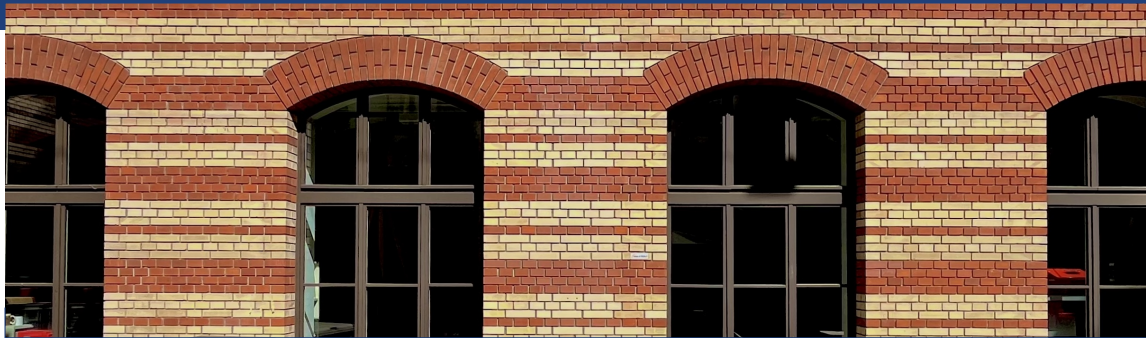




Informatik

Zeichen, Rekursion

Adel Gavranović

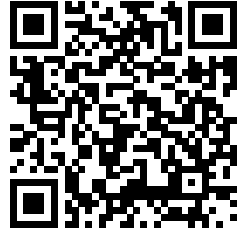
ETH Zürich — 2026

Überblick

Zeichen (char)

Rekursion

Extra: Fortgeschrittene Rekursion



adelgavranovic.ch

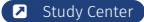
 Webpage

 Mail

1. Follow-up

2. Feedback zu **code** expert

Allgemeines bezüglich **code expert**

- Die Abgabefrist für Woche 6 war gestern. “Sorting by Swapping” [Hidden tests] wird jetzt mit *neuen* Tests neu bewertet, mein Feedback dazu kommt danach
- Woche 7: Wenn eine Aufgabe Rekursion verlangt, löst sie auch rekursiv. Eine Lösung ohne Rekursion, die die Aufgabe trivial macht, bekommt keine TA-Punkte (die XP behaltet ihr)
- Challenge Exercise 1: Abgabe bis 9. November, also in knapp zwei Wochen
- Bei einer Aufgabe steckengeblieben? Kommt ins Study Center (jeden Mittwoch, Infos auf der Study-Center-Webseite)! 

Fragen bezüglich **code expert** eurerseits?

3. Lernziele

Ziele

- ☐ Programme tracen und schreiben können, die ASCII-Zeichen verändern
- ☐ Einfache rekursive Algorithmen in C++ implementieren können

4. Zusammenfassung

Getting on the same page

- Was wurde diese Woche behandelt?

5. Zeichen (char)

char: Zeichen sind Zahlen

```
char ch = 'd';  
int i = ch;    // char --> int: 100  
++ch;         // 101, i.e. 'e'  
++i;  
std::cout << (ch == i); // 1
```

- 'a': ein Zeichen (*einfache* Anführungszeichen)
"Hello": ein String (*doppelte*)
- Ein **char** ist eine kleine Ganzzahl. Welche Zahl zu welchem Zeichen gehört, sagt die ASCII-Tabelle
- a-z, A-Z und 0-9 liegen je am Stück und in der richtigen Reihenfolge

ASCII = American Standard Code for Information Interchange

 ASCII-Tabelle

Mit Zeichen rechnen: Was wird ausgegeben?

Code	Ausgabe	
<code>std::cout << 'a' - 'A';</code>	32	Abstand klein → gross
<code>std::cout << '7' - '0';</code>	7	Ziffer → ihr Wert
<code>char c = 'a' + 1; std::cout << c; b</code>		als <code>char</code> gespeichert
<code>std::cout << 'a' + 1;</code>	98	<code>char + int</code> ist ein <code>int</code> !

💡 Die Zahlen aus der ASCII-Tabelle müsst ihr nicht auswendig wissen: rechnet mit 'a', 'A' und '0'.

