Algorithmen

Inf-Einf-B · Vorlesung 3, zweiter Teil

Prof. Dr. Dominik Herrmann

Einführung in die Informatik (Inf-Einf-B)

Otto-Friedrich-Universität Bamberg

WiSe 2026/27

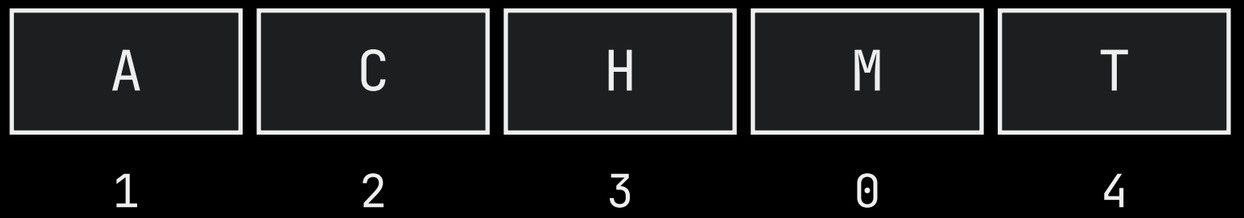
Pause · 20 Minuten

Aus dem letzten Jahrgang

2 von 10 wünschten sich eine Lerngruppe und hatten keine.
Ins Tutorium ging etwa 1 von 10.

Probieren Sie erst selbst. Mit dem, was nicht läuft, ins Tutorium.

Wie sortiert der Rechner MACHT?



Wie sortiert der Rechner?

Heute

Selection Sort	an den Kisten
Bubble Sort	Nachbarn tauschen
Rekursion	eine Funktion ruft sich selbst auf
Merge Sort	halbieren und zusammenführen

Warum sortieren

20	500	10	5	100	1	50

↓ Sortieren ↓

1	5	10	20	50	100	500

Danach findet die binäre Suche die 50 in 3 Schritten.

Selection Sort

Was muss ich mir mindestens merken?

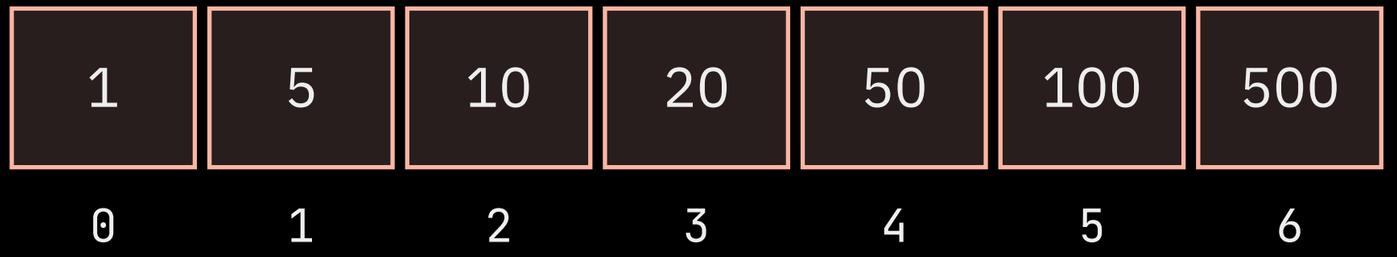
Ich suche die kleinste Kiste und tausche sie nach vorn.

A den kleinsten Schein

B wo er liegt

C beides

B: Die Position reicht. Den Schein kann ich jederzeit wieder nachsehen.



6 + 5 + 4 + 3 + 2 + 1 = 21 Vergleiche



Den letzten Durchgang sparen wir: i nur bis n-2

Selection Sort als Pseudocode

```
Für i von 0 bis n-2
    min = i
    Für j von i+1 bis n-1
        Wenn kisten[j] < kisten[min]
            min = j
    Tausche kisten[i] und kisten[min]
```

Vergleiche: $(n - 1) + (n - 2) + \ldots + 1 = n(n - 1)/2$

Bei 7 Kisten: $7 \cdot 6 / 2 = 21$.

Laufzeit $O(n^2)$, quadratisch.

Und wenn die Kisten schon sortiert sind?

Wie viele Vergleiche braucht Selection Sort bei 1, 5, 10, 20, 50, 100, 500?

