung

```
// PRE: n >= 0
int f_it(const int n) {
  if (n <= 2) return 1;
  int a = 1;  // f(0)
  int b = 1;  // f(1)
  int c = 1;  // f(2)
  for (int i = 3; i < n; ++i) {
    const int a_prev = a;  // f(i-3)
    a = b;                 // f(i-2)
    b = c;                 // f(i-1)
    c = b + 2 * a_prev;    // f(i)
  }
  return c + 2 * a;        // f(n-1) + 2 * f(n-3)
}
```

$f(n)$ braucht nur die letzten drei Werte, also reichen drei Variablen.

Aufgabe “Recursion to Iteration 2”

Aufgabe

Schreibt die folgende rekursive Funktion in iterativer Form.

```
// PRE: n >= 0
int f(const int n) {
  if (n == 0) return 1;
  return f(n - 1) + 2 * f(n / 2);
}
```

Alleine, mit Stift und Papier, 10 Minuten.

$f(0), \ldots, f(7) = 1, 3, 9, 15, 33, 51, 81, 111$

“Recursion to Iteration 2” — Lösung

```cpp
// PRE: n >= 0
int f_it(const int n) {
  if (n == 0) return 1;  // special case
  std::vector<int> f_values(n + 1, 0);
  f_values[0] = 1;
  for (int i = 1; i <= n; ++i) {
    f_values[i] = f_values[i - 1] + 2 * f_values[i / 2];
  }
  return f_values[n];
}
```

“Recursion to Iteration 2” — Lösung

```cpp
// PRE: n >= 0
int f_it(const int n) {
  if (n == 0) return 1;  // special case
  std::vector<int> f_values(n + 1, 0);
  f_values[0] = 1;
  for (int i = 1; i <= n; ++i) {
    f_values[i] = f_values[i - 1] + 2 * f_values[i / 2];
  }
  return f_values[n];
}
```

$f(n/2)$ liegt beliebig weit zurück, darum merken wir uns alle Werte.

Fragen/Unklarheiten?

10. Extra: Wiederholung Fliesskommazahlen

Aufgabe “Floating Point Number Systems”

Aufgabe

Berechnet in $F^*(2, 3, -4, 4)$ wie die Klammern es verlangen, und stellt jedes Zwischenresultat im System dar:

$$(10 + 0.5) + 0.5 \qquad\qquad (0.5 + 0.5) + 10$$

Alleine, 5 Minuten.

Lösung

$(10 + 0.5) + 0.5$

dezimal	binär
$+$	
$=$	
$\approx$	
$+$	
$\approx$	

$(0.5 + 0.5) + 10$

dezimal	binär
$+$	
$=$	
$+$	
$=$	
$\approx$	

Lösung

$(10 + 0.5) + 0.5$

	dezimal	binär
	10	$1.01 \cdot 2^3$
$+$	0.5	$0.0001 \cdot 2^3$
$=$		
$\approx$		
$+$		
$\approx$		

$(0.5 + 0.5) + 10$

	dezimal	binär
$+$		
$=$		
$+$		
$=$		
$\approx$		

Lösung

$(10 + 0.5) + 0.5$

	dezimal	binär
	10	$1.01 \cdot 2^3$
$+$	0.5	$0.0001 \cdot 2^3$
$=$		$1.01\color{magenta}{01} \cdot 2^3$
$\approx$		
$+$		
$\approx$		

$(0.5 + 0.5) + 10$

	dezimal	binär
$+$		
$=$		
$+$		
$=$		
$\approx$		

Lösung

$(10 + 0.5) + 0.5$

	dezimal	binär
	10	$1.01 \cdot 2^3$
$+$	0.5	$0.0001 \cdot 2^3$
$=$		$1.01\color{magenta}{01} \cdot 2^3$
$\approx$		$1.01 \cdot 2^3$
$+$		
$\approx$		

$(0.5 + 0.5) + 10$

	dezimal	binär
$+$		
$=$		
$+$		
$=$		
$\approx$		

Lösung

$(10 + 0.5) + 0.5$

	dezimal	binär
	10	$1.01 \cdot 2^3$
$+$	0.5	$0.0001 \cdot 2^3$
$=$		$1.0101 \cdot 2^3$
$\approx$		$1.01 \cdot 2^3$
$+$	0.5	$0.0001 \cdot 2^3$
$\approx$		

$(0.5 + 0.5) + 10$

	dezimal	binär
$+$		
$=$		
$+$		
$=$		
$\approx$		

Lösung

$(10 + 0.5) + 0.5$

	dezimal	binär
	10	$1.01 \cdot 2^3$
$+$	0.5	$0.0001 \cdot 2^3$
$=$		$1.0101 \cdot 2^3$
$\approx$		$1.01 \cdot 2^3$
$+$	0.5	$0.0001 \cdot 2^3$
$\approx$	10	$1.01 \cdot 2^3$

