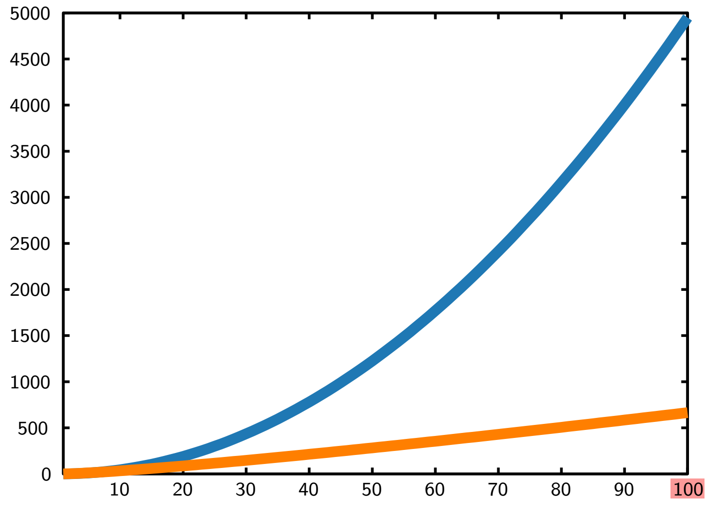
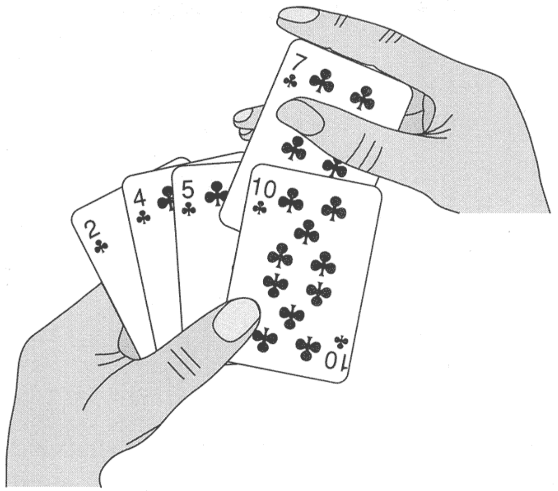
Julius-Maximilians-
UNIVERSITÄT
WÜRZBURG

Algorithmen und Datenstrukturen



Vorlesung 3: Laufzeitanalyse



Alexander Wolff

Wintersemester 2024

Recap

Recap

1. Was sind die zwei (oder drei?) entscheidensten Fragen, die wir uns über einen Algorithmus stellen?

Recap

1. Was sind die zwei (oder drei?) entscheidensten Fragen, die wir uns über einen Algorithmus stellen?

2. Warum eigentlich interessieren wir uns fürs Sortieren?

Recap

1. Was sind die zwei (oder drei?) entscheidensten Fragen,
   die wir uns über einen Algorithmus stellen?

2. Warum eigentlich interessieren wir uns fürs Sortieren?

3. Welche Entwurfstechniken für Algorithmen kennen wir schon?

Recap

1. Was sind die zwei (oder drei?) entscheidensten Fragen,
   die wir uns über einen Algorithmus stellen?

2. Warum eigentlich interessieren wir uns fürs Sortieren?

3. Welche Entwurfstechniken für Algorithmen kennen wir schon?

4. Wie beweisen wir die Korrektheit

   a) einer Schleife?

Recap

1. Was sind die zwei (oder drei?) entscheidensten Fragen, die wir uns über einen Algorithmus stellen?

2. Warum eigentlich interessieren wir uns fürs Sortieren?

3. Welche Entwurfstechniken für Algorithmen kennen wir schon?

4. Wie beweisen wir die Korrektheit

   a) einer Schleife?

   b) eines inkrementellen Algorithmus?

Recap

1. Was sind die zwei (oder drei?) entscheidensten Fragen, die wir uns über einen Algorithmus stellen?

2. Warum eigentlich interessieren wir uns fürs Sortieren?

3. Welche Entwurfstechniken für Algorithmen kennen wir schon?

4. Wie beweisen wir die Korrektheit

   a) einer Schleife?

   b) eines inkrementellen Algorithmus?

   c) eines rekursiven Algorithmus?