A 6, einen je Kiste

B 21, genauso viele

C keinen

B: Er sucht trotzdem jedes Mal die kleinste.
Auch im besten Fall n^2, geschrieben $\Omega(n^2)$.

selection.c

```c
int numbers[] = {20, 500, 10, 5, 100, 1, 50};
int n = 7;

for (int i = 0; i < n - 1; i++)
{
    int min = find_index_of_smallest(numbers, i, n);

    // Kleinste Zahl ab Position i nach vorne tauschen
    int tmp = numbers[i];
    numbers[i] = numbers[min];
    numbers[min] = tmp;
}
```

`i < n - 1` heißt: `i` läuft bis `n - 2`, wie im Pseudocode.

Die kleinste ab `start` finden


```c
int find_index_of_smallest(int arr[], int start, int length)
{
    int min = start;
    for (int j = start + 1; j < length; j++)
    {
        if (arr[j] < arr[min])
        {
            min = j;
        }
    }
    return min;
}
```

```
$ ./selection
1 5 10 20 50 100 500
```

Was gibt `selection_swap` aus?

```
swap(numbers[i], numbers[min]);
```

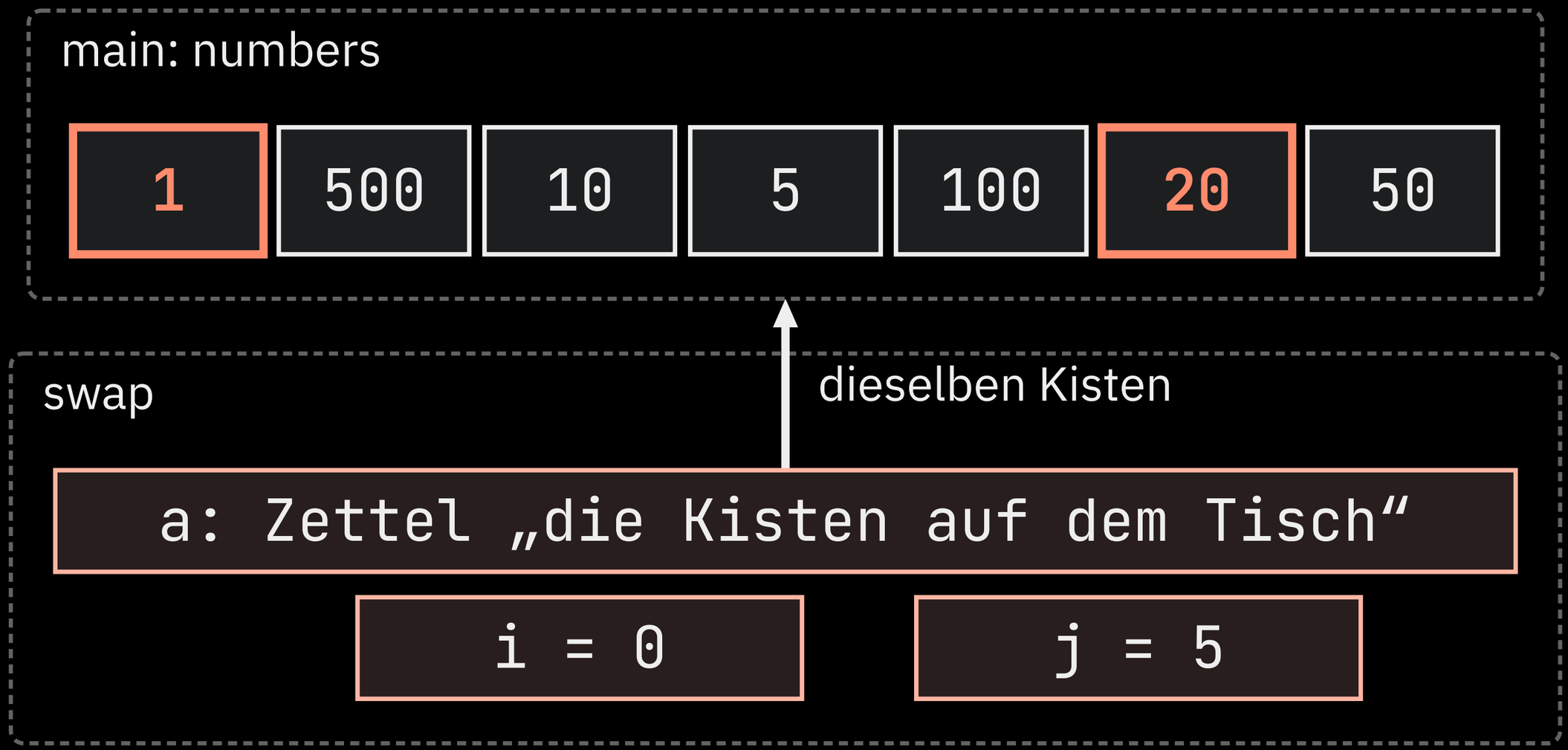
```
void swap(int a, int b)
{
    int tmp = a;
    a = b;
    b = tmp;
}
```

A sortiert · B unverändert · C Fehler beim Übersetzen

```
$ ./selection_swap
20 500 10 5 100 1 50
```



Die Kisten von main: unverändert



Die Kisten von main: getauscht

Einzelne Werte werden kopiert.