Kurz wiederholt: `push_back`

```
std::vector<char> letters; // empty, size 0
letters.push_back('h');    // h
letters.push_back('i');    // h i
std::cout << std::ssize(letters); // 2
```

- `push_back` hängt ein Element hinten an, der Vektor wird um eins länger
- Praktisch, wenn man die Länge im Voraus nicht kennt: wie bei der nächsten Aufgabe

Aufgabe “Converting Input to UPPER CASE”

Aufgabe

Schreibt ein Programm, das eine Zeichenfolge, die mit einem Zeilenumbruch endet, als Vektor von `chars` einliest. Danach gibt es die Folge aus, wobei alle *kleinbuchstaben* in *GROSSBUCHSTABEN* umgewandelt sind.

Zum Einlesen:

1. ein einzelnes Zeichen von der Standardeingabe lesen
2. es in einen Vektor von `chars` einfügen
3. wiederholen, bis ein Zeilenumbruch (`'\n'`) kommt

Die Umwandlung der ganzen Folge und die eines einzelnen Zeichens kommen in eigene Funktionen (mindestens drei Funktionen).

Beispiel: `Hello World, 42 times!` → `HELLO WORLD, 42 TIMES!`

Mal schauen, was ihr gelernt habt

- Öffnet “Converting Input to UPPER CASE” auf **code expert**
- Überlegt euch alleine, wie ihr das mit Stift und Papier lösen würdet (10 Minuten)
- Diskutiert eure Ideen zu zweit und programmiert die Lösung (15 Minuten)
- Danach diktiert ihr mir eure Lösung (10 Minuten)

💡 Variablen vom Typ **char** kann man wie Zahlen behandeln.

Mögliche Lösung (1/2)

```
#include <iostream>
#include <vector>
#include <ios>

// POST: Converts the letter to upper case.
void char_to_upper(char& letter) {
    if ('a' <= letter && letter <= 'z') {
        letter -= 'a' - 'A'; // 'a' > 'A'
    }
}

// POST: Converts all letters to upper-case.
void to_upper(std::vector<char>& letters) {
    for (int i = 0; i < std::ssize(letters); ++i) {
        char_to_upper(letters[i]);
    }
}
```

- Nur a–z ändern sich. Leerzeichen, Ziffern und ! bleiben, wie sie sind
- 'a' - 'A' ist der Abstand. Die Zahl 32 braucht ihr nicht
- **char&**: wir ändern das Zeichen *im* Vektor (Woche 6!)

Mögliche Lösung (2/2)

```
int main () {  
    // Say to std::cin not to ignore whitespace.  
    std::cin >> std::noskipws;  
    std::vector<char> letters;  
    char ch;  
    // Step 1: Read input.  
    do {  
        std::cin >> ch;  
        letters.push_back(ch);  
    } while (ch != '\n');  
    // Step 2: Convert to upper-case.  
    to_upper(letters);  
    // Step 3: Output.  
    for (int i = 0; i < std::ssize(letters); ++i) {  
        std::cout << letters[i];  
    }  
    return 0;  
}
```

- **do-while**: auch das `'\n'` landet im Vektor und wird mit ausgegeben
- Drei Schritte, jeder für sich verständlich und testbar
- Und Zeile 3?

Warum `std::noskipws`?

- `std::cin >> ch` überspringt normalerweise Leerzeichen, Tabs und Zeilenumbrüche
- Ohne `std::noskipws` kommt das `'\n'` also nie in `ch` an
- Die Schleife endet nie: das Programm hängt und gibt nichts aus (auf **code expert**: Timeout)
- Und die Leerzeichen wären auch weg

`std::noskipws` kommt aus `<ios>`. `<iostream>` bindet es in der Praxis schon mit ein.

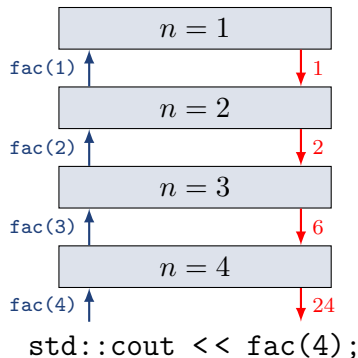
Fragen/Unklarheiten?

6. Rekursion

Rekursion: eine Funktion ruft sich selbst auf

```
// POST: return value is n!  
int fac(const int n) {  
  
    if (n <= 1) {                // base case  
        return 1;  
    }  
  
    return n * fac(n - 1);    // recursive step  
}
```

Jeder Aufruf hat seine eigenen Variablen:
beim Tracen eine eigene Tabelle pro Aufruf.



