Rekursion

Mathematische Rekursion

- Viele mathematische Funktionen sind sehr natürlich *rekursiv* definierbar, d.h.
- die Funktion erscheint in ihrer eigenen Definition.

Mathematische Rekursion

- Viele mathematische Funktionen sind sehr natürlich *rekursiv* definierbar, d.h.
- die Funktion erscheint in ihrer eigenen Definition.

$$n! = \begin{cases} 1, & \text{falls } n \leq 1 \\ n \cdot (n-1)!, & \text{falls } n > 1 \end{cases}$$

Rekursion in C++

- modelliert oft direkt die mathematische Rekursionsformel

Rekursion in C++

- modelliert oft direkt die mathematische Rekursionsformel.

```
// POST: return value is n!
unsigned int fac (unsigned int n)
{
  if (n <= 1) return 1;
  return n * fac(n-1); // n > 1
}
```

Unendliche Rekursion

- ist so leicht zu erzeugen wie eine unendliche Schleife,
- sollte aber genauso vermieden werden.

Unendliche Rekursion

- ist so leicht zu erzeugen wie eine unendliche Schleife,
- sollte aber genauso vermieden werden.

```
void f()
{
  f(); // calls f(), calls f(), calls f(),...
}
```

Terminierung von rekursiven Funktionsaufrufen

Wie bei Schleifen brauchen wir

- Fortschritt Richtung Terminierung

fac(n) :

terminiert sofort für $n \leq 1$, andernfalls wird die Funktion rekursiv mit Argument $< n$ aufgerufen.

Terminierung von rekursiven Funktionsaufrufen

Wie bei Schleifen brauchen wir

- Fortschritt Richtung Terminierung

`fac(n)` :

terminiert sofort für $n \leq 1$, andernfalls wird die Funktion rekursiv mit Argument $< n$ aufgerufen.

"n wird mit jedem Aufruf kleiner."

Auswertung rekursiver Funktionsaufrufe (z.B. `fac(3)`)

```
// POST: return value is n!
unsigned int fac (unsigned int n)
{
  if (n <= 1) return 1;
  return n * fac(n-1); // n > 1
}
```

Auswertung rekursiver Funktionsaufrufe (z.B. `fac(3)`)

```
// POST: return value is n!
unsigned int fac (unsigned int n)
{ // n = 3
  if (n <= 1) return 1;
  return n * fac(n-1); // n > 1
}
```

Initialisierung des formalen Arguments mit dem Wert des Aufrufarguments

Auswertung rekursiver Funktionsaufrufe (z.B. `fac(3)`)

```
// POST: return value is n!
unsigned int fac (unsigned int n)
{ // n = 3
  if (n <= 1) return 1;
  return n * fac(n-1); // n > 1
}
```

Ausführen des Funktionsrumpfs: Auswertung des Rückgabeausdrucks

Auswertung rekursiver Funktionsaufrufe (z.B. `fac(3)`)

```
// POST: return value is n!
unsigned int fac (unsigned int n)
{ // n = 3
  if (n <= 1) return 1;
  return n * fac(n-1); // n > 1
}
```

Ausführen des Funktionsrumpfs: Rekursiver Aufruf von fac mit Aufrufargument `n-1 == 2`

Auswertung rekursiver Funktionsaufrufe (z.B. `fac(3)`)

```
// POST: return value is n!
unsigned int fac (unsigned int n)
{ // n = 2
  if (n <= 1) return 1;
  return n * fac(n-1); // n > 1
}
```

Initialisierung des formalen Arguments mit dem Wert des Aufrufarguments

Auswertung rekursiver Funktionsaufrufe (z.B. `fac(3)`)

```
// POST: return value is n!
unsigned int fac (unsigned int n)
{ // n = 2
  if (n <= 1) return 1;
  return n * fac(n-1); // n > 1
}
```

Es gibt jetzt zwei *n* ! Das von `fac (3)`, und das von `fac (2)`

Initialisierung *des* formalen Arguments mit dem Wert des Aufrufarguments

Auswertung rekursiver Funktionsaufrufe (z.B. `fac(3)`)

```
// POST: return value is n!
unsigned int fac (unsigned int n)
{ // n = 2
  if (n <= 1) return 1;
  return n * fac(n-1); // n > 1
}
```

Wir nehmen das Argument des *aktuellen* Aufrufs, `fac (2)`

Initialisierung *des* formalen Arguments mit dem Wert des Aufrufarguments

Der Aufrufstapel

Bei Auswertung jedes Funktionsaufrufs:

- Wert des Aufrufarguments kommt auf einen Stapel (erst n = 3, dann n = 2,...)
- es wird stets mit dem obersten Wert gearbeitet
- nach Ende des Funktionsaufrufs wird der oberste Wert vom Stapel gelöscht

Der Aufrufstapel

Bei Auswertung jedes Funktionsaufrufs:

- Wert des Aufrufarguments kommt auf einen Stapel (erst n = 3, dann n = 2,...)
- es wird stets mit dem obersten Wert gearbeitet
- nach Ende eines Funktionsaufrufs wird der oberste Wert vom Stapel gelöscht

Bei mehreren Argumenten analog (alle kommen auf den Stapel, wir arbeiten jeweils mit der obersten Version jedes Arguments)

Grösster gemeinsamer Teiler

Euklidischer Algorithmus

- findet den grössten gemeinsamen Teiler *gcd (a, b)* zweier natürlicher Zahlen *a* und *b*

Grösster gemeinsamer Teiler

Euklidischer Algorithmus

- findet den grössten gemeinsamen Teiler *gcd (a, b)* zweier natürlicher Zahlen *a* und *b*
- basiert auf folgendem Lemma (Beweis im Skript):

$gcd(a, b) = gcd(b, a \bmod b)$ für $b > 0$.

Grösster gemeinsamer Teiler

```
// POST: return value is the greatest common
//       divisor of a and b
unsigned int gcd (unsigned int a, unsigned int b)
{
  if (b == 0) return a;
  return gcd(b, a % b);  // b != 0
}
```

Grösster gemeinsamer Teiler

```
// POST: return value is the greatest common
//       divisor of a and b
unsigned int gcd (unsigned int a, unsigned int b)
{
  if (b == 0) return a;
  return gcd(b, a % b);  // b != 0
}
```

Korrektheit: $gcd\ (a,\ 0) = a$

$gcd\ (a,\ b) \underset{\text{Lemma!}}{=} gcd\ (b,\ a\ mod\ b),\quad b > 0$

Grösster gemeinsamer Teiler

```
// POST: return value is the greatest common
//       divisor of a and b
unsigned int gcd (unsigned int a, unsigned int b)
{
  if (b == 0) return a;
  return gcd(b, a % b);  // b != 0
}
```

Terminierung: $a\ mod\ b < b$, also wird b in jedem rekursiven Aufruf kleiner.

Grösster gemeinsamer Teiler

```
// POST: return value is the greatest common
//       divisor of a and b
unsigned int gcd (unsigned int a, unsigned int b)
{
  if (b == 0) return a;
  return gcd(b, a % b);  // b != 0
}
```

Korrektheit *und* Terminierung müssen bei rekursiven Funktionen stets separat bewiesen werden!

Fibonacci-Zahlen

- $F_0 := 0$
- $F_1 := 1$
- $F_n := F_{n-1} + F_{n-2}$, $n > 1$

Fibonacci-Zahlen

- $F_0 := 0$
- $F_1 := 1$
- $F_n := F_{n-1} + F_{n-2}$, $n > 1$

0, 1, 1, 2, 3, 5, 8, 13, 21, ...

Fibonacci-Zahlen

- $F_0 := 0$
- $F_1 := 1$
- $F_n := F_{n-1} + F_{n-2}$, $n > 1$

0, 1, 1, 2, 3, 5, 8, 13, 21, ...

- Leonardo von Pisa (Fibonacci) modellierte damit Hasenpopulationen (1202)

Fibonacci-Zahlen

```
// POST: return value is the n-th
//       Fibonacci number F_n
unsigned int fib (unsigned int n)
{
  if (n == 0) return 0;
  if (n == 1) return 1;
  return fib(n-1) + fib(n-2); // n > 1
}
```

Fibonacci-Zahlen

```
// POST: return value is the n-th
//       Fibonacci number F_n
unsigned int fib (unsigned int n)
{
  if (n == 0) return 0;
  if (n == 1) return 1;
  return fib(n-1) + fib(n-2); // n > 1
}
```

Korrektheit und Terminierung sind klar.

Fibonacci-Zahlen

```
// POST: return value is the n-th
//       Fibonacci number F_n
unsigned int fib (unsigned int n)
{
  if (n == 0) return 0;
  if (n == 1) return 1;
  return fib(n-1) + fib(n-2); // n > 1
}
```

Laufzeit:

`fib (50)` dauert "ewig", denn es berechnet

F_{48} 2-mal, F_{47} 3-mal, F_{46} 5-mal,
F_{45} 8-mal, F_{44} 13-mal, ...