Bei einem Array arbeitet
die Funktion am Original.

Warum das so ist:
nächste Woche, Speicher.

Vor der Pause:
`next_block(start, n)` bekam
eine Kopie von `start`. Deshalb
hörte die Ausgabe nicht auf.

`swap` mit Array und Positionen

```
swap(numbers, i, min);
```

```
void swap(int a[], int i, int j)
{
    int tmp = a[i];
    a[i] = a[j];
    a[j] = tmp;
}
```

```
$ ./selection_swap2
1 5 10 20 50 100 500
```

Bubble Sort



Nach Durchgang 2: die 100 davor

`numbers[j + 1]`: wie weit darf `j` laufen?

```
Wiederhole n-1 Mal
    Für j von 0 bis ?
        Wenn numbers[j] > numbers[j + 1]
            Tausche beide
    Kein Tausch in diesem Durchgang? Dann fertig
```

A `j < n` · B `j < n - 1` · C `j < n - 2`

B: `j` höchstens n – 2. Bei `j < n` läse `numbers[j + 1]` hinter dem Ende.



Umgekehrt: jeder Vergleich ein Tausch, n – 1 Durchgänge → O(n²)



Sortiert: ein Durchgang, n – 1 Vergleiche, kein Tausch → fertig

Rekursion

Dasselbe auf der Hälfte noch einmal

```
Wenn keine Kisten übrig sind
    Gib false zurück
Wenn 50 in der mittleren Kiste ist
    Gib true zurück
Sonst wenn 50 < mittlere Kiste
    Durchsuche die linke Hälfte
Sonst wenn 50 > mittlere Kiste
    Durchsuche die rechte Hälfte
```

Rekursion: dasselbe Verfahren auf einem kleineren Problem.

Basisfall: Hier hört es auf, ohne sich selbst aufzurufen.

Eine Pyramide mit Schleifen


```c
void draw(int n)
{
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < i + 1; j++)
        {
            printf("#");
        }
        printf("\n");
    }
}
```

Höhe 3

#

##

###

####

+ eine Zeile mit 4 Steinen

draw ruft draw auf


```c
void draw(int n)
{
    draw(n - 1);

    for (int i = 0; i < n; i++)
    {
        printf("#");
    }
    printf("\n");
}
```

```
$ make draw
draw.c:13:1: error: all paths through this function
will call itself [-Werror,-Winfinite-recursion]
```

Der Basisfall


```
void draw(int n)
{
    if (n <= 0)
    {
        return;
    }
    draw(n - 1);
    for (int i = 0; i < n; i++)
    {
        printf("#");
    }
    printf("\n");
}
```