Laufzeit von InsertionSort

InsertionSort(int[] A)
  for $j = 2$ **to** $A.length$ **do**
    $key = A[j]$
    $i = j - 1$
    while $i > 0$ **and** $A[i] > key$ **do**
      $A[i+1] = A[i]$
      $i = i - 1$
    $A[i+1] = key$

Wie lang braucht dieser Algorithmus?

Laufzeit von InsertionSort

```
InsertionSort(int[] A)
  for j = 2 to A.length do
    key = A[j]
    i = j − 1
    while i > 0 and A[i] > key do
      A[i + 1] = A[i]
      i = i − 1
    A[i + 1] = key
```

Wie lang braucht dieser Algorithmus?

Wir zählen nur Vergleiche zwischen Schlüsseln.

Laufzeit von InsertionSort

```
InsertionSort(int[] A)
  for j = 2 to A.length do
      key = A[j]
      i = j − 1
      while i > 0 and A[i] > key do
          A[i + 1] = A[i]
          i = i − 1
      A[i + 1] = key
```

Wie lang braucht dieser Algorithmus?

Wir zählen nur Vergleiche zwischen Schlüsseln.

Laufzeit wird bzgl. der **Eingabegröße** n gemessen.

Laufzeit von InsertionSort

```
InsertionSort(int[] A)
  for j = 2 to A.length do
      key = A[j]
      i = j − 1
      while i > 0 and A[i] > key do
          A[i + 1] = A[i]
          i = i − 1
      A[i + 1] = key
```

Wie lang braucht dieser Algorithmus?

Wir zählen nur Vergleiche zwischen Schlüsseln.

Laufzeit wird bzgl. der **Eingabegröße** n gemessen.

Hier: $n = A.length$.

Laufzeit von InsertionSort

```
InsertionSort(int[] A)
  for j = 2 to A.length do
      key = A[j]
      i = j − 1
      while i > 0 and A[i] > key do
          A[i + 1] = A[i]
          i = i − 1
      A[i + 1] = key
```

Wie lang braucht dieser Algorithmus?

Wir zählen nur Vergleiche zwischen Schlüsseln.

Laufzeit wird bzgl. der **Eingabegröße** n gemessen.

Hier: $n = A.length$.

Antwort: kommt drauf an...

Laufzeit von InsertionSort

```
InsertionSort(int[] A)
  for j = 2 to A.length do
      key = A[j]
      i = j − 1
      while i > 0 and A[i] > key do
          A[i + 1] = A[i]
          i = i − 1
      A[i + 1] = key
```

Wie lang braucht dieser Algorithmus?

Wir zählen nur Vergleiche zwischen Schlüsseln.

Laufzeit wird bzgl. der **Eingabegröße** n gemessen.

Hier: $n = A.length$.

Antwort: kommt drauf an...

Bester Fall.

Schlechtester Fall.

Laufzeit von InsertionSort

InsertionSort(int[] A)
  for $j = 2$ **to** $A.length$ **do**
    $key = A[j]$
    $i = j - 1$
    while $i > 0$ **and** $A[i] > key$ **do**
      $A[i+1] = A[i]$
      $i = i - 1$
    $A[i+1] = key$

Wie lang braucht dieser Algorithmus?

Wir zählen nur Vergleiche zwischen Schlüsseln.

Laufzeit wird bzgl. der **Eingabegröße** n gemessen.

Hier: $n = A.length$.

Antwort: kommt drauf an...

Bester Fall.

Array ist bereits aufsteigend sortiert.

1	2	3	4	5	6	7	8	9

Schlechtester Fall.

Laufzeit von InsertionSort

```
InsertionSort(int[] A)
  for j = 2 to A.length do
    key = A[j]
    i = j − 1
    while i > 0 and A[i] > key do
      A[i + 1] = A[i]
      i = i − 1
    A[i + 1] = key
```

Wie lang braucht dieser Algorithmus?

Wir zählen nur Vergleiche zwischen Schlüsseln.

Laufzeit wird bzgl. der **Eingabegröße** n gemessen.

Hier: $n = A.length$.

Antwort: kommt drauf an...

Bester Fall.

Array ist bereits aufsteigend sortiert.

1	2	3	4	5	6	7	8	9

$\Rightarrow A[i] > key$ **nie** erfüllt.

Schlechtester Fall.

Laufzeit von InsertionSort

```
InsertionSort(int[] A)
  for j = 2 to A.length do
      key = A[j]
      i = j − 1
      while i > 0 and A[i] > key do
          A[i + 1] = A[i]
          i = i − 1
      A[i + 1] = key
```

Wie lang braucht dieser Algorithmus?