Rekursion und Iteration

Rekursion kann im Prinzip ersetzt werden durch

- Iteration (Schleifen) und
- expliziten Aufrufstapel (z.B. Feld).

Oft sind direkte rekursive Formulierungen einfacher, aber manchmal auch weniger effizient.

Endrekursion

Eine Funktion ist endrekursiv, wenn sie genau einen rekursiven Aufruf ganz am Ende enthält.

Endrekursion

Eine Funktion ist endrekursiv, wenn sie genau einen rekursiven Aufruf ganz am Ende enthält.

```
// POST: return value is the greatest common
//       divisor of a and b
unsigned int gcd (unsigned int a, unsigned int b)
{
  if (b == 0) return a;
  return gcd(b, a % b);  // b != 0
}
```

Endrekursion

- ist leicht iterativ zu schreiben.

```
// POST: return value is the greatest common divisor of a and b
unsigned int gcd2 (unsigned int a, unsigned int b)
{
  while (b != 0) {
    unsigned int a_prev = a;
    a = b;
    b = a_prev % b;
  }
  return a;
}
```

Endrekursion

- ist leicht iterativ zu schreiben.

```
// POST: return value is the greatest common divisor of a and b
unsigned int gcd2 (unsigned int a, unsigned int b)
{
  while (b != 0) {
    unsigned int a_prev = a;
    a = b;
    b = a_prev % b;
  }
  return a;
}
```

"(a,b) → (b, a mod b)"

Endrekursion

- ist leicht iterativ zu schreiben.

```
// POST: return value is the greatest common divisor of a and b
unsigned int gcd2 (unsigned int a, unsigned int b)
{
  while (b != 0) {
    unsigned int a_prev = a;
    a = b;
    b = a_prev % b;
  }
  return a;
}
```

"(a,b) → (b, a mod b)"

Endrekursion

- ist leicht iterativ zu schreiben.

```
// POST: return value is the greatest common divisor of a and b
unsigned int gcd2 (unsigned int a, unsigned int b)
{
  while (b != 0) {
    unsigned int a_prev = a;
    a = b;
    b = a_prev % b;
  }
  return a;
}
```

"(a,b) → (b, a mod b)"

Endrekursion

- ist leicht iterativ zu schreiben.

```
// POST: return value is the greatest common divisor of a and b
unsigned int gcd2 (unsigned int a, unsigned int b)
{
  while (b != 0) {
    unsigned int a_prev = a;
    a = b;
    b = a_prev % b;
  }
  return a;
}
```

"(a,b) → (b, a mod b)"

...aber die rekursive Version is lesbarer und (fast) genauso effizient.

Fibonacci-Zahlen iterativ

Idee:

- berechne jede Zahl genau einmal, in der Reihenfolge $F_0, F_1, F_2, F_3, \ldots$
- speichere die jeweils letzten beiden berechneten Zahlen (Variablen **`a, b`**), dann kann die nächste Zahl durch eine Addition erhalten werden.

Fibonacci-Zahlen iterativ

```
// POST: return value is the n-th Fibonacci number F_n
unsigned int fib2 (unsigned int n)
{
  if (n == 0) return 0;
  if (n <= 2) return 1;
  unsigned int a = 1;  // F_1
  unsigned int b = 1;  // F_2
  for (unsigned int i = 3; i <= n; ++i) {
    unsigned int a_prev = a;  // F_{i-2}
    a = b;                    // F_{i-1}
    b += a_prev;              // F_{i-1} += F_{i-2} -> F_i
  }
  return b;
}
```

Fibonacci-Zahlen iterativ

```
// POST: return value is the n-th Fibonacci number F_n
unsigned int fib2 (unsigned int n)
{
  if (n == 0) return 0;
  if (n <= 2) return 1;
  unsigned int a = 1;  // F_1
  unsigned int b = 1;  // F_2
  for (unsigned int i = 3; i <= n; ++i) {
    unsigned int a_prev = a;  // F_{i-2}
    a = b;                    // F_{i-1}
    b += a_prev;              // F_{i-1} += F_{i-2} -> F_i
  }
  return b;
}
```