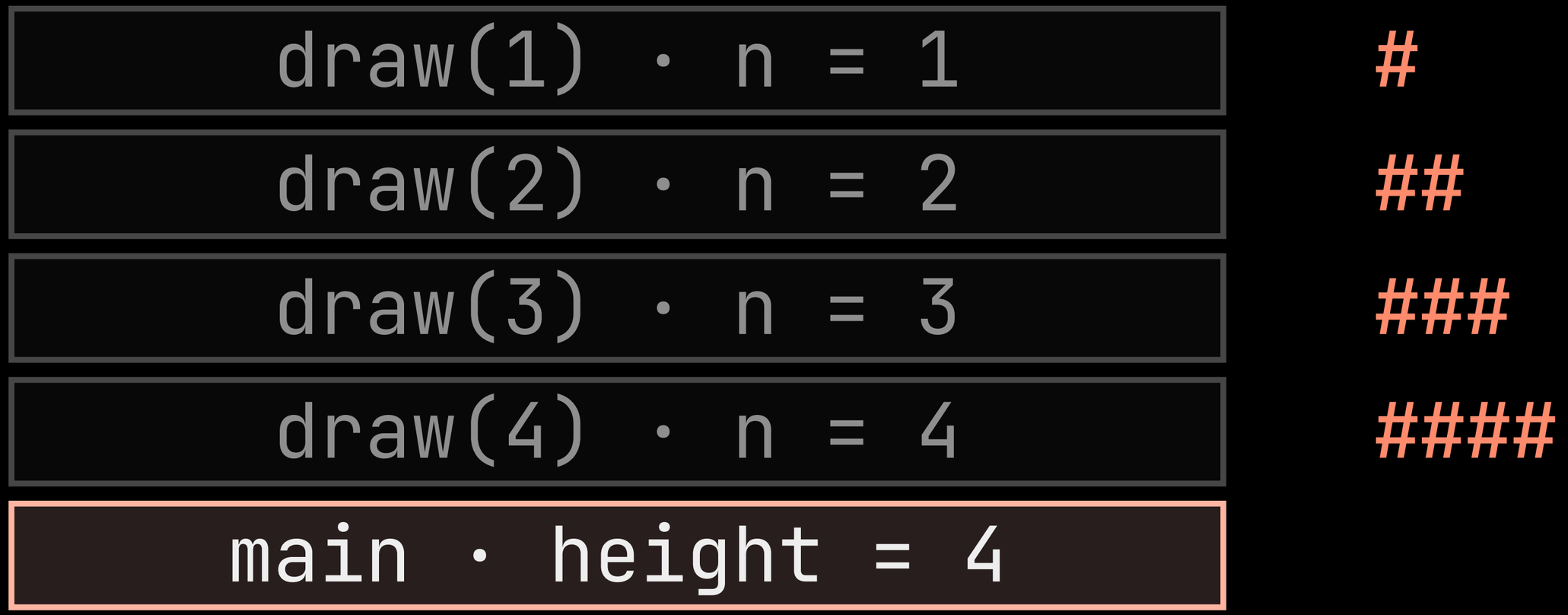
Die Pyramide steht, ohne Schleife über die Zeilen.

Eine richtig große Pyramide

Was passiert bei Höhe 1 000 000?

```
$ ./draw
Height: 1000000
Segmentation fault
```

Kein einziger Stein.


draw(1) · n = 1
draw(2) · n = 2
draw(3) · n = 3
draw(4) · n = 4
main · height = 4
#
##
###
####

Eine Million Kartons

Bei Höhe 1 000 000 stehen **eine Million Kartons** übereinander, bevor der erste Stein gedruckt wird.

Dafür reicht der Platz nicht:
`Segmentation fault`.

Die Zeile vor dem Aufruf

```c
void draw(int n)
{
    if (n <= 0)
    {
        return;
    }
    for (int i = 0; i < n; i++)
    {
        printf("#");
    }
    printf("\n");
    draw(n - 1);
}
```

A gleich · B umgekehrt · C nur eine Zeile

B: Die Pyramide steht auf dem Kopf. Jeder Aufruf druckt beim Aufbau.

Jetzt Sie: Fakultät

3 Minuten, auf Papier, Laptop zu. Wählen Sie die Stufe, die Sie schaffen.

$n! = n \cdot (n - 1)!$ und $1! = 1$ (Beispiel: $4 \cdot 3 \cdot 2 \cdot 1 = 24$)

Stufe 1 Aufrufe von `fakultaet(4)` als Stapel, mit Rückgabewerten

Stufe 2 Lücken unten füllen

Stufe 3 Mit Schleife, `int i;` davor. Was steht danach in `i`?

```
int fakultaet(int n)
{
    if (n <= ___)
    {
        return ___;
    }
    return n * ____________;
}
```

Auflösung: Fakultät rekursiv

Stufe 1 `fakultaet(4)` ruft `fakultaet(3)`, `fakultaet(2)`, `fakultaet(1)` auf. Zurück kommen 1, 2, 6, **24**.

```
int fakultaet(int n)
{
    // Basisfall: 1! = 1 (und 0! = 1)
    if (n <= 1)
    {
        return 1;
    }

    // n! = n * (n - 1)!
    return n * fakultaet(n - 1);
}
```

Auflösung: Fakultät mit Schleife

```c
int fakultaet(int n)
{
    int ergebnis = 1;
    int i;
    for (i = 1; i <= n; i++)
    {
        ergebnis = ergebnis * i;
    }

    // Hier steht in i der Wert n + 1
    return ergebnis;
}
```