Rezept für rekursive Funktionen

1. **Basisfall:** Was ist der kleinste Fall? Den löst ihr direkt, ohne Selbstaufruf
2. **Rekursionsschritt:** Macht das Problem kleiner, ruft die Funktion damit auf und baut aus dem Resultat die Lösung
3. **Vertrauen:** Nehmt an, dass der Aufruf für den kleineren Fall schon stimmt (*Recursive Leap of Faith*). Das ist die Induktionsannahme aus der Mathematik
4. **Terminierung:** Jeder Aufruf muss dem Basisfall näher kommen, sonst ruft sich die Funktion endlos auf

Fragen/Unklarheiten?

Alte Prüfungsaufgabe

Eckdaten

- Prüfung: 2022.01 Computer Science (MATH/PHYS/RW)
- Einfache Rekursionsaufgabe
- Punkte für die ganze Prüfung: 84
- Zeit für die ganze Prüfung: 120 Minuten
- Punkte für die Aufgabe: 5
- Geschätzte Zeit für die Aufgabe: 7 Minuten $\approx 120 \cdot 5/84$

💡 An der Prüfung kommen Aufgaben in diesem Stil und dieser Schwierigkeit vor.

Kleine Prüfungssimulation

- Schaltet in den “Prüfungsmodus” und legt euch alles bereit, was ihr brauchen könntet
- Öffnet “[Exam 2022.01 (MATH/PHYS/RW)] Compute Series” auf **code expert**
- Programmiert eine (rekursive) Lösung
- Zeit: 7 Minuten

“Compute Series”

Implementiert `compute_serie_element` in `functions.h` gemäss PRE und POST.

```
// PRE:    a positive integer n
// POST:    returns the n-th number of a serie x_n, defined as
//          x_n = 2,                for n = 1
//          x_n = 1,                for n = 2
//          x_n = x_(n-1) + x_(n-2), for n > 2
// Example:
// * n == 1 ~~> 2
// * n == 2 ~~> 1
// * n == 3 ~~> 3
int compute_serie_element(int n) {
    // TODO
    return -1;
}
```

Signatur nicht ändern, keine zusätzlichen Bibliotheken, kein `assert` nötig.

“Compute Series” – Lösung

```
int compute_serie_element(int n) {  
    if (n == 1) {  
        return 2;  
    } else if (n == 2) {  
        return 1;  
    } else {  
        return compute_serie_element(n - 1) + compute_serie_element(n - 2);  
    }  
}
```

- Zwei Basisfälle, weil der Rekursionsschritt zwei Vorgänger braucht
- Die POST ist fast schon der Code: Fall für Fall abschreiben
- Korrekt, aber langsam: für $n = 40$ braucht es 204 668 309 Aufrufe

Fragen/Unklarheiten?

Aufgabe “Partial Sum”

Aufgabe

Schreibt eine Funktion, die

1. die Summe aller natürlichen Zahlen bis und mit n rekursiv berechnet und zurückgibt
2. in derselben rekursiven Funktion alle addierten Terme aufsteigend (von 0 bis n) ausgibt

Wie könntet ihr die Terme absteigend ausgeben?

Öffnet “Partial Sum” auf **code expert**

5 Minuten alleine überlegen, 5 Minuten zu zweit programmieren, dann diktiert ihr mir die Lösung.

Eingabe

5

Ausgabe

1

2

3

4

5

15

“Partial Sum” – Lösung

```
int partial_sum(const int n) {  
    if (n <= 0) {  
        return 0;  
    } else {  
        // print descending: output here  
        int partial = partial_sum(n - 1);  
        std::cout << n << "\n"; // ascending  
        return n + partial;  
    }  
}  
  
int main() {  
    int n;  
    std::cin >> n;  
    std::cout << partial_sum(n) << "\n";  
    return 0;  
}
```

- Basisfall $n \leq 0$: Summe 0, nichts ausgeben
- Ausgabe *nach* dem Aufruf: erst wenn alle kleineren Terme ausgegeben sind → aufsteigend
- Ausgabe *vor* dem Aufruf (Zeile 5) → absteigend

“Partial Sum” – Aufrufe für $n = 3$

Aufruf	passiert	Ausgabe
partial_sum(3)	ruft partial_sum(2)	
partial_sum(2)	ruft partial_sum(1)	
partial_sum(1)	ruft partial_sum(0)	
partial_sum(0)	Basisfall: gibt 0 zurück	
partial_sum(1)	gibt 1 aus, gibt $1 + 0 = 1$ zurück	1
partial_sum(2)	gibt 2 aus, gibt $2 + 1 = 3$ zurück	1 2
partial_sum(3)	gibt 3 aus, gibt $3 + 3 = 6$ zurück	1 2 3
main	gibt 6 aus	1 2 3 6

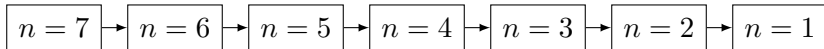
Fragen/Unklarheiten?