"$(F_{i-2} , F_{i-1}) \rightarrow (F_{i-1} , F_i)$"

a b

Fibonacci-Zahlen iterativ

```
// POST: return value is the n-th Fibonacci number F_n
unsigned int fib2 (unsigned int n)
{
  if (n == 0) return 0;
  if (n <= 2) return 1;
  unsigned int a = 1;  // F_1
  unsigned int b = 1;  // F_2
  for (unsigned int i = 3; i <= n; ++i) {
    unsigned int a_prev = a;  // F_{i-2}
    a = b;                    // F_{i-1}
    b += a_prev;              // F_{i-1} += F_{i-2} -> F_i
  }
  return b;
}
```

"$(F_{i-2} , F_{i-1}) \rightarrow (F_{i-1} , F_i)$"

a b

Fibonacci-Zahlen iterativ

```
// POST: return value is the n-th Fibonacci number F_n
unsigned int fib2 (unsigned int n)
{
  if (n == 0) return 0;
  if (n <= 2) return 1;
  unsigned int a = 1;  // F_1
  unsigned int b = 1;  // F_2
  for (unsigned int i = 3; i <= n; ++i) {
    unsigned int a_prev = a;  // F_{i-2}
    a = b;                    // F_{i-1}
    b += a_prev;              // F_{i-1} += F_{i-2} -> F_i
  }
  return b;
}
```

"$(F_{i-2} , F_{i-1}) \rightarrow (F_{i-1} , F_i)$"

a b

Fibonacci-Zahlen iterativ

```
// POST: return value is the n-th Fibonacci number F_n
unsigned int fib2 (unsigned int n)
{
  if (n == 0) return 0;
  if (n <= 2) return 1;
  unsigned int a = 1;  // F_1
  unsigned int b = 1;  // F_2
  for (unsigned int i = 3; i <= n; ++i) {
    unsigned int a_prev = a;  // F_{i-2}
    a = b;                    // F_{i-1}
    b += a_prev;              // F_{i-1} += F_{i-2} -> F_i
  }
  return b;
}
```

sehr schnell auch bei **`fib2(50)`**

"$(F_{i-2} , F_{i-1}) \rightarrow (F_{i-1} , F_i)$"

a **b**

Die Ackermann-Funktion

$$A\,(m, n) = \begin{cases} n+1\,, & \text{falls } m = 0 \\ A\,(m-1,\, 1)\,, & \text{falls } m > 0,\ n = 0 \\ A\,(m-1,\, A\,(m,\, n-1))\,, & \text{falls } m > 0,\ n > 0 \end{cases}$$

Die Ackermann-Funktion

$$A\,(m, n) = \begin{cases} n+1\,, & \text{falls } m = 0 \\ A\,(m-1,\, 1)\,, & \text{falls } m > 0,\ n = 0 \\ A\,(m-1,\, A\,(m,\, n-1))\,, & \text{falls } m > 0,\ n > 0 \end{cases}$$

- ist *berechenbar*, aber *nicht primitiv rekursiv* (man dachte Anfang des 20. Jahrhunderts, dass es diese Kombination gar nicht gibt)
- wächst *extrem* schnell

Die Ackermann-Funktion

```
// POST: return value is the Ackermann function
//       value A(m,n)
unsigned int A (unsigned int m, unsigned int n) {
  if (m == 0) return n+1;
  if (n == 0) return A(m-1,1);
  return A(m-1, A(m, n-1));
}
```

Die Ackermann-Funktion

m

	0	1	2	3	...	*n*
0	1	2	3	4	...	*n+1*
1						
2						
3						
4						

n

Die Ackermann-Funktion

m

	0	1	2	3	...	*n*
0	1	2	3	4	...	*n+1*
1	2	3	4	5	...	*n+2*
2						
3						
4						

n

Die Ackermann-Funktion

m

	0	1	2	3	...	*n*
0	1	2	3	4	...	*n+1*
1	2	3	4	5	...	*n+2*
2	3	5	7	9	...	*2n+3*
3						
4						

n

Die Ackermann-Funktion

m

	0	1	2	3	...	*n*
0	1	2	3	4	...	*n+1*
1	2	3	4	5	...	*n+2*
2	3	5	7	9	...	*2n+3*
3	5	13	29	61	...	$2^{n+3}-3$
4						

n

Die Ackermann-Funktion

m (rows), *n* (columns)

	0	1	2	3	...	*n*
0	1	2	3	4	...	$n+1$
1	2	3	4	5	...	$n+2$
2	3	5	7	9	...	$2n+3$
3	5	13	29	61	...	$2^{n+3}-3$
4	13	65533	$2^{65536}-3$	$2^{2^{65536}}-3$	...	$2^{2^{\cdot^{\cdot^{2}}}}-3$