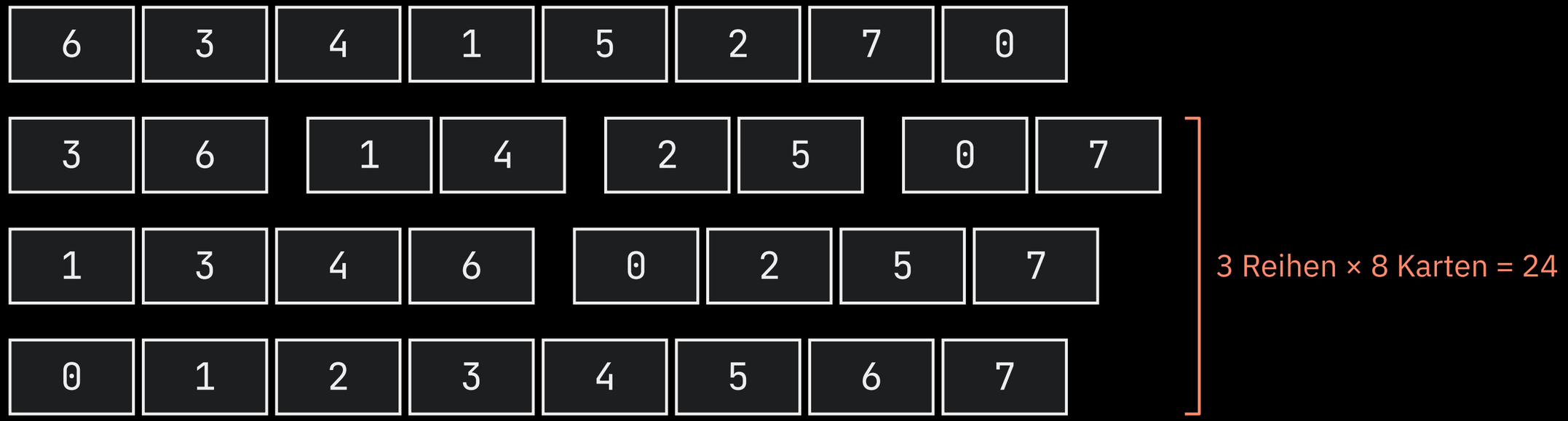
`i` ist `n + 1`. Die Schleife endet erst, wenn `i <= n` nicht mehr gilt.

Merge Sort

Merge Sort als Pseudocode


```
Wenn nur eine Zahl übrig ist
    Fertig
Sortiere die linke Hälfte
Sortiere die rechte Hälfte
Füge die beiden sortierten Hälften zusammen
```


links sortiert
rechts sortiert
1
3
4
6
0
2
5
7
0
1
2
3
4
5
6
7


6 3 4 1 5 2 7 0
3 6 1 4 2 5 0 7
1 3 4 6 0 2 5 7
0 1 2 3 4 5 6 7
3 Reihen × 8 Karten = 24

Laufzeit von Merge Sort

$\log_2$ n Reihen	mit je **n Karten**: O(n log n)
Bester Fall	genauso: Ω(n log n)
Preis	ein zweites Array

n = 1000: Wie viel schneller?

Selection Sort braucht etwa n^2 Schritte, Merge Sort etwa $n \cdot \log_2 n$. Dabei ist $\log_2 1000$ ungefähr 10.

A 2-mal

B 10-mal

C 100-mal

C: 1 000 000 gegen etwa 10 000 Schritte, hundertmal weniger.

Drei Verfahren im Vergleich

Gemessen

Aufruf	Zeit
`time ./selection 60000`	1,1 s
`time ./selection 120000`	4,3 s
`time ./merge 120000`	0,01 s

Verfahren	Worst Case	Best Case
Selection Sort	$O(n^2)$	$\Omega(n^2)$
Bubble Sort	$O(n^2)$	$\Omega(n)$
Merge Sort	$O(n \log n)$	$\Omega(n \log n)$

Übung 4: drei Programme, drei Verfahren.
Sie finden heraus, welches welches ist.

Zusammenfassung

Warum ist ein Array anders?

`swap` tauscht, wenn es das Array bekommt. Wie kommt eine Funktion an ein einzelnes `int` von `main`?

Und wo liegen das zweite Array von Merge Sort und die Million Kartons?

Dies war Inf-Einf-B.

Übung 4 startet heute: Sortieren, Mehrheitswahl, Stichwahl. Abgabe bis Montag, 23. November.

Übung 3 läuft: Cäsar, Abgabe bis Montag, 16. November.

Shorts 3.3 bis 3.6: Bubble Sort, Selection Sort, Rekursion, Merge Sort.