Aufgabe “Power Function”

Diese Funktion berechnet x^n für $n \geq 0$, mit der Idee $x^n = x \cdot x^{n-1}$:

```
int power(const int x, const int n) {  
    if (n == 0) {  
        return 1;  
    } else if (n == 1) {  
        return x;  
    }  
    return x * power(x, n - 1);  
}
```

Wie oft wird `power` aufgerufen, um x^7 zu berechnen? **7**



Aufgabe “Power Function”

Aufgabe

Für x^n braucht diese Funktion n Aufrufe. Das ist für grosse n ineffizient. Schreibt eine Funktion, die für grosse n deutlich weniger rekursive Aufrufe braucht. Wie viele Aufrufe braucht eure Lösung?

Öffnet “Power Function” auf **code expert**

10 Minuten alleine überlegen, 5 Minuten zu zweit programmieren, dann diktiert ihr mir die Lösung.

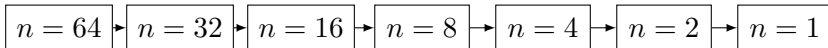
💡 Die Aufgabe verallgemeinert “Multiply with 29” aus Woche 1.

Erste Idee: $x^{2m} = x^m \cdot x^m$

```
int power(const int x, const int n) {  
    if (n == 0) {  
        return 1;  
    } else if (n == 1) {  
        return x;  
    }  
  
    int temp = power(x, n / 2);  
    return temp * temp;  
}
```

Was ist falsch?

- `power(2, 3)` gibt 4 statt 8:
3 / 2 ist 1
- Richtig nur, wenn n eine Zweierpotenz ist. Dafür sehr schnell:



“Power Function” — Lösung

```
// POST: returns x^n
int power(const int x, const int n) {
    if (n == 0) {
        return 1;
    } else if (n == 1) {
        return x;
    } else if (n % 2 == 0) {          // n = 2m
        int temp = power(x, n / 2);
        return temp * temp;          // x^m * x^m
    } else {                         // n odd
        return x * power(x, n - 1); // n - 1 is even
    }
}
```

- Gerades n : halbieren
- Ungerades n : einen Schritt wie vorher. Danach ist $n - 1$ gerade
- Höchstens $2\log_2(n + 1) - 1$ Aufrufe: logarithmisch statt linear

Wie viele Aufrufe?

vorher, x^7 : $n = 7 \rightarrow n = 6 \rightarrow n = 5 \rightarrow n = 4 \rightarrow n = 3 \rightarrow n = 2 \rightarrow n = 1$

neu, x^7 : $n = 7 \rightarrow n = 6 \rightarrow n = 3 \rightarrow n = 2 \rightarrow n = 1$

neu, x^{15} : $n = 15 \rightarrow n = 14 \rightarrow n = 7 \rightarrow n = 6 \rightarrow n = 3 \rightarrow n = 2 \rightarrow n = 1$

Für x^{1000} : 15 Aufrufe statt 1000.

Genauer hingeschaut: Wozu temp ?

Warum nicht einfach so?

```
} else if (n % 2 == 0) {  
    return power(x, n / 2) * power(x, n / 2);  
}
```

- Das Resultat stimmt, aber jeder Aufruf mit geradem n ruft sich jetzt *zweimal* auf
- Für x^{1000} : 1511 Aufrufe. Das ist sogar mehr als die 1000 der einfachen Variante!

💡 Rechnet ein Resultat einmal aus und speichert es, statt dieselbe Rekursion zweimal zu starten. Genau das macht "Compute Series" so langsam.

Fragen/Unklarheiten?

7. Extra: Fortgeschrittene Rekursion

Die Türme von Hanoi

An dieser Aufgabe zu verzweifeln, gehört für Programmieranfänger:innen fast schon dazu.

Ohne Rekursion ist sie berüchtigt schwierig. Mit Rekursion sind es drei Zeilen.

