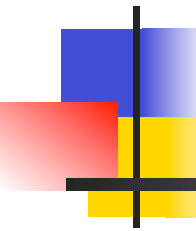
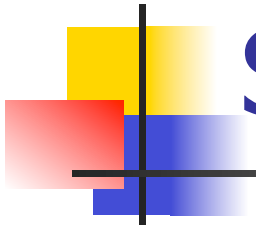


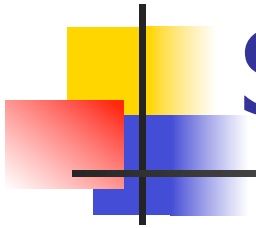
Rekursion: Anwendungen

Sortieren mit Merge-Sort, Fraktale zeichnen mit Lindenmayer-Systemen



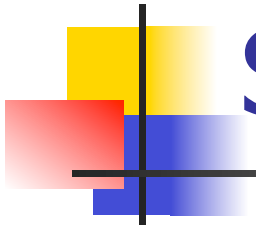
Sortieren

- Wie schnell kann man *n* Zahlen (oder Wörter,...) aufsteigend sortieren?



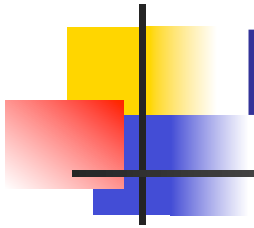
Sortieren

- Wie schnell kann man *n* Zahlen (oder Wörter,...) aufsteigend sortieren?
- Wir haben in den Übungen bereits sortiert, uns aber keine Gedanken über die Effizienz gemacht



Sortieren

- Wie schnell kann man *n* Zahlen (oder Wörter,...) aufsteigend sortieren?
- Wir haben in den Übungen bereits sortiert, uns aber keine Gedanken über die Effizienz gemacht
- Das holen wir jetzt nach, denn Sortieren ist eine der grundlegenden Operationen in der Informatik

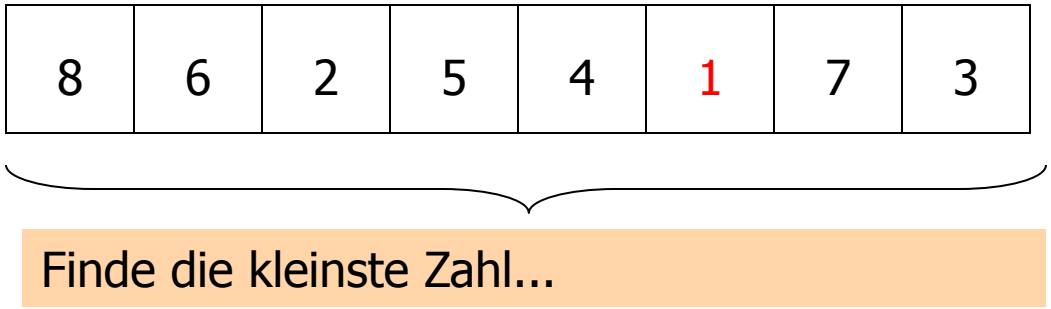


Minimum-sort

- ist ein sehr einfaches Sortierverfahren
- haben einige wahrscheinlich in den Übungen implementiert

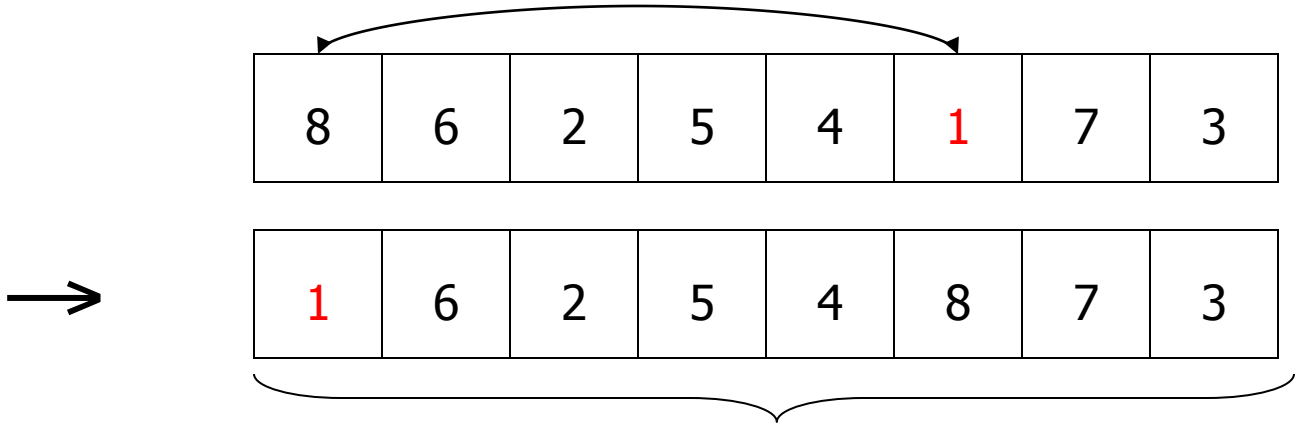
Minimum-sort

- ist ein sehr einfaches Sortierverfahren
- haben einige wahrscheinlich in den Übungen implementiert



Minimum-sort

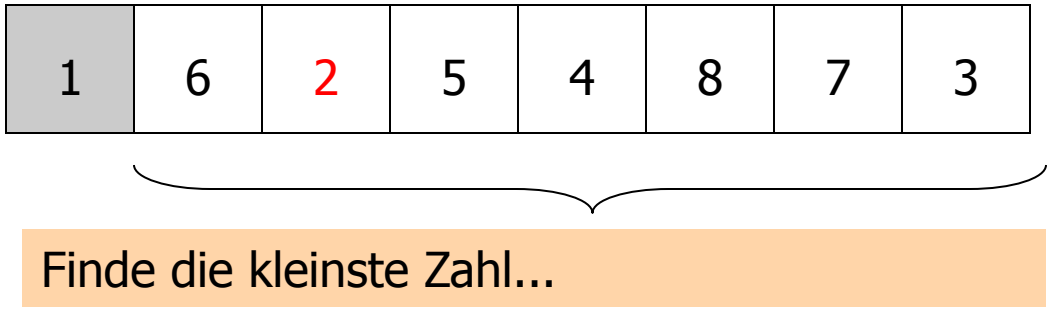
- ist ein sehr einfaches Sortierverfahren
- haben einige wahrscheinlich in den Übungen implementiert



...und vertausche sie mit der ersten Zahl.

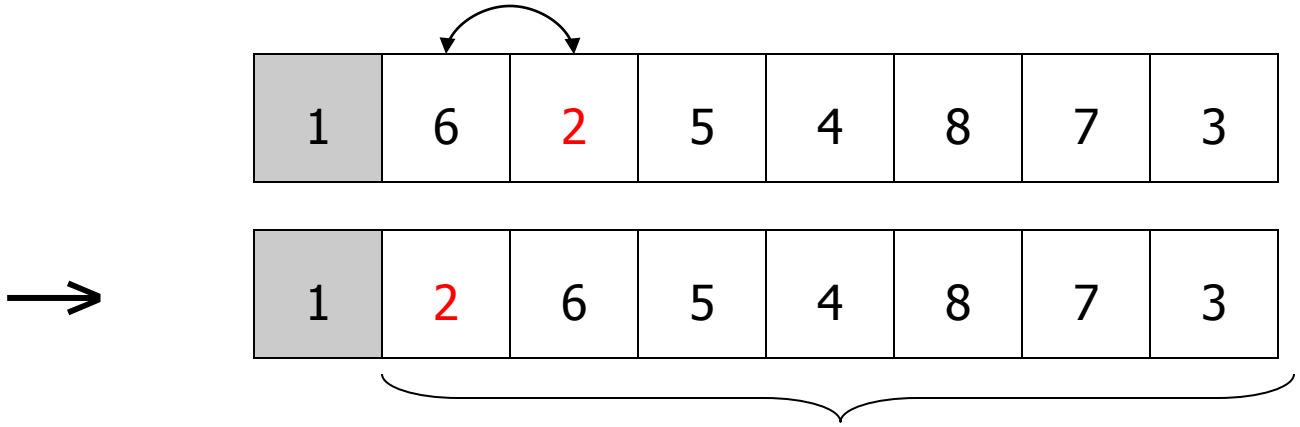
Minimum-sort

- ist ein sehr einfaches Sortierverfahren
- haben einige wahrscheinlich in den Übungen implementiert



Minimum-sort

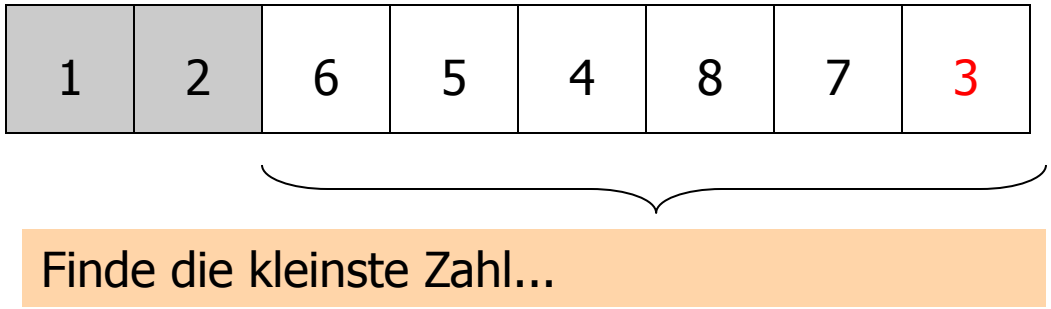
- ist ein sehr einfaches Sortierverfahren
- haben einige wahrscheinlich in den Übungen implementiert



...und vertausche sie mit der ersten Zahl.

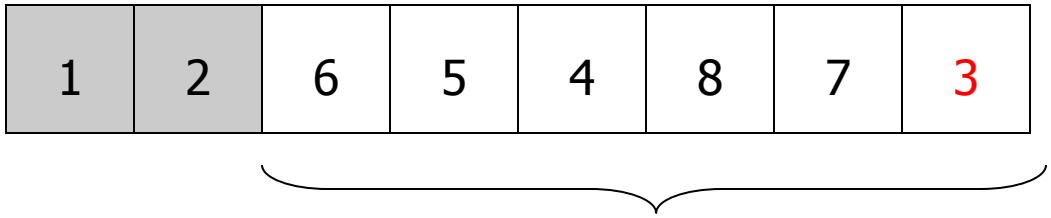
Minimum-sort

- ist ein sehr einfaches Sortierverfahren
- haben einige wahrscheinlich in den Übungen implementiert



Minimum-sort

- ist ein sehr einfaches Sortierverfahren
- haben einige wahrscheinlich in den Übungen implementiert

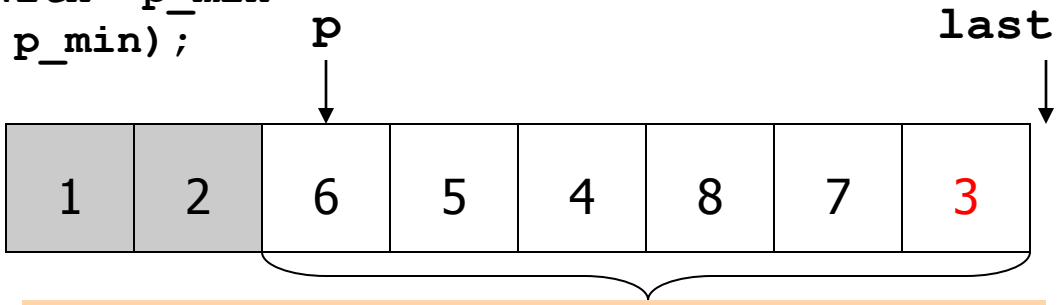


usw....

```
typedef std::vector<int >::iterator Vit;
```

Minimum-sort

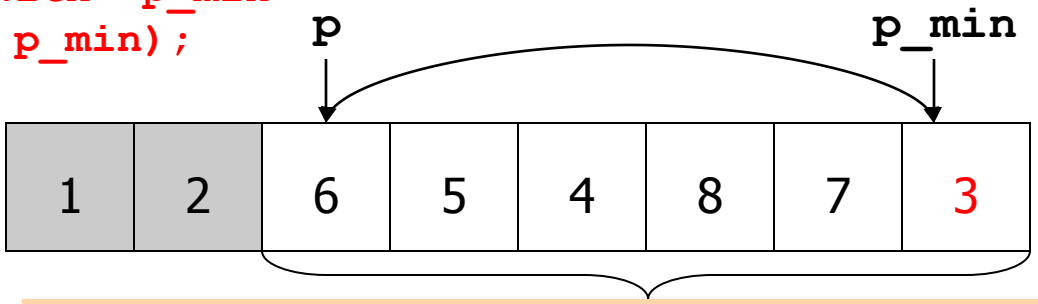
```
// PRE: [first, last) is a valid range
// POST: the elements *p, p in [first, last) are in
//       ascending order
void minimum_sort (Vit first, Vit last)
{
  for (Vit p = first; p != last; ++p) {
    // find minimum in nonempty range described by [p, last)
    Vit p_min = p; // iterator pointing to current minimum
    Vit q = p;     // iterator pointing to current element
    while (++q != last)
      if (*q < *p_min) p_min = q;
    // interchange *p with *p_min
    std::iter_swap (p, p_min);
  }
}
```



Finde die kleinste Zahl...

Minimum-sort

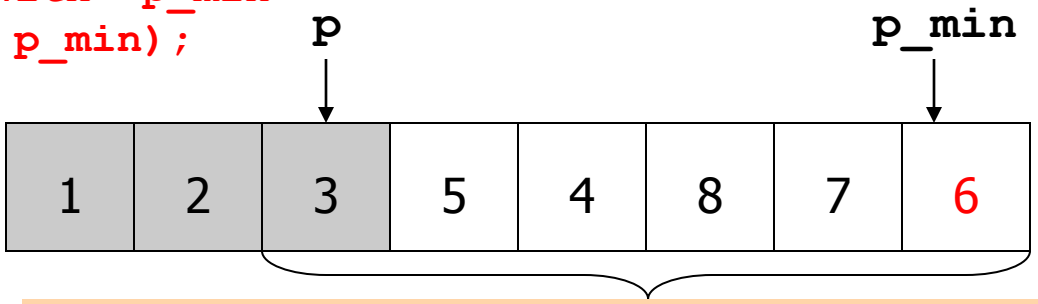
```
// PRE: [first, last) is a valid range
// POST: the elements *p, p in [first, last) are in
//       ascending order
void minimum_sort (Vit first, Vit last)
{
  for (Vit p = first; p != last; ++p) {
    // find minimum in nonempty range described by [p, last)
    Vit p_min = p; // iterator pointing to current minimum
    Vit q = p;     // iterator pointing to current element
    while (++q != last)
      if (*q < *p_min) p_min = q;
    // interchange *p with *p_min
    std::iter_swap (p, p_min);
  }
}
```



..und vertausche sie mit der ersten Zahl

Minimum-sort

```
// PRE: [first, last) is a valid range
// POST: the elements *p, p in [first, last) are in
//       ascending order
void minimum_sort (Vit first, Vit last)
{
  for (Vit p = first; p != last; ++p) {
    // find minimum in nonempty range described by [p, last)
    Vit p_min = p; // iterator pointing to current minimum
    Vit q = p;     // iterator pointing to current element
    while (++q != last)
      if (*q < *p_min) p_min = q;
    // interchange *p with *p_min
    std::iter_swap (p, p_min);
  }
}
```

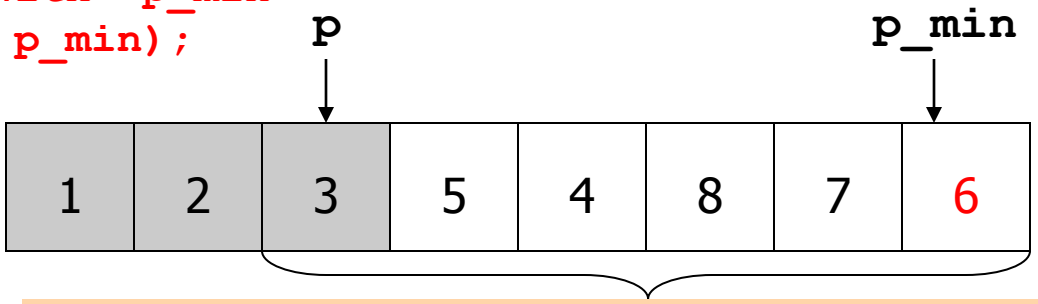


..und vertausche sie mit der ersten Zahl

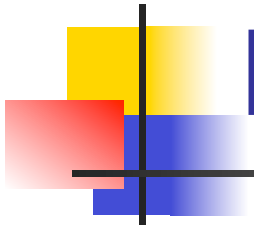
Minimum-sort

```
// PRE: [first, last) is a valid range
// POST: the elements *p, p in [first, last) are in
//       ascending order
void minimum_sort (Vit first, Vit last)
{
  for (Vit p = first; p != last; ++p) {
    // find minimum in nonempty range described by [p, last)
    Vit p_min = p; // iterator pointing to current minimum
    Vit q = p;     // iterator pointing to current element
    while (++q != last)
      if (*q < *p_min) p_min = q;
    // interchange *p with *p_min
    std::iter_swap (p, p_min);
  }
}
```

```
#include<algorithm>
```



..und vertausche sie mit der ersten Zahl



Minimum-sort: Laufzeit

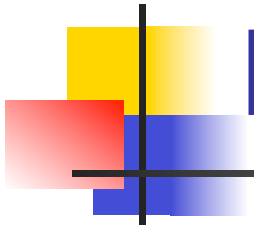
- hängt von der Plattform ab.

Minimum-sort: Laufzeit

- hängt von der Plattform ab
- Trotzdem gibt es ein plattformunabhängiges Laufzeitmass:

Anzahl der Vergleiche `*q < *p_min`

Minimum-sort: Laufzeit



- hängt von der Plattform ab
- Trotzdem gibt es ein plattformunabhängiges Laufzeitmass:

  Anzahl der Vergleiche `*q < *p_min`

- Anzahl anderer Operationen ist wesentlich kleiner oder proportional dazu

Minimum-sort: Laufzeit

```
// PRE: [first, last) is a valid range
// POST: the elements *p, p in [first, last) are in
//       ascending order
void minimum_sort (Vit first, Vit last)
{
  for (Vit p = first; p != last; ++p) {
    // find minimum in nonempty range described by [p, last)
    Vit p_min = p; // iterator pointing to current minimum
    Vit q = p;     // iterator pointing to current element
    while (++q != last)
      if (*q < *p_min) p_min = q;
    // interchange *p with *p_min
    std::iter_swap (p, p_min);
  }
}
```

n Elemente

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

Minimum-sort: Laufzeit

```
// PRE: [first, last) is a valid range
// POST: the elements *p, p in [first, last) are in
//       ascending order
void minimum_sort (Vit first, Vit last)
{
  for (Vit p = first; p != last; ++p) {
    // find minimum in nonempty range described by [p, last)
    Vit p_min = p; // iterator pointing to current minimum
    Vit  q = p;    // iterator pointing to current element
    while (++q != last)
      if (*q < *p_min) p_min = q;
    // interchange *p with *p_min
    std::iter_swap (p, p_min);
  }
}
```

n Elemente

jeweils $\leq n$ Operationen (wesentlich kleiner)

Minimum-sort: Laufzeit

```
// PRE: [first, last) is a valid range
// POST: the elements *p, p in [first, last) are in
//       ascending order
void minimum_sort (Vit first, Vit last)
{
  for (Vit p = first; p != last; ++p) {
    // find minimum in nonempty range described by [p, last)
    Vit p_min = p; // iterator pointing to current minimum
    Vit q = p;     // iterator pointing to current element
    while (++q != last)
      if (*q < *p_min) p_min = q;
    // interchange *p with *p_min
    std::iter_swap (p, p_min);
  }
}
```

n Elemente

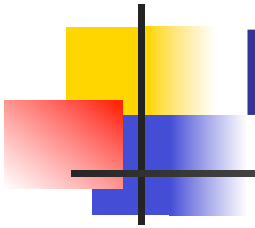
höchstens $n\,(n-1)\,/\,2$ Operationen (proportional)

Minimum-sort: Laufzeit

```
// PRE: [first, last) is a valid range
// POST: the elements *p, p in [first, last) are in
//       ascending order
void minimum_sort (Vit first, Vit last)
{
  for (Vit p = first; p != last; ++p) {
    // find minimum in nonempty range described by [p, last)
    Vit p_min = p; // iterator pointing to current minimum
    Vit q = p;     // iterator pointing to current element
    while (++q != last)
      if (*q < *p_min) p_min = q;
    // interchange *p with *p_min
    std::iter_swap (p, p_min);
  }
}
```

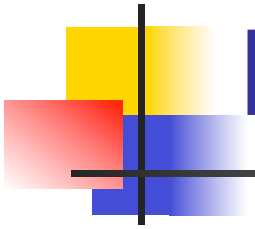
n Elemente

$n\,(n-1)\,/\,2 + n$ Operationen (proportional)



Minimum-sort: Laufzeit

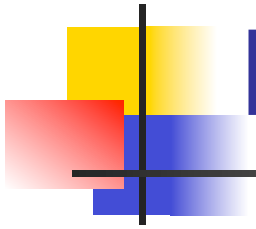
- Auf “jeder” Plattform: Gesamtlaufzeit ist proportional zur Zeit, die mit den Vergleichen `*q < *p_min` verbracht wird



Minimum-sort: Laufzeit

- Auf "jeder" Plattform: Gesamtlaufzeit ist proportional zur Zeit, die mit den Vergleichen `*q < *p_min` verbracht wird
- Diese wiederum ist proportional zur Anzahl $n\,(n-1)\,/\,2$ dieser Vergleiche

Minimum-sort: Laufzeit



- Auf "jeder" Plattform: Gesamtlaufzeit ist proportional zur Zeit, die mit den Vergleichen `*q < *p_min` verbracht wird
- Diese wiederum ist proportional zur Anzahl $n\,(n-1)\,/\,2$ dieser Vergleiche
- Anzahl der Vergleiche ist deshalb ein gutes Mass für die Gesamtlaufzeit!

Minimum-sort: Laufzeit (alter Mac) auf zufälliger Eingabe

n	100,000	200,000	400,000	800,000	1,600,000
Gcomp (Milliarden Vergleiche)	5	20	80	320	1280
Laufzeit in min : s	0:15	1:05	4:26	15:39	64:22
Laufzeit / GComp (s)	3.0	3.25	3.325	2.93	3.01

Minimum-sort: Laufzeit (alter Mac) auf zufälliger Eingabe

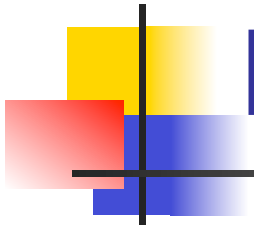
n	100,000	200,000	400,000	800,000	1,600,000
Gcomp (Milliarden Vergleiche)	5	20	80	320	1280
Laufzeit in min : s	0:15	1:05	4:26	15:39	64:22
Laufzeit / GComp (s)	3.0	3.25	3.325	2.93	3.01

Ungefähr konstant! Anzahl Vergleiche *ist* ein gutes Laufzeitmass!

Minimum-sort: Laufzeit (alter Mac) auf zufälliger Eingabe

n	100,000	200,000	400,000	800,000	1,600,000
Gcomp (Milliarden Vergleiche)	5	20	80	320	1280
Laufzeit in min : s	0:15	1:05	4:26	15:39	64:22
Laufzeit / GComp (s)	3.0	3.25	3.325	2.93	3.01

D.h.: für 10,000,000 Zahlen bräuchte Minimum-sort ca. 2 Tage!



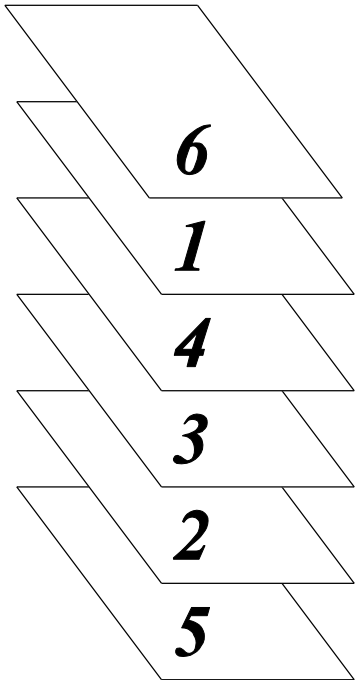
Merge-sort

- ein schnelleres (und rekursives) Sortierverfahren
- folgt dem Paradigma “Teile und Herrsche” (Divide & Conquer)

Teile das Problem in kleinere Teilprobleme des gleichen Typs auf, löse diese rekursiv, und berechne die Gesamtlösung aus den Lösungen der Teilprobleme

Merge-sort

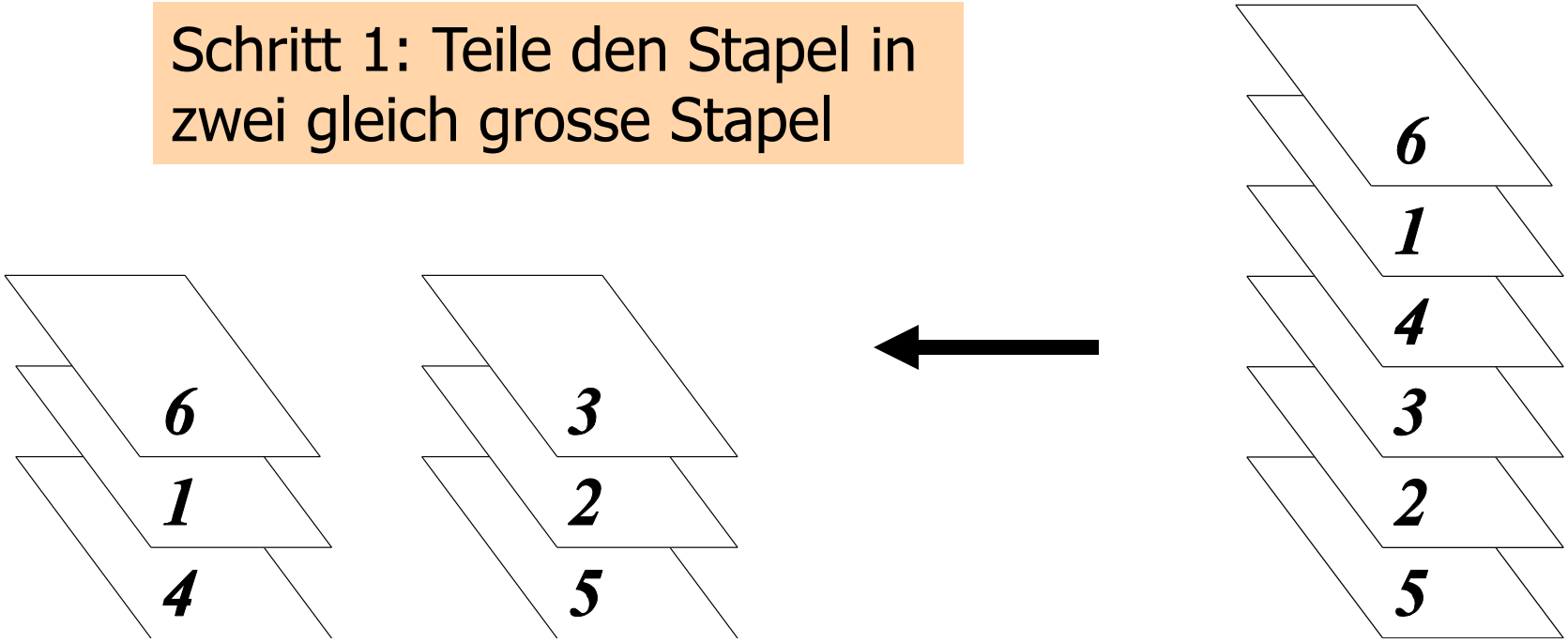
- Analogie: Sortieren eines Kartenstapels
- Karten sind mit Zahlen beschriftet



Merge-sort: Teile

- Analogie: Sortieren eines Kartenstapels
- Karten sind mit Zahlen beschriftet

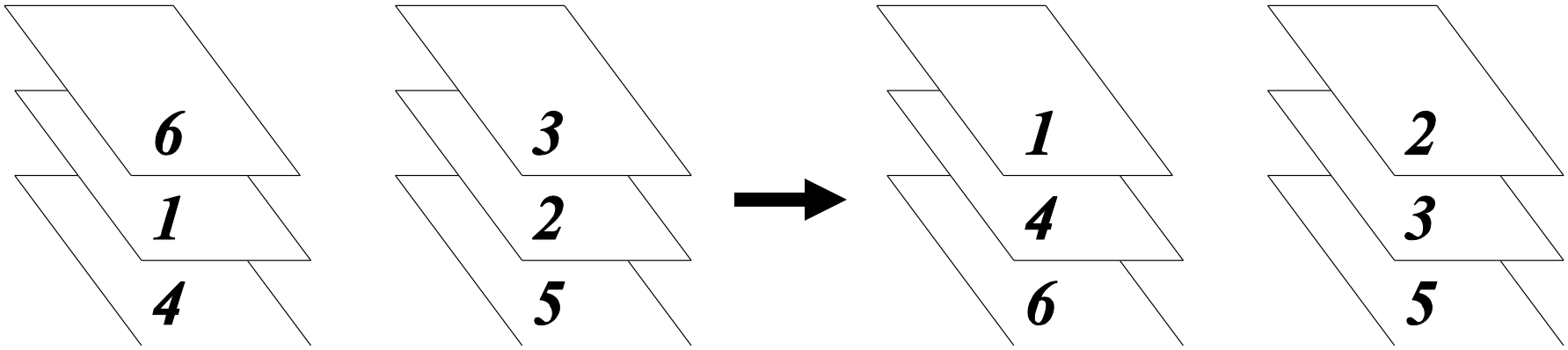
Schritt 1: Teile den Stapel in zwei gleich grosse Stapel



Merge-sort: Teile

- Analogie: Sortieren eines Kartenstapels
- Karten sind mit Zahlen beschriftet

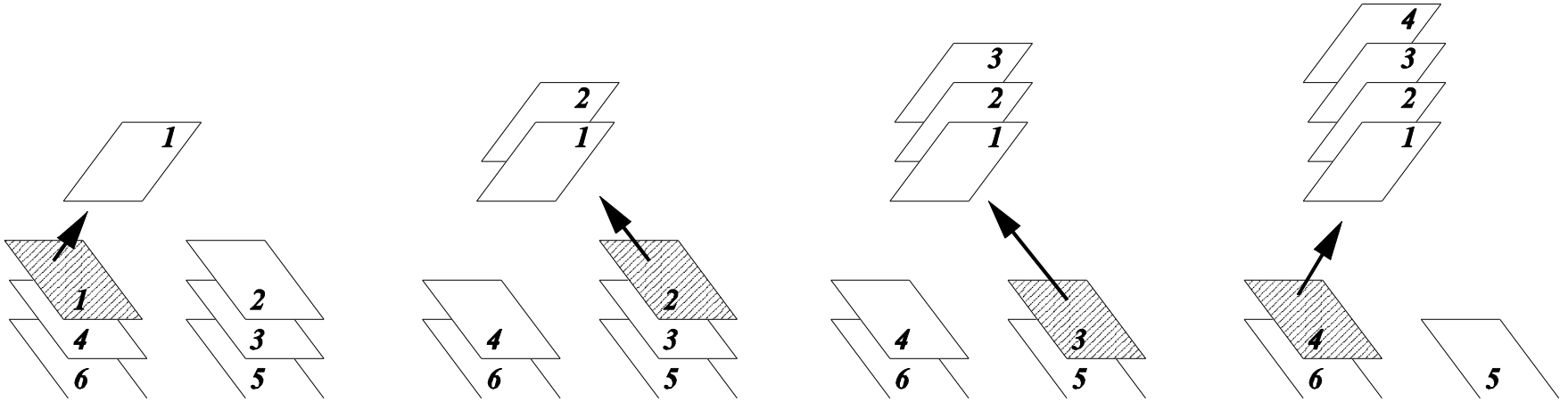
Schritt 2: Sortiere beide Stapel getrennt

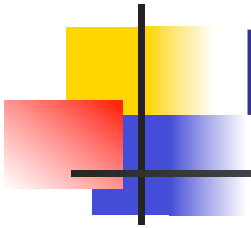


Merge-sort: Herrsche

- Analogie: Sortieren eines Kartenstapels
- Karten sind mit Zahlen beschriftet

Schritt 3: Mische die beiden sortierten Stapel zu einem sortierten Stapel: lege dazu jeweils die kleinere der beiden oberen Karten *umgedreht* auf einen neuen Stapel





Merge-sort in C++

```
// PRE: [first, last) is a valid range
// POST: the elements *p, p in [first, last) are in
//       ascending order
void merge_sort (Vit first, Vit last)
{
  const int n = last - first;
  if (n <= 1) return;           // nothing to do
  const Vit middle = first + n/2;
  merge_sort (first, middle);  // sort first half
  merge_sort (middle, last);   // sort second half
  merge (first, middle, last); // merge both halfs
}
```

Merge-sort in C++: Teile

```
// PRE: [first, last) is a valid range
// POST: the elements *p, p in [first, last) are in
//       ascending order
void merge_sort (Vit first, Vit last)
{
  const int n = last - first;
  if (n <= 1) return;              // nothing to do
  const Vit middle = first + n/2;
  merge_sort (first, middle);   // sort first half
  merge_sort (middle, last);    // sort second half
  merge (first, middle, last); // merge both halfs
}
```

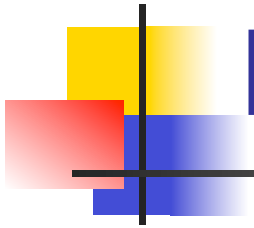
$\lfloor n / 2 \rfloor$ Elemente ($n / 2$ abgerundet)

$\lceil n / 2 \rceil$ Elemente ($n / 2$ aufgerundet)

Merge-sort in C++: Herrsche

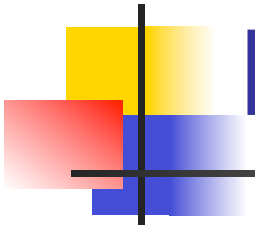
```
// PRE: [first, last) is a valid range
// POST: the elements *p, p in [first, last) are in
//       ascending order
void merge_sort (Vit first, Vit last)
{
  const int n = last - first;
  if (n <= 1) return;              // nothing to do
  const Vit middle = first + n/2;
  merge_sort (first, middle);  // sort first half
  merge_sort (middle, last);   // sort second half
  merge (first, middle, last); // merge both halfs
}
```


Diese Funktion übernimmt das Mischen zu einem Stapel



Die Merge-Funktion

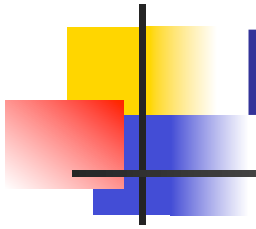
```
// PRE: [first, middle), [middle, last) are valid ranges; in
//      both of them, the elements are in ascending order
void merge (Vit first, Vit middle, Vit last)
{
  const int n = last - first;   // total number of cards
  std::vector<int> deck (n);    // new deck to be built

  Vit left = first;   // top card of left deck
  Vit right = middle; // top card of right deck
  for (Vit d = deck.begin(); d != deck.end(); ++d)
    // put next card onto new deck
    if      (left == middle) *d = *right++; // left deck is empty
    else if (right == last)  *d = *left++;  // right deck is empty
    else if (*left < *right) *d = *left++;  // smaller top card left
    else                     *d = *right++; // smaller top card right

  // copy new deck back into [first, last)
  Vit d = deck.begin();
  while (first != middle) *first++ = *d++;
  while (middle != last) *middle++ = *d++;
}
```



Merge-sort: Laufzeit

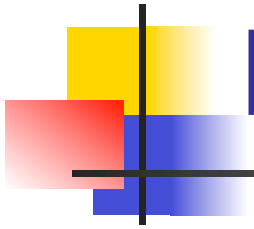
- ist wieder proportional zur Anzahl der Vergleiche `*left < *right` (das muss man aber nicht sofort sehen)
- Alle Vergleiche werden von der Funktion **`merge`** durchgeführt

Merge-sort: Laufzeit



- ist wieder proportional zur Anzahl der Vergleiche **`*left < *right`** (das muss man aber nicht sofort sehen)
- Alle Vergleiche werden von der Funktion **`merge`** durchgeführt

Beim Mischen zweier Stapel zu einem Stapel der Grösse *n* braucht **`merge`** höchstens *n* -1 Vergleiche (maximal einer für jede Karte des neuen Stapels, ausser der letzten)



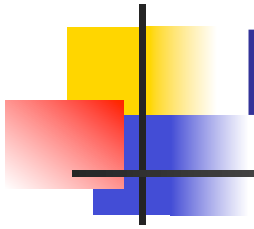
Merge-sort: Laufzeit

Satz:

Die Funktion **`merge_sort`** sortiert eine Folge von $n \geq 1$ Zahlen mit höchstens

$$(n - 1)\ \lceil \log_2 n \rceil$$

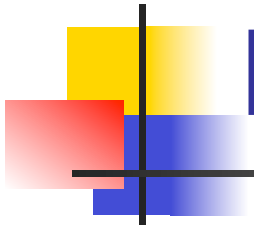
Vergleichen zwischen zwei Zahlen.



Merge-sort: Laufzeit

Beweis:

$T(n)$ sei die *maximal* mögliche Anzahl von Vergleichen zwischen Zahlen, die beim Sortieren von n Zahlen mit **`merge_sort`** auftreten können.



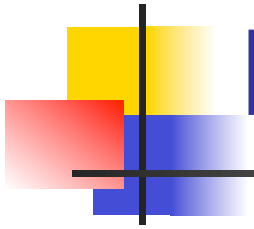
Merge-sort: Laufzeit

Beweis:

$T(n)$ sei die *maximal* mögliche Anzahl von Vergleichen zwischen Zahlen, die beim Sortieren von n Zahlen mit **`merge_sort`** auftreten können.

- $T(0) = T(1) = 0$

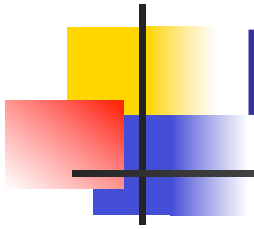
Merge-sort: Laufzeit



Beweis:

$T(n)$ sei die *maximal* mögliche Anzahl von Vergleichen zwischen Zahlen, die beim Sortieren von n Zahlen mit **`merge_sort`** auftreten können.

- $T(0) = T(1) = 0$
- $T(2) = 1$



Merge-sort: Laufzeit

Beweis:

$T(n)$ sei die *maximal* mögliche Anzahl von Vergleichen zwischen Zahlen, die beim Sortieren von n Zahlen mit **`merge_sort`** auftreten können.

- $T(0) = T(1) = 0$
- $T(2) = 1$
- $T(3) = 2$

Merge-sort: Laufzeit

Beweis:

$T(n)$ sei die *maximal* mögliche Anzahl von Vergleichen zwischen Zahlen, die beim Sortieren von n Zahlen mit **`merge_sort`** auftreten können.

- $T(n) \leq T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + n - 1$

  für $n \geq 2$

Merge-sort: Laufzeit

Beweis:

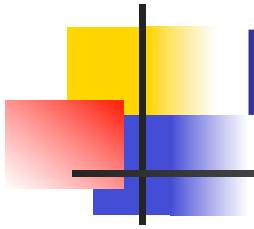
$T(n)$ sei die *maximal* mögliche Anzahl von Vergleichen zwischen Zahlen, die beim Sortieren von n Zahlen mit **`merge_sort`** auftreten können.

- $T(n) \leq T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + n - 1$ für $n \geq 2$

maximale Anzahl im linken Stapel

maximale Anzahl im rechten Stapel

maximale Anzahl beim Mischen



Merge-sort: Laufzeit

- Mit vollständiger Induktion beweisen wir nun (für $n \geq 1$) die Aussage

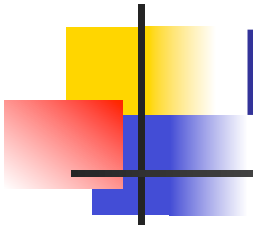
$$T(n) \leq (n-1)\lceil \log_2 n \rceil$$

Merge-sort: Laufzeit

- Mit vollständiger Induktion beweisen wir nun (für $n \geq 1$) die Aussage

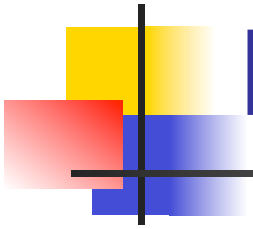
$$T(n) \leq (n-1)\lceil \log_2 n \rceil$$

- n=1: $T(1) = 0 = (1 - 0)\lceil \log_2 1 \rceil$ 



Merge-sort: Laufzeit

- Sei nun $n \geq 2$ und gelte die Aussage für alle Werte in $\{1,\ldots,n-1\}$ (Induktionsannahme)



Merge-sort: Laufzeit

- Sei nun $n \geq 2$ und gelte die Aussage für alle Werte in $\{1,\ldots,n-1\}$ (Induktionsannahme)
- Zu zeigen ist, dass die Aussage dann auch für n gilt (Induktionsschritt)

Merge-sort: Laufzeit

- Es gilt

$$T(n) \leq T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + n - 1$$

≥ 1 und $< n$

Merge-sort: Laufzeit

- Es gilt

$$T(n) \le T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + n - 1$$

$$\le (\lfloor n/2 \rfloor - 1)\ \lceil \log_2 \lfloor n/2 \rfloor \rceil\ +$$

$$(\lceil n/2 \rceil - 1)\ \lceil \log_2 \lceil n/2 \rceil \rceil\ + n - 1$$

Induktionsannahme

Merge-sort: Laufzeit

- Es gilt

$$
\begin{aligned}
T(n) &\le T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + n - 1 \\
&\le (\lfloor n/2 \rfloor - 1)\, \lceil \log_2 \lfloor n/2 \rfloor \rceil \; + \\
&\quad (\lceil n/2 \rceil - 1)\, \lceil \log_2 \lceil n/2 \rceil \rceil \; + n - 1 \\
&\le (\lfloor n/2 \rfloor - 1)\, (\lceil \log_2 n \rceil - 1) \; + \\
&\quad (\lceil n/2 \rceil - 1)\, (\lceil \log_2 n \rceil - 1) + n - 1
\end{aligned}
$$

Aufgabe 121

Merge-sort: Laufzeit

- Es gilt

$$
\begin{aligned}
T(n) &\le T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + n - 1 \\
&\le (\lfloor n/2 \rfloor - 1)\, \lceil \log_2 \lfloor n/2 \rfloor \rceil \; + \\
&\quad (\lceil n/2 \rceil - 1)\, \lceil \log_2 \lceil n/2 \rceil \rceil \; + n - 1 \\
&\le (\lfloor n/2 \rfloor - 1)\, (\lceil \log_2 n \rceil - 1) + \\
&\quad (\lceil n/2 \rceil - 1)\, (\lceil \log_2 n \rceil - 1) + n - 1 \\
&= (n - 2)\, (\lceil \log_2 n \rceil - 1) + n - 1
\end{aligned}
$$

Merge-sort: Laufzeit

- Es gilt

$$
\begin{aligned}
T(n) &\le T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + n - 1 \\
&\le (\lfloor n/2 \rfloor - 1)\, \lceil \log_2 \lfloor n/2 \rfloor \rceil \; + \\
&\quad (\lceil n/2 \rceil - 1)\, \lceil \log_2 \lceil n/2 \rceil \rceil \; + n - 1 \\
&\le (\lfloor n/2 \rfloor - 1)\, (\lceil \log_2 n \rceil - 1) + \\
&\quad (\lceil n/2 \rceil - 1)\, (\lceil \log_2 n \rceil - 1) + n - 1 \\
&\le (n - 1)\, (\lceil \log_2 n \rceil - 1) + n - 1
\end{aligned}
$$

Merge-sort: Laufzeit

- Es gilt

$$
\begin{aligned}
T(n) &\le T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + n - 1 \\
&\le (\lfloor n/2 \rfloor - 1)\, \lceil \log_2 \lfloor n/2 \rfloor \rceil \; + \\
&\quad (\lceil n/2 \rceil - 1)\, \lceil \log_2 \lceil n/2 \rceil \rceil \; + n - 1 \\
&\le (\lfloor n/2 \rfloor - 1)\, (\lceil \log_2 n \rceil - 1) + \\
&\quad (\lceil n/2 \rceil - 1)\, (\lceil \log_2 n \rceil - 1) + n - 1 \\
&\le (n - 1)\, (\lceil \log_2 n \rceil - 1) + n - 1 \\
&= (n - 1)\, (\lceil \log_2 n \rceil)
\end{aligned}
$$

Merge-sort: Laufzeit (alter Mac) auf zufälliger Eingabe

n	100,000	200,000	400,000	800,000	1,600,000
Mcomp (Millionen Vergleiche)	1.7	3.6	7.6	16	33.6
Laufzeit in ms	46	93	190	390	834
Laufzeit / GComp (s)	27	25.8	25	24.4	25.1

`merge_sort` braucht 8-mal mehr Zeit pro Vergleich als `minimum_sort...`

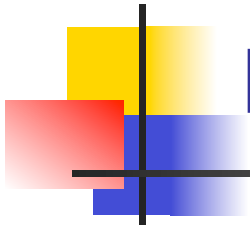
Merge-sort: Laufzeit (alter Mac) auf zufälliger Eingabe

n	100,000	200,000	400,000	800,000	1,600,000
Mcomp (Millionen Vergleiche)	1.7	3.6	7.6	16	33.6
Laufzeit in ms	46	93	190	390	834
Laufzeit / GComp (s)	27	25.8	25	24.4	25.1

> 1h bei minimum_sort

...ist aber aufgrund *sehr* viel weniger Vergleichen trotzdem sehr viel schneller!

Mit wievielen Vergleichen kann man *n* Zahlen sortieren?



- **`minimum_sort`**: $n(n-1)/2$
- **`merge_sort`**: $\leq (n-1)\,(\lceil \log_2 n \rceil)$
- **`optimal_sort`**: ???

Mit wievielen Vergleichen kann man *n* Zahlen sortieren?

- **`minimum_sort`**: $n(n-1)/2$
- **`merge_sort`**: $\leq (n-1)\,(\lceil \log_2 n \rceil)$
- **`optimal_sort`**: ???

Satz: jedes vergleichsbasierte Sortierverfahren braucht mindestens $\lceil \log_2 (n!) \rceil \approx n \log_2 n - 1.44\, n$ Vergleiche; also ist **`merge_sort`** fast optimal.

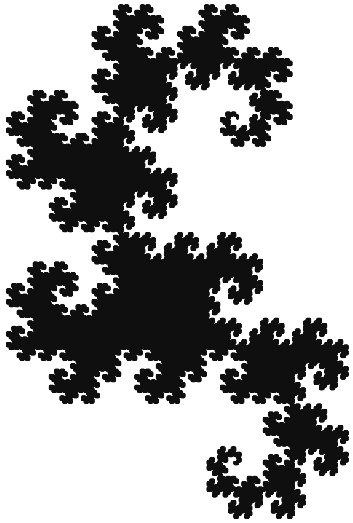
Mit wievielen Vergleichen kann man *n* Zahlen sortieren?

- **`minimum_sort`**: $n(n-1)/2$
- **`merge_sort`**: $\leq (n-1)\,(\lceil \log_2 n \rceil)$
- **`optimal_sort`**: ???

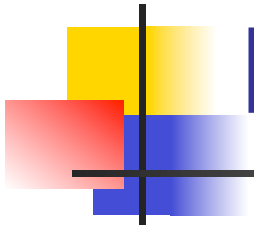
Satz: jedes vergleichsbasierte Sortierverfahren braucht mindestens $\lceil \log_2 (n!) \rceil \approx n \log_2 n - 1.44\, n$ Vergleiche; also ist **`merge_sort`** fast optimal.

Aber: Anzahl der Vergleiche ist nur bedingt geeignet zur Vorhersage praktischer Effizienz. **`quick_sort`** : braucht etwas mehr Vergleiche, ist aber schneller

Lindenmayer-Systeme: Fraktale rekursiv zeichnen

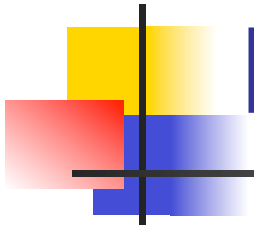


Lindenmayer-Systeme: Definition



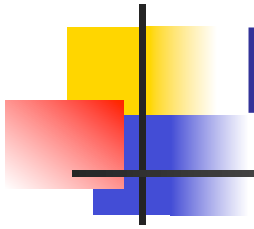
- *Alphabet* Σ (Beispiel: {F, +, -})

Lindenmayer-Systeme: Definition



- *Alphabet* Σ (Beispiel: {F, +, -})
- Σ^* = Menge aller endlichen Wörter über Σ (Beispiel: F+F+ ist in Σ^*)

Lindenmayer-Systeme: Definition



- *Alphabet* Σ (Beispiel: {F, +, -})
- Σ^* = Menge aller endlichen Wörter über Σ (Beispiel: F+F+ ist in Σ^*)
- P: Σ -> Σ^* eine *Produktion*

Lindenmayer-Systeme: Definition

- *Alphabet* Σ (Beispiel: {F, +, -})
- Σ^* = Menge aller endlichen Wörter über Σ (Beispiel: F+F+ ist in Σ^*)
- P: Σ -> Σ^* eine *Produktion*

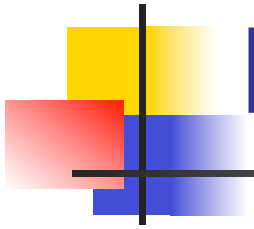
Beispiel:

P(F) = F+F+

P(+) = +

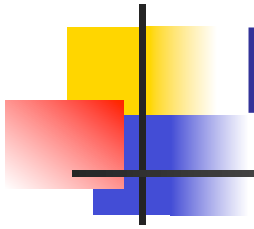
P(-) = -

Lindenmayer-Systeme: Definition



- *Alphabet* Σ (Beispiel: {F, +, -})
- Σ^* = Menge aller endlichen Wörter über Σ (Beispiel: F+F+ ist in Σ^*)
- P: Σ -> Σ^* eine *Produktion*
- *s* aus Σ^* ein *Startwort* (Beispiel: F)

Lindenmayer-Systeme: Definition



- *Alphabet* Σ (Beispiel: {F, +, -})
- Σ^* = Menge aller endlichen Wörter über Σ (Beispiel: F+F+ ist in Σ^*)
- P: Σ -> Σ^* eine *Produktion*
- *s* aus Σ^* ein *Startwort* (Beispiel: F)

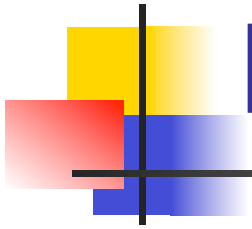
Def.: (Σ, P, s) ist *Lindenmayer-System*.

Lindenmayer-Systeme: Die beschriebenen Wörter

Die von (Σ, P, s) *beschriebenen* Wörter:

- w_0 = F
- w_1 = F+F+
- w_2 = F+F++F+F++
- w_3 = F+F++F+F+++F+F++F+F+++
- ...

Lindenmayer-Systeme: Die beschriebenen Wörter



Die von (Σ, P, s) *beschriebenen* Wörter:

- w_0 = F
- w_1 = F+F+
- w_2 = F+F++F+F++
- w_3 = F+F++F+F+++F+F++F+F+++
- ...

w_i entsteht aus w_{i-1} durch Ersetzen aller Symbole mittels P.

Lindenmayer-Systeme: Die beschriebenen Wörter

Die von (Σ, P, s) *beschriebenen* Wörter:

- w_0 = F
- w_1 = F+F+
- w_2 = F+F++F+F++
- w_3 = F+F++F+F+++F+F++F+F+++
- ...

w_2 entsteht aus w_1 durch Ersetzen aller Symbole mittels P.

F → F+F+ + → + - → -

Lindenmayer-Systeme: Turtle-Grafik

Turtle-Grafik:

- Schildkröte mit *Position* und *Richtung*

Lindenmayer-Systeme: Turtle-Grafik

Turtle-Grafik:

- Schildkröte mit *Position* und *Richtung*

- versteht folgende Kommandos:
  - F: gehe einen Schritt in deine Richtung (und markiere ihn in Schwarz)
  - + / - : drehe dich um 90^o gegen / im UZS

Lindenmayer-Systeme: Turtle-Grafik

“Zeichnen eines Wortes”:

F+F+

Lindenmayer-Systeme: Turtle-Grafik

“Zeichnen eines Wortes”:

F+F+

Lindenmayer-Systeme: Turtle-Grafik

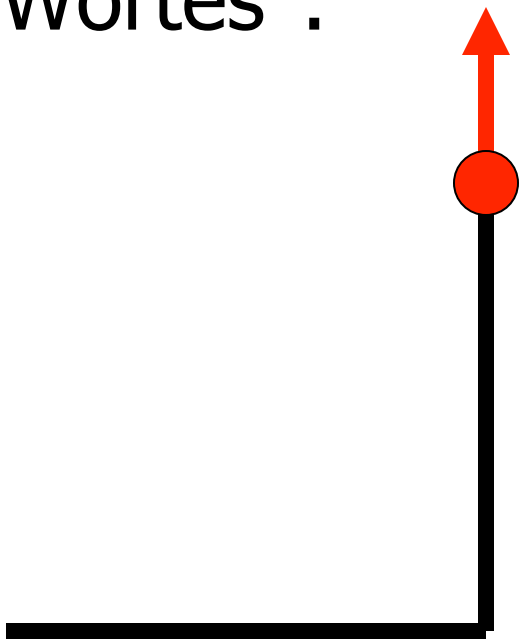
“Zeichnen eines Wortes”:

F+F+

Lindenmayer-Systeme: Turtle-Grafik

"Zeichnen eines Wortes":

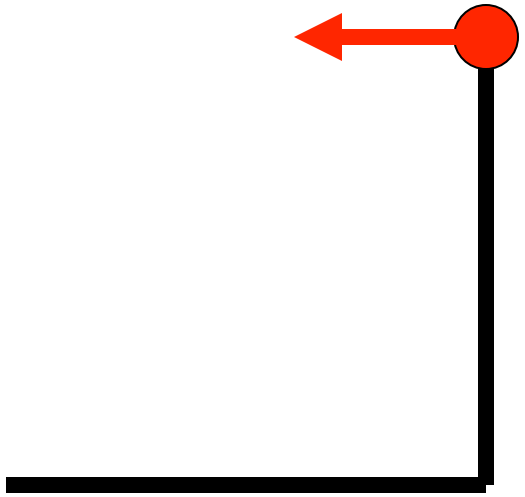
F+F+



Lindenmayer-Systeme: Turtle-Grafik

"Zeichnen eines Wortes":

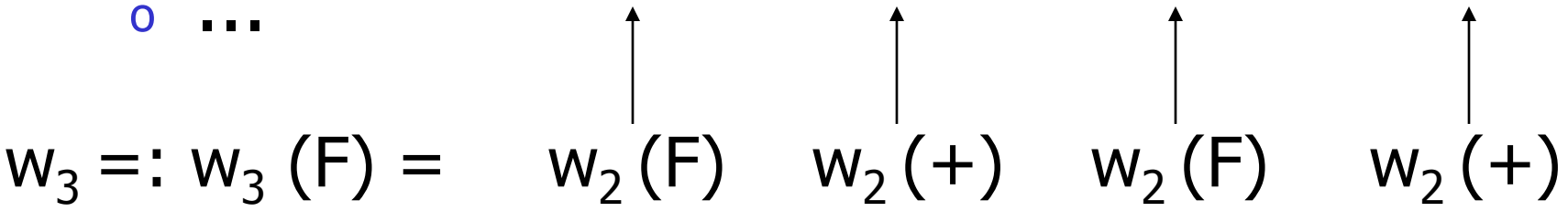
F+F+



Lindenmayer-Systeme: Rekursives Zeichnen

Zeichnen von w_i:

- w_0 = F
- w_1 = F+F+
- w_2 = F+F++F+F++
- w_3 = F+F++F+F+++F+F++F+F+++
- ...



Lindenmayer-Systeme: Rekursives Zeichnen

Zeichnen von w_i:

- w_0 = F
- w_1 = F+F+
- w_2 = F+F++F+F++
- w_3 = $\underbrace{\text{F+F++F+F++}}$+$\underbrace{\text{F+F++F+F++}}$+
- ...

$w_i =: w_i(F) =$ $w_{i-1}(F)$ $w_{i-1}(+)$ $w_{i-1}(F)$ $w_{i-1}(+)$

Lindenmayer-Systeme: Rekursives Zeichnen

Zeichnen von w_i:

- $w_0 = F$
- $w_1 = F+F+$
- $w_2 = F+F++F+F++$
- $w_3 = F+F++F+F+++\underbrace{F+F++F+F+}_{w_1(F)}\underbrace{++F+F++F+F+}_{w_1(F)}++$
- ...

$F+F+ = w_1(F)$

$w_i =: w_i(F) = w_{i-1}(F)\ \underbrace{w_{i-1}(+)}_{+}\ w_{i-1}(F)\ \underbrace{w_{i-1}(+)}_{+}$

Lindenmayer-Systeme: Rekursives Zeichnen

Zeichnen von w_i:

- w_0 = F
- w_1 = F+F+
- w_2 = F+F++F+F++
- w_3 = $\underbrace{\text{F+F++F+F++}}$+$\underbrace{\text{F+F++F+F++}}$+
- ...

$w_i =: w_i(F) =$ $w_{i-1}(F)$ + $w_{i-1}(F)$ +

Lindenmayer-Systeme: Rekursives Zeichnen (Beispiel)

```
// POST: the word w_i^F is drawn
void f (const unsigned int i) {
  if (i == 0)
    ifm::forward();  // F
  else {
    f(i-1);          // w_{i-1}^F
    ifm::left(90);   // +
    f(i-1);          // w_{i-1}^F
    ifm::left(90);   // +
  }
}
```

$w_i(F) = w_{i-1}(F)+w_{i-1}(F)+$

Lindenmayer-Systeme: Rekursives Zeichnen (Beispiel)

```
// POST: the word w_i^F is drawn
void f (const unsigned int i) {
  if (i == 0)
    ifm::forward();   // F
  else {
    f(i-1);           // w_{i-1}^F
    ifm::left(90);    // +
    f(i-1);           // w_{i-1}^F
    ifm::left(90);    // +
  }
}
```

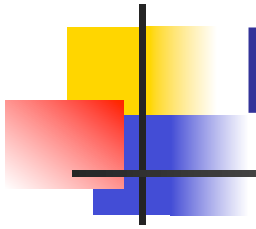
Befehle für Turtle-Grafik (aus der **libwindow**-Bibliothek)

$w_i(F) = w_{i-1}(F) + w_{i-1}(F) +$

Lindenmayer-Systeme: Rekursives Zeichnen (Beispiel)

```
int main () {
  std::cout << "Number of iterations =? ";
  unsigned int n;
  std::cin >> n;

  // draw w_n = w_n(F)
  f(n);

  return 0;
}
```

Lindenmayer-Systeme: Erweiterungen



Neue Symbole (ohne Interpretation in Turtle-Grafik):

Beispiel *Drachenkurve*:

- s = X
- P(X) = X+YF+, P(Y) = -FX-Y
- $w_i = w_i(X) = w_{i-1}(X)+w_{i-1}(Y)F+$

Lindenmayer-Systeme: Erweiterungen (Drachen)

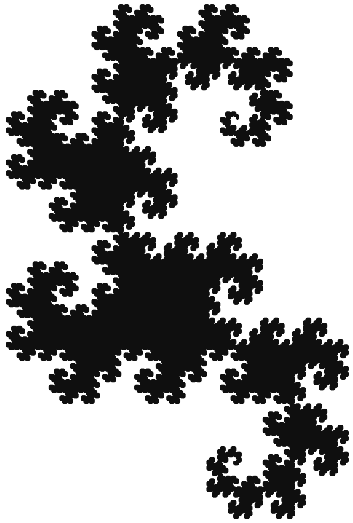
```
// POST: w_i^X is drawn
void x (const unsigned int i) {
  if (i > 0) {
    x(i-1);            // w_{i-1}^X
    ifm::left(90);     // +
    y(i-1);            // w_{i-1}^Y
    ifm::forward();    // F
    ifm::left(90);     // +
  }
}

// POST: w_i^Y is drawn
void y (const unsigned int i) {
  if (i > 0) {
    ifm::right(90);    // -
    ifm::forward();    // F
    x(i-1);            // w_{i-1}^X
    ifm::right(90);    // -
    y(i-1);            // w_{i-1}^Y
  }
}
```

$$w_i(X) = w_{i-1}(X)+w_{i-1}(Y)F+$$

$$w_i(Y) = -Fw_{i-1}(X)-w_{i-1}(Y)$$

Lindenmayer-Systeme: Drachen

Programm **`dragon.cpp`** :



Lindenmayer-Systeme: Erweiterungen

Drehwinkel α kann frei gewählt werden.

Beispiel *Schneeflocke*:

- $\alpha = 60^o$
- s = F++F++F
- P(F) = F-F++F-F
- $w_i = w_i(F++F++F) = \underbrace{w_i(F)++w_i(F)++w_i(F)}_{w_{i-1}(F)-w_{i-1}(F)++w_{i-1}(F)-w_{i-1}(F)}$

Lindenmayer-Systeme: Schneeflocke

```
// POST: the word w_i^F is drawn
void f (const unsigned int i) {
  if (i == 0)
    ifm::forward();  // F
  else {
    f(i-1);          // w_{i-1}^F
    ifm::right(60);  // -
    f(i-1);          // w_{i-1}^F
    ifm::left(120);  // ++
    f(i-1);          // w_{i-1}^F
    ifm::right(60);  // -
    f(i-1);          // w_{i-1}^F
  }
}
```

$w_i(F) = w_{i-1}(F)\text{-}w_{i-1}(F)\text{++}w_{i-1}(F)\text{-}w_{i-1}(F)$

Lindenmayer-Systeme: Schneeflocke

```
int main () {
  std::cout << "Number of iterations =? ";
  unsigned int n;
  std::cin >> n;

  // draw w_n = w_n^F++w_n^F++w_n^F
  f(n);                 // w_n^F
  ifm::left(120);       // ++
  f(n);                 // w_n^F
  ifm::left(120);       // ++
  f(n);                 // w_n^F

  return 0;
}
```

Lindenmayer-Systeme: Schneeflocke

Programm **`snowflake.cpp`** :