$(0.5 + 0.5) + 10$

	dezimal	binär
$+$		
$=$		
$+$		
$=$		
$\approx$		

Lösung

$(10 + 0.5) + 0.5$

	dezimal	binär
	10	$1.01 \cdot 2^3$
$+$	0.5	$0.0001 \cdot 2^3$
$=$		$1.0101 \cdot 2^3$
$\approx$		$1.01 \cdot 2^3$
$+$	0.5	$0.0001 \cdot 2^3$
$\approx$	10	$1.01 \cdot 2^3$

$(0.5 + 0.5) + 10$

	dezimal	binär
	0.5	$1.00 \cdot 2^{-1}$
$+$	0.5	$1.00 \cdot 2^{-1}$
$=$		
$+$		
$=$		
$\approx$		

Lösung

$(10 + 0.5) + 0.5$

	dezimal	binär
	10	$1.01 \cdot 2^3$
$+$	0.5	$0.0001 \cdot 2^3$
$=$		$1.0101 \cdot 2^3$
$\approx$		$1.01 \cdot 2^3$
$+$	0.5	$0.0001 \cdot 2^3$
$\approx$	10	$1.01 \cdot 2^3$

$(0.5 + 0.5) + 10$

	dezimal	binär
	0.5	$1.00 \cdot 2^{-1}$
$+$	0.5	$1.00 \cdot 2^{-1}$
$=$	1	$1.00 \cdot 2^0$
$+$		
$=$		
$\approx$		

Lösung

$(10 + 0.5) + 0.5$

	dezimal	binär
	10	$1.01 \cdot 2^3$
$+$	0.5	$0.0001 \cdot 2^3$
$=$		$1.0101 \cdot 2^3$
$\approx$		$1.01 \cdot 2^3$
$+$	0.5	$0.0001 \cdot 2^3$
$\approx$	10	$1.01 \cdot 2^3$

$(0.5 + 0.5) + 10$

	dezimal	binär
	0.5	$1.00 \cdot 2^{-1}$
$+$	0.5	$1.00 \cdot 2^{-1}$
$=$	1	$1.00 \cdot 2^0$
$+$	10	$1010.00 \cdot 2^0$
$=$		
$\approx$		

Lösung

$(10 + 0.5) + 0.5$

	dezimal	binär
	10	$1.01 \cdot 2^3$
$+$	0.5	$0.0001 \cdot 2^3$
$=$		$1.0101 \cdot 2^3$
$\approx$		$1.01 \cdot 2^3$
$+$	0.5	$0.0001 \cdot 2^3$
$\approx$	10	$1.01 \cdot 2^3$

$(0.5 + 0.5) + 10$

	dezimal	binär
	0.5	$1.00 \cdot 2^{-1}$
$+$	0.5	$1.00 \cdot 2^{-1}$
$=$	1	$1.00 \cdot 2^0$
$+$	10	$1010.00 \cdot 2^0$
$=$	11	$1.011 \cdot 2^3$
$\approx$		

Lösung

$(10 + 0.5) + 0.5$

	dezimal	binär
	10	$1.01 \cdot 2^3$
$+$	0.5	$0.0001 \cdot 2^3$
$=$		$1.0101 \cdot 2^3$
$\approx$		$1.01 \cdot 2^3$
$+$	0.5	$0.0001 \cdot 2^3$
$\approx$	10	$1.01 \cdot 2^3$

$(0.5 + 0.5) + 10$

	dezimal	binär
	0.5	$1.00 \cdot 2^{-1}$
$+$	0.5	$1.00 \cdot 2^{-1}$
$=$	1	$1.00 \cdot 2^0$
$+$	10	$1010.00 \cdot 2^0$
$=$	11	$1.011 \cdot 2^3$
$\approx$	12	$1.10 \cdot 2^3$

Lösung

$(10 + 0.5) + 0.5$

	dezimal	binär
	10	$1.01 \cdot 2^3$
$+$	0.5	$0.0001 \cdot 2^3$
$=$		$1.0101 \cdot 2^3$
$\approx$		$1.01 \cdot 2^3$
$+$	0.5	$0.0001 \cdot 2^3$
$\approx$	10	$1.01 \cdot 2^3$

$(0.5 + 0.5) + 10$

	dezimal	binär
	0.5	$1.00 \cdot 2^{-1}$
$+$	0.5	$1.00 \cdot 2^{-1}$
$=$	1	$1.00 \cdot 2^0$
$+$	10	$1010.00 \cdot 2^0$
$=$	11	$1.011 \cdot 2^3$
$\approx$	12	$1.10 \cdot 2^3$

Die Reihenfolge ändert das Resultat. Die Addition ist hier nicht assoziativ.

Fragen/Unklarheiten?

11. Outro

Allgemeine Fragen?

Bis nächste Woche!