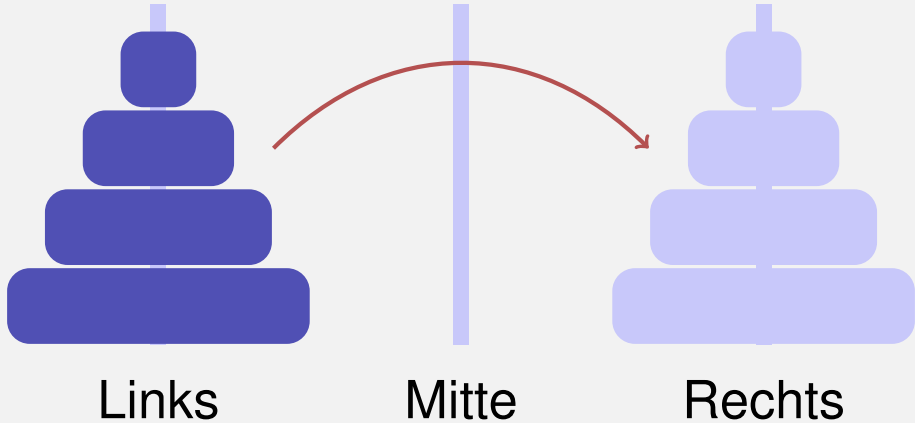
Everyone: it's a game for kids



Programmers:



Experiment: Die Türme von Hanoi



Die Türme von Hanoi

Angenommen, wir wissen schon, wie man 3 Scheiben von einem Stab auf einen anderen verschiebt.

Wie verschieben wir dann 4 Scheiben?

Die Türme von Hanoi - Rekursiver Lösungsansatz



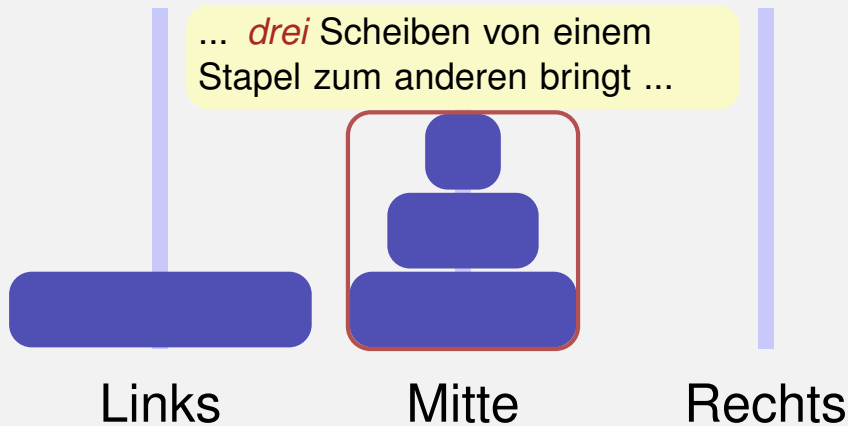
Mal *angenommen*, wir
wüssten wie man ...

Links

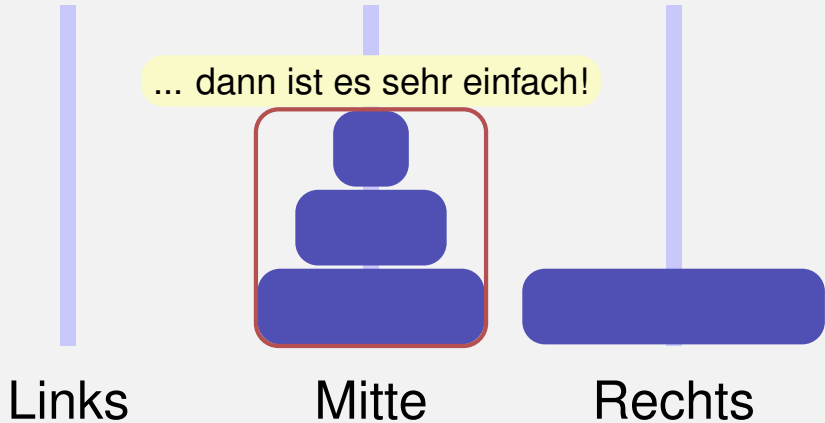
Mitte

Rechts

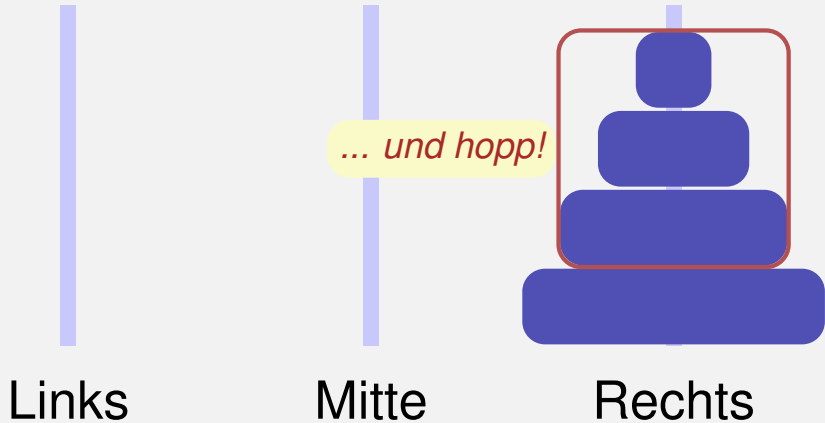
Die Türme von Hanoi - Rekursiver Lösungsansatz



Die Türme von Hanoi - Rekursiver Lösungsansatz



Die Türme von Hanoi - Rekursiver Lösungsansatz



Die Türme von Hanoi

- Wie verschieben wir n Scheiben ($n \geq 4$), wenn wir wissen, wie man $n - 1$ Scheiben verschiebt?
Genau gleich wie für $n = 4$.
- Für welche n funktioniert das? Was ist unser Basisfall?
 $n = 1$: Die eine Scheibe verschieben wir einfach direkt ans Ziel.

Die Türme von Hanoi - Rekursiver Lösungsansatz



Aufgabe “The Towers of Hanoi”

Aufgabe

Schreibt ein Programm, das ausgibt, wie man n Scheiben vom linken auf den rechten Stab verschiebt.

Öffnet “The Towers of Hanoi” auf **code expert** 10 Minuten zu zweit programmieren, dann diktiert ihr mir die Lösung.

💡 Gebt den Stäben in jedem Aufruf neue Rollen: Start, Hilfsstab, Ziel.

Eingabe

2

Ausgabe

```
left --> middle  
left --> right  
middle --> right
```

Die Türme von Hanoi - Code

`move(n, src, aux, dst)` $\Rightarrow$

- 1 Bewege die obersten $n - 1$ Scheiben von *src* nach *aux* mit Hilfsstapel *dst*:
`move( $n - 1$ , src, dst, aux);`
- 2 Bewege 1 Scheibe von *src* nach *dst*
`move(1, src, aux, dst);`
- 3 Bewege die obersten $n - 1$ Scheiben von *aux* nach *dst* mit Hilfsstapel *src*:
`move( $n - 1$ , aux, src, dst);`

Die Türme von Hanoi - Code

```
void move(int n, const string &src, const string &aux, const string &dst){
    if (n == 1) {
        // base case ('move' the disc)
        std::cout << src << "  --> " << dst << std::endl;
    } else {
        // recursive case
        move(n-1, src, dst, aux);
        move(1, src, aux, dst);
        move(n-1, aux, src, dst);
    }
}

int main() {
    move(4, "left ", "middle", "right ");
    return 0;
}
```

Die Türme von Hanoi - Code Alternative

```
void move(int n, const string &src, const string &aux, const string &dst){  
    // base case  
    if (n == 0) return;  
  
    // recursive case  
    move(n-1, src, dst, aux);  
    std::cout << src << "  --> " << dst << "\n";  
    move(n-1, aux, src, dst);  
}
```

```
int main() {  
    move(4, "left ", "middle", "right ");  
    return 0;  
}
```

Genauer hingeschaut

- Wie viele Züge braucht es für n Scheiben? $2^n - 1$
1, 3, 7, 15, ..., für $n = 10$ schon 1023
- Was macht die Lösung mit Basisfall $n = 1$ bei der Eingabe 0?
Sie ruft sich mit $-1, -2, \dots$ endlos auf, bis der Stack voll ist:
Segmentation fault. Also PRE: $n \geq 1$, oder der Basisfall $n = 0$ der Alternative.

Videoempfehlungen

Versucht vor allem, den *Recursive Leap of Faith* nachzuvollziehen. Er entspricht der Induktionsannahme in einem Induktionsbeweis.

Videos zum Thema Rekursion

- [5 Simple Steps for Solving Any Recursive Problem](#)
- [Towers of Hanoi: A Complete Recursive Visualization](#)

Fragen/Unklarheiten?

8. Outro

Allgemeine Fragen?

Gibt es noch offene Fragen zu dem, was wir bisher in den Übungsstunden gemacht haben?

Bis nächste Woche!