Wir zählen nur Vergleiche zwischen Schlüsseln.

Laufzeit wird bzgl. der **Eingabegröße** n gemessen.

Hier: $n = A.length$.

Antwort: kommt drauf an...

Bester Fall.

Array ist bereits aufsteigend sortiert.

1	2	3	4	5	6	7	8	9

$\Rightarrow$ $A[i] > key$ **nie** erfüllt.

$\Rightarrow$ Für jedes j nur 1 Vergleich.

Schlechtester Fall.

Laufzeit von InsertionSort

```
InsertionSort(int[] A)
  for j = 2 to A.length do
    key = A[j]
    i = j − 1
    while i > 0 and A[i] > key do
      A[i + 1] = A[i]
      i = i − 1
    A[i + 1] = key
```

Wie lang braucht dieser Algorithmus?

Wir zählen nur Vergleiche zwischen Schlüsseln.

Laufzeit wird bzgl. der **Eingabegröße** n gemessen.

Hier: $n = A.length$.

Antwort: kommt drauf an...

Bester Fall.

Array ist bereits aufsteigend sortiert.

1	2	3	4	5	6	7	8	9

$\Rightarrow$ $A[i] > key$ **nie** erfüllt.

$\Rightarrow$ Für jedes j nur 1 Vergleich.

$\Rightarrow$ Laufzeit: $n - 1$ Vergleiche

Schlechtester Fall.

Laufzeit von InsertionSort

$\textsc{InsertionSort}(\text{int}[]\ A)$
for $j = 2$ **to** $A.length$ **do**
  $key = A[j]$
  $i = j - 1$
  while $i > 0$ **and** $A[i] > key$ **do**
    $A[i+1] = A[i]$
    $i = i - 1$
  $A[i+1] = key$

Wie lang braucht dieser Algorithmus?

Wir zählen nur Vergleiche zwischen Schlüsseln.

Laufzeit wird bzgl. der **Eingabegröße** n gemessen.

Hier: $n = A.length$.

Antwort: kommt drauf an...

Analysieren meistens schlechtesten Fall

Bester Fall.

Array ist bereits aufsteigend sortiert.

1	2	3	4	5	6	7	8	9

$\Rightarrow$ $A[i] > key$ **nie** erfüllt.

$\Rightarrow$ Für jedes j nur 1 Vergleich.

$\Rightarrow$ Laufzeit: $n - 1$ Vergleiche

Schlechtester Fall.

Laufzeit von InsertionSort

```
InsertionSort(int[] A)
  for j = 2 to A.length do
      key = A[j]
      i = j − 1
      while i > 0 and A[i] > key do
          A[i + 1] = A[i]
          i = i − 1
      A[i + 1] = key
```

Wie lang braucht dieser Algorithmus?

Wir zählen nur Vergleiche zwischen Schlüsseln.

Laufzeit wird bzgl. der **Eingabegröße** n gemessen.

Hier: $n = A.length$.

Antwort: kommt drauf an...

Analysieren meistens schlechtesten Fall

Bester Fall.

Array ist bereits aufsteigend sortiert.

1	2	3	4	5	6	7	8	9

$\Rightarrow A[i] > key$ **nie** erfüllt.

$\Rightarrow$ Für jedes j nur 1 Vergleich.

$\Rightarrow$ Laufzeit: $n - 1$ Vergleiche

Schlechtester Fall.

Array ist absteigend sortiert.

9	8	7	6	5	4	3	2	1

Laufzeit von InsertionSort

```
InsertionSort(int[] A)
  for j = 2 to A.length do
    key = A[j]
    i = j − 1
    while i > 0 and A[i] > key do
      A[i + 1] = A[i]
      i = i − 1
    A[i + 1] = key
```

Wie lang braucht dieser Algorithmus?

Wir zählen nur Vergleiche zwischen Schlüsseln.

Laufzeit wird bzgl. der **Eingabegröße** n gemessen.

Hier: $n = A.length$.

Antwort: kommt drauf an...

Analysieren meistens schlechtesten Fall

Bester Fall.

Array ist bereits aufsteigend sortiert.

1	2	3	4	5	6	7	8	9

$\Rightarrow A[i] > key$ **nie** erfüllt.

$\Rightarrow$ Für jedes j nur 1 Vergleich.

$\Rightarrow$ Laufzeit: $n - 1$ Vergleiche

Schlechtester Fall.

Array ist absteigend sortiert.

9	8	7	6	5	4	3	2	1

$\Rightarrow A[i] > key$ **immer** erfüllt

Laufzeit von InsertionSort

```
InsertionSort(int[] A)
  for j = 2 to A.length do
    key = A[j]
    i = j − 1
    while i > 0 and A[i] > key do
      A[i + 1] = A[i]
      i = i − 1
    A[i + 1] = key
```

Wie lang braucht dieser Algorithmus?

Wir zählen nur Vergleiche zwischen Schlüsseln.

Laufzeit wird bzgl. der **Eingabegröße** n gemessen.

Hier: $n = A.length$.

Antwort: kommt drauf an...

Analysieren meistens schlechtesten Fall

Bester Fall.

Array ist bereits aufsteigend sortiert.

1	2	3	4	5	6	7	8	9

$\Rightarrow$ $A[i] > key$ **nie** erfüllt.

$\Rightarrow$ Für jedes j nur 1 Vergleich.

$\Rightarrow$ Laufzeit: $n - 1$ Vergleiche

Schlechtester Fall.

Array ist absteigend sortiert.

9	8	7	6	5	4	3	2	1

$\Rightarrow$ $A[i] > key$ **immer** erfüllt

$\Rightarrow$ Für jedes j gibt es $j - 1$ Vergleiche.

Laufzeit von InsertionSort

```
InsertionSort(int[] A)
  for j = 2 to A.length do
    key = A[j]
    i = j − 1
    while i > 0 and A[i] > key do
      A[i + 1] = A[i]
      i = i − 1
    A[i + 1] = key
```

Wie lang braucht dieser Algorithmus?

Wir zählen nur Vergleiche zwischen Schlüsseln.

Laufzeit wird bzgl. der **Eingabegröße** n gemessen.

Hier: $n = A.length$.

Antwort: kommt drauf an...

Analysieren meistens schlechtesten Fall

Bester Fall.

Array ist bereits aufsteigend sortiert.

1	2	3	4	5	6	7	8	9

$\Rightarrow A[i] > key$ **nie** erfüllt.

$\Rightarrow$ Für jedes j nur 1 Vergleich.

$\Rightarrow$ Laufzeit: $n - 1$ Vergleiche

Schlechtester Fall.

Array ist absteigend sortiert.

9	8	7	6	5	4	3	2	1

$\Rightarrow A[i] > key$ **immer** erfüllt (bis $i = 0$)

$\Rightarrow$ Für jedes j gibt es $j - 1$ Vergleiche.

Laufzeit von InsertionSort

```
InsertionSort(int[] A)
  for j = 2 to A.length do
    key = A[j]
    i = j − 1
    while i > 0 and A[i] > key do
      A[i + 1] = A[i]
      i = i − 1
    A[i + 1] = key
```

Wie lang braucht dieser Algorithmus?

Wir zählen nur Vergleiche zwischen Schlüsseln.

Laufzeit wird bzgl. der **Eingabegröße** n gemessen.

Hier: $n = A.length$.

Antwort: kommt drauf an...

Analysieren meistens schlechtesten Fall

Bester Fall.

Array ist bereits aufsteigend sortiert.

1	2	3	4	5	6	7	8	9

$\Rightarrow$ $A[i] > key$ **nie** erfüllt.

$\Rightarrow$ Für jedes j nur 1 Vergleich.

$\Rightarrow$ Laufzeit: $n - 1$ Vergleiche

Schlechtester Fall.

Array ist absteigend sortiert.

9	8	7	6	5	4	3	2	1

$\Rightarrow$ $A[i] > key$ **immer** erfüllt (bis $i = 0$)

$\Rightarrow$ Für jedes j gibt es $j - 1$ Vergleiche.

$\Rightarrow$ Laufzeit $\sum_{j=2}^{n}(j-1) =$

Laufzeit von InsertionSort

```
InsertionSort(int[] A)
  for j = 2 to A.length do
    key = A[j]
    i = j − 1
    while i > 0 and A[i] > key do
      A[i + 1] = A[i]
      i = i − 1
    A[i + 1] = key
```

Wie lang braucht dieser Algorithmus?

Wir zählen nur Vergleiche zwischen Schlüsseln.

Laufzeit wird bzgl. der **Eingabegröße** n gemessen.

Hier: $n = A.length$.

Antwort: kommt drauf an...

Analysieren meistens schlechtesten Fall

Bester Fall.

Array ist bereits aufsteigend sortiert.

1	2	3	4	5	6	7	8	9

$\Rightarrow$ $A[i] > key$ **nie** erfüllt.

$\Rightarrow$ Für jedes j nur 1 Vergleich.

$\Rightarrow$ Laufzeit: $n - 1$ Vergleiche

Schlechtester Fall.

Array ist absteigend sortiert.

9	8	7	6	5	4	3	2	1

$\Rightarrow$ $A[i] > key$ **immer** erfüllt (bis $i = 0$)

$\Rightarrow$ Für jedes j gibt es $j - 1$ Vergleiche.

$\Rightarrow$ Laufzeit $\sum_{j=2}^{n}(j-1) = \sum_{j=1}^{n-1} j =$

Laufzeit von InsertionSort

```
InsertionSort(int[] A)
  for j = 2 to A.length do
      key = A[j]
      i = j − 1
      while i > 0 and A[i] > key do
          A[i + 1] = A[i]
          i = i − 1
      A[i + 1] = key
```

Wie lang braucht dieser Algorithmus?

Wir zählen nur Vergleiche zwischen Schlüsseln.

Laufzeit wird bzgl. der **Eingabegröße** n gemessen.

Hier: $n = A.length$.

Antwort: kommt drauf an...

Analysieren meistens schlechtesten Fall

Bester Fall.

Array ist bereits aufsteigend sortiert.

1	2	3	4	5	6	7	8	9

$\Rightarrow$ $A[i] > key$ **nie** erfüllt.

$\Rightarrow$ Für jedes j nur 1 Vergleich.

$\Rightarrow$ Laufzeit: $n - 1$ Vergleiche

Schlechtester Fall.

Array ist absteigend sortiert.

9	8	7	6	5	4	3	2	1

$\Rightarrow$ $A[i] > key$ **immer** erfüllt (bis $i = 0$)

$\Rightarrow$ Für jedes j gibt es $j - 1$ Vergleiche.

$\Rightarrow$ Laufzeit $\sum_{j=2}^{n} (j-1) = \sum_{j=1}^{n-1} j =$

1) $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$ **arithmetische Reihe**

Laufzeit von InsertionSort

```
InsertionSort(int[] A)
  for j = 2 to A.length do
    key = A[j]
    i = j − 1
    while i > 0 and A[i] > key do
      A[i + 1] = A[i]
      i = i − 1
    A[i + 1] = key
```

Wie lang braucht dieser Algorithmus?

Wir zählen nur Vergleiche zwischen Schlüsseln.

Laufzeit wird bzgl. der **Eingabegröße** n gemessen.

Hier: $n = A.length$.

Antwort: kommt drauf an...

Analysieren meistens schlechtesten Fall

Bester Fall.

Array ist bereits aufsteigend sortiert.

1	2	3	4	5	6	7	8	9

$\Rightarrow$ $A[i] > key$ **nie** erfüllt.

$\Rightarrow$ Für jedes j nur 1 Vergleich.

$\Rightarrow$ Laufzeit: $n - 1$ Vergleiche

Schlechtester Fall.

Array ist absteigend sortiert.

9	8	7	6	5	4	3	2	1

$\Rightarrow$ $A[i] > key$ **immer** erfüllt (bis $i = 0$)

$\Rightarrow$ Für jedes j gibt es $j - 1$ Vergleiche.

$\Rightarrow$ Laufzeit $\sum_{j=2}^{n}(j-1) = \sum_{j=1}^{n-1} j = \frac{n(n-1)}{2}$

1) $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$ **arithmetische Reihe**

Laufzeit von MergeSort

MergeSort(int[] A, int $\ell = 1$, int $r = A.length$)

if $\ell < r$ **then**

$m = \lfloor (\ell + r)/2 \rfloor$ } **teile**

MergeSort(A, ℓ, m)
MergeSort($A, m+1, r$) } **herrsche**

Merge(A, ℓ, m, r) } **kombiniere**

Laufzeit von MergeSort

MergeSort(int[] A, int $\ell = 1$, int $r = A.length$)

if $\ell < r$ **then**

- $m = \lfloor(\ell + r)/2\rfloor$ } **teile**
- MergeSort(A, ℓ, m)
- MergeSort($A, m + 1, r$) } **herrsche**
- Merge(A, ℓ, m, r) } **kombiniere**

Korrekt?

Laufzeit von MergeSort

$\textsc{MergeSort}(\text{int}[]\ A,\ \text{int}\ \ell = 1,\ \text{int}\ r = A.length)$

if $\ell < r$ **then**

$m = \lfloor (\ell + r)/2 \rfloor$ } **teile**

$\textsc{MergeSort}(A, \ell, m)$
$\textsc{MergeSort}(A, m+1, r)$ } **herrsche**

$\textsc{Merge}(A, \ell, m, r)$ } **kombiniere**

Korrekt? ✓

Laufzeit von MergeSort

$\textsc{MergeSort}(\text{int}[\,]\ A,\ \text{int}\ \ell = 1,\ \text{int}\ r = A.\mathit{length})$

if $\ell < r$ **then**

$m = \lfloor (\ell + r)/2 \rfloor$ } **teile**

$\textsc{MergeSort}(A, \ell, m)$

$\textsc{MergeSort}(A, m+1, r)$ } **herrsche**

$\textsc{Merge}(A, \ell, m, r)$ } **kombiniere**

Korrekt? ✓

Effizient?

Laufzeit von MergeSort

```
MergeSort(int[] A, int ℓ = 1, int r = A.length)
if ℓ < r then
    m = ⌊(ℓ + r)/2⌋                  } teile
    MergeSort(A, ℓ, m)               } herrsche
    MergeSort(A, m + 1, r)           }
    Merge(A, ℓ, m, r)                } kombiniere
```

Effizient?

Sei $V_{\mathrm{MS}}(n)$ die maximale Anzahl von Vergleichen, die MergeSort zum Sortieren von n Zahlen benötigt.

Laufzeit von MergeSort

MergeSort(int[] A, int $\ell = 1$, int $r = A.length$)
if $\ell < r$ **then**
  $m = \lfloor (\ell + r)/2 \rfloor$ } **teile**
  MergeSort(A, ℓ, m)
  MergeSort($A, m+1, r$) } **herrsche**
  Merge(A, ℓ, m, r) } **kombiniere**

Effizient? Sei $V_{\mathrm{MS}}(n)$ die maximale Anzahl von Vergleichen, die MergeSort zum Sortieren von n Zahlen benötigt.

Dann gilt

$$V_{\mathrm{MS}}(n) = \begin{cases} & \text{falls } n = 1, \\ & \text{sonst.} \end{cases}$$

Laufzeit von MergeSort

$\textsc{MergeSort}(\text{int}[]\ A, \text{int}\ \ell = 1, \text{int}\ r = A.length)$

if $\ell < r$ **then**

$m = \lfloor(\ell + r)/2\rfloor$ } **teile**

$\textsc{MergeSort}(A, \ell, m)$
$\textsc{MergeSort}(A, m+1, r)$ } **herrsche**

$\textsc{Merge}(A, \ell, m, r)$ } **kombiniere**

Effizient?

Sei $V_{\text{MS}}(n)$ die maximale Anzahl von Vergleichen, die MergeSort zum Sortieren von n Zahlen benötigt.

Dann gilt

$$V_{\text{MS}}(n) = \begin{cases} 0 & \text{falls } n = 1, \\ & \text{sonst.} \end{cases}$$

Laufzeit von MergeSort

```
MergeSort(int[] A, int ℓ = 1, int r = A.length)
 if ℓ < r then
   m = ⌊(ℓ + r)/2⌋                    } teile
   MergeSort(A, ℓ, m)                 } herrsche
   MergeSort(A, m + 1, r)
   Merge(A, ℓ, m, r)                  } kombiniere
```

Effizient? Sei $V_{\mathrm{MS}}(n)$ die maximale Anzahl von Vergleichen, die MergeSort zum Sortieren von n Zahlen benötigt.

Dann gilt

$$V_{\mathrm{MS}}(n) = \begin{cases} 0 & \text{falls } n = 1, \\ + & \text{sonst.} \end{cases}$$

Laufzeit von MergeSort

$\textsc{MergeSort}(\text{int}[]\ A, \text{int}\ \ell = 1, \text{int}\ r = A.\mathit{length})$
if $\ell < r$ **then**
  $m = \lfloor (\ell + r)/2 \rfloor$ } **teile**
  $\textsc{MergeSort}(A, \ell, m)$
  $\textsc{MergeSort}(A, m+1, r)$ } **herrsche**
  $\textsc{Merge}(A, \ell, m, r)$ } **kombiniere**

Effizient? Sei $V_{\text{MS}}(n)$ die maximale Anzahl von Vergleichen, die MergeSort zum Sortieren von n Zahlen benötigt.

Dann gilt

$$V_{\text{MS}}(n) = \begin{cases} 0 & \text{falls } n = 1, \\ 2V_{\text{MS}}(n/2) + \boxed{} & \text{sonst.} \end{cases}$$

Laufzeit von MergeSort

MergeSort(int[] A, int $\ell = 1$, int $r = A.length$)
if $\ell < r$ **then**
  $m = \lfloor(\ell + r)/2\rfloor$ } **teile**
  MergeSort(A, ℓ, m)
  MergeSort($A, m+1, r$) } **herrsche**
  Merge(A, ℓ, m, r) } **kombiniere**

Effizient? Sei $V_{\mathrm{MS}}(n)$ die maximale Anzahl von Vergleichen, die MergeSort zum Sortieren von n Zahlen benötigt.

Dann gilt

$$V_{\mathrm{MS}}(n) = \begin{cases} 0 & \text{falls } n = 1, \\ \boxed{2V_{\mathrm{MS}}(n/2)} + \boxed{n} & \text{sonst.} \end{cases}$$

Laufzeit von MergeSort

$\textsc{MergeSort}(\text{int}[\,]\ A, \text{int}\ \ell = 1, \text{int}\ r = A.\mathit{length})$

if $\ell < r$ **then**

$m = \lfloor(\ell + r)/2\rfloor$ } **teile**

$\textsc{MergeSort}(A, \ell, m)$

$\textsc{MergeSort}(A, m+1, r)$ } **herrsche**

$\textsc{Merge}(A, \ell, m, r)$ } **kombiniere**

Effizient? Sei $V_{\text{MS}}(n)$ die maximale Anzahl von Vergleichen, die MergeSort zum Sortieren von n Zahlen benötigt.

Dann gilt

$$V_{\text{MS}}(n) = \left.\begin{cases} 0 & \text{falls } n = 1, \\ 2V_{\text{MS}}(n/2) + n & \text{sonst.} \end{cases}\right\} = \textbf{?}$$

Laufzeit von MergeSort

$\textsc{MergeSort}(\text{int}[\,]\ A,\ \text{int}\ \ell = 1,\ \text{int}\ r = A.length)$
if $\ell < r$ **then**
$\quad m = \lfloor (\ell + r)/2 \rfloor$ } **teile**
$\quad \textsc{MergeSort}(A, \ell, m)$
$\quad \textsc{MergeSort}(A, m+1, r)$ } **herrsche**
$\quad \textsc{Merge}(A, \ell, m, r)$ } **kombiniere**

Effizient?

Sei $V_{\text{MS}}(n)$ die maximale Anzahl von Vergleichen, die MergeSort zum Sortieren von n Zahlen benötigt.

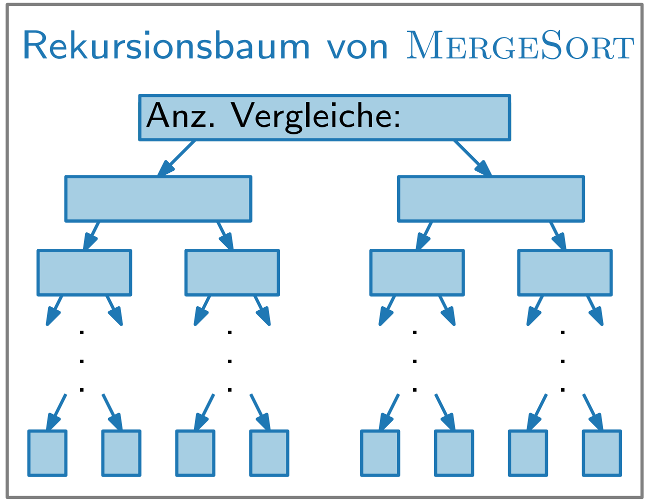
Dann gilt

$$V_{\text{MS}}(n) = \begin{cases} 0 & \text{falls } n = 1, \\ 2V_{\text{MS}}(n/2) + n & \text{sonst.} \end{cases} = \ ?$$

falls n Zweierpotenz

Laufzeit von MergeSort

MergeSort(int[] A, int $\ell = 1$, int $r = A.length$)
if $\ell < r$ **then**
- $m = \lfloor(\ell + r)/2\rfloor$ } **teile**
- MergeSort(A, ℓ, m)
- MergeSort($A, m+1, r$) } **herrsche**
- Merge(A, ℓ, m, r) } **kombiniere**



Effizient?

Sei $V_{\mathrm{MS}}(n)$ die maximale Anzahl von Vergleichen, die MergeSort zum Sortieren von n Zahlen benötigt.

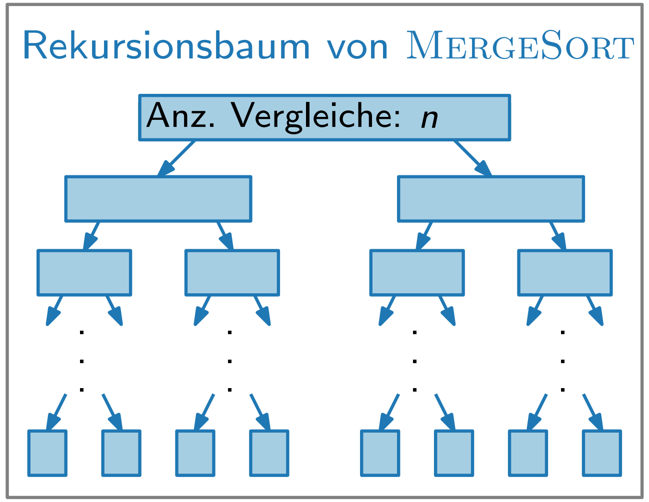
Dann gilt

$$V_{\mathrm{MS}}(n) = \left\{\begin{array}{ll} 0 & \text{falls } n = 1, \\ 2V_{\mathrm{MS}}(n/2) + n & \text{sonst.} \end{array}\right\} = ?$$

falls n Zweierpotenz

Laufzeit von MergeSort

MergeSort(int[] A, int $\ell = 1$, int $r = A.length$)
if $\ell < r$ **then**
  $m = \lfloor (\ell + r)/2 \rfloor$ } **teile**
  MergeSort(A, ℓ, m)
  MergeSort($A, m+1, r$) } **herrsche**
  Merge(A, ℓ, m, r) } **kombiniere**



Effizient?

Sei $V_{\text{MS}}(n)$ die maximale Anzahl von Vergleichen, die MergeSort zum Sortieren von n Zahlen benötigt.

Dann gilt

$$V_{\text{MS}}(n) = \left.\begin{cases} 0 & \text{falls } n = 1, \\ 2V_{\text{MS}}(n/2) + n & \text{sonst.} \end{cases}\right\} = \textbf{?}$$

falls n Zweierpotenz

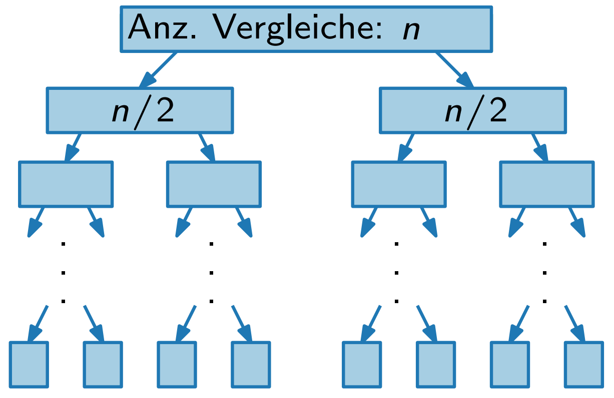
Laufzeit von MergeSort

MergeSort(int[] A, int $\ell = 1$, int $r = A.length$)

if $\ell < r$ **then**

- $m = \lfloor (\ell + r)/2 \rfloor$ } **teile**
- MergeSort(A, ℓ, m)
- MergeSort($A, m+1, r$) } **herrsche**
- Merge(A, ℓ, m, r) } **kombiniere**

Rekursionsbaum