Turm von n+3 Zweierpotenzen

Die Ackermann-Funktion

m (rows), *n* (columns)

	0	1	2	3	...	*n*
0	1	2	3	4	...	$n+1$
1	2	3	4	5	...	$n+2$
2	3	5	7	9	...	$2n+3$
3	5	13	29	61	...	$2^{n+3}-3$
4	13	65533	$2^{65536}-3$	$2^{2^{65536}}-3$	...	$2^{2^{\cdot^{\cdot^{2}}}}-3$

nicht mehr praktisch berechenbar

Die Ackermann-Funktion - Moral

- Rekursion ist sehr mächtig...
- ... aber auch gefährlich:

Die Ackermann-Funktion - Moral

- Rekursion ist sehr mächtig...
- ... aber auch gefährlich:

Es ist leicht, harmlos aussehende rekursive Funktionen hinzuschreiben, die theoretisch korrekt sind und terminieren, praktisch aber jeden Berechenbarkeitsrahmen sprengen.

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

8	6	2	5	4	1	7	3

Finde die kleinste Zahl...

Minimum-sort

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

8	6	2	5	4	1	7	3

→

1	6	2	5	4	8	7	3

...und vertausche sie mit der ersten Zahl.

Minimum-sort

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

1	6	2	5	4	8	7	3

Finde die kleinste Zahl...

Minimum-sort

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

1	6	2	5	4	8	7	3

→

1	2	6	5	4	8	7	3

...und vertausche sie mit der ersten Zahl.

Minimum-sort

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

1	2	6	5	4	8	7	3

Finde die kleinste Zahl...

Minimum-sort

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

1	2	6	5	4	8	7	3

Finde die kleinste Zahl...

usw....

Minimum-sort

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

		p					last
1	2	6	5	4	8	7	3

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 (int* first, int* last)
{
  for (int* p = first; p != last; ++p) {
    // find minimum in nonempty range described by [p, last)
    int* p_min = p; // pointer to current minimum
    int* q = p;     // pointer to current element
    while (++q != last)
      if (*q < *p_min) p_min = q;
    // interchange *p with *p_min
    std::iter_swap (p, p_min);
  }
}
```

		p					p_min
1	2	6	5	4	8	7	3

..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 (int* first, int* last)
{
  for (int* p = first; p != last; ++p) {
    // find minimum in nonempty range described by [p, last)
    int* p_min = p; // pointer to current minimum
    int* q = p;     // pointer to current element
    while (++q != last)
      if (*q < *p_min) p_min = q;
    // interchange *p with *p_min
    std::iter_swap (p, p_min);
  }
}
```

p → 3, p_min → 6

1	2	3	5	4	8	7	6

..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 (int* first, int* last)
{
  for (int* p = first; p != last; ++p) {
    // find minimum in nonempty range described by [p, last)
    int* p_min = p; // pointer to current minimum
    int* q = p;     // pointer 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>`

p → 3, p_min → 6

1	2	3	5	4	8	7	6

..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 plattformunab-hängiges 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 (int* first, int* last)
{
  for (int* p = first; p != last; ++p) {
    // find minimum in nonempty range described by [p, last)
    int* p_min = p; // pointer to current minimum
    int* q = p;     // pointer 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 (int* first, int* last)
{
  for (int* p = first; p != last; ++p) {
    // find minimum in nonempty range described by [p, last)
    int* p_min = p; // pointer to current minimum
    int* q = p;     // pointer 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 (int* first, int* last)
{
  for (int* p = first; p != last; ++p) {
    // find minimum in nonempty range described by [p, last)
    int* p_min = p; // pointer to current minimum
    int* q = p;     // pointer 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 (int* first, int* last)
{
  for (int* p = first; p != last; ++p) {
    // find minimum in nonempty range described by [p, last)
    int* p_min = p; // pointer to current minimum
    int* q = p;     // pointer 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 (PC) auf Eingabe 0, *n* -1, 1, *n* -2,...

n	100,000	200,000	400,000	800,000	1,600,000
Gcomp (Milliarden Vergleiche)	5	20	80	320	1280
Laufzeit in min : s	0:17	1:07	4:24	18:06	73:32
Laufzeit / GComp (s)	3.4	3.35	3.3	3.4	3.45

Minimum-sort: Laufzeit (PC) auf Eingabe 0, *n* -1, 1, *n* -2,...

n	100,000	200,000	400,000	800,000	1,600,000
Gcomp (Milliarden Vergleiche)	5	20	80	320	1280
Laufzeit in min : s	0:17	1:07	4:24	18:06	73:32
Laufzeit / GComp (s)	3.4	3.35	3.3	3.4	3.45

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

Minimum-sort: Laufzeit (PC) auf Eingabe 0, *n* -1, 1, *n* -2,...

n	100,000	200,000	400,000	800,000	1,600,000
Gcomp (Milliarden Vergleiche)	5	20	80	320	1280
Laufzeit in min : s	0:17	1:07	4:24	18:06	73:32
Laufzeit / GComp (s)	3.4	3.35	3.3	3.4	3.45

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

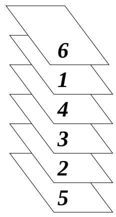
Merge-sort

- 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

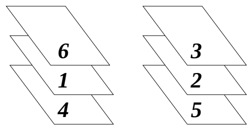
- Analogie: Sortieren eines Kartenstapels
- Karten sind mit Zahlen beschriftet

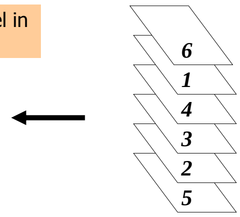


Merge-sort: Teile

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

Schritt 1: Teile den Stapel in zwei gleich grosse Stapel

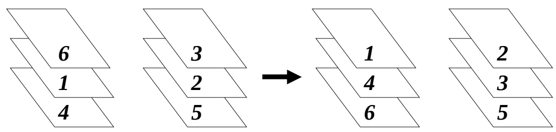




Merge-sort: Teile

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

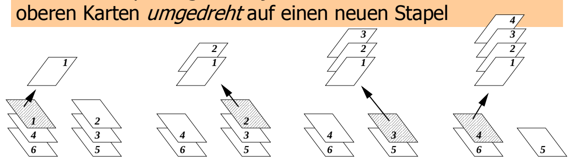
Schritt 2: Sortiere beide Stapel getrennt



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 (int* first, int* last)
{
  int n = last - first;
  if (n <= 1) return;          // nothing to do
  int* 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 (int* first, int* last)
{
  int n = last - first;
  if (n <= 1) return;          // nothing to do
  int* 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 (int* first, int* last)
{
  int n = last - first;
  if (n <= 1) return;          // nothing to do
  int* 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 (int* first, int* middle, int* last)
{
  int n = last - first;   // total number of cards
  int* deck = new int[n]; // new deck to be built

  int* left = first;   // top card of left deck
  int* right = middle; // top card of right deck
  for (int* d = deck; d != deck + n; ++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)
  int *d = deck;
  while (first != middle) *first++ = *d++;
  while (middle != last) *middle++ = *d++;

  delete[] deck;
}
```

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.

- $T(n) \leq \underbrace{T(\lfloor n/2 \rfloor)}_{\text{maximale Anzahl im linken Stapel}} + \underbrace{T(\lceil n/2 \rceil)}_{\text{maximale Anzahl im rechten Stapel}} + \underbrace{n-1}_{\text{maximale Anzahl beim Mischen}}$ für n ≥ 2

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

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

Induktionsannahme

Merge-sort: Laufzeit

- Es gilt

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

Aufgabe 103

Merge-sort: Laufzeit

- Es gilt

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

Merge-sort: Laufzeit

- Es gilt

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

Merge-sort: Laufzeit

- Es gilt

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

Merge-sort: Laufzeit (PC) auf Eingabe 0, n-1, 1, n-2,...

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 s	0.75	1.29	1.96	3.2	5.36
Laufzeit / GComp (s)	441	358	257	200	160

Abnehmend! Je grösser n, desto weniger fallen andere Operationen ins Gewicht

Merge-sort: Laufzeit (PC) auf Eingabe 0, *n* -1, 1, *n* -2,...

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 s	0.75	1.29	1.96	3.2	5.36
Laufzeit / GComp (s)	441	358	257	200	160

`merge_sort` braucht ≥ 50-mal so viel Zeit pro Gcomp wie `minimum_sort`...

Merge-sort: Laufzeit (PC) auf Eingabe 0, *n* -1, 1, *n* -2,...

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 s	0.75	1.29	1.96	3.2	5.36
Laufzeit / GComp (s)	441	358	257	200	160

> 2h bei minimum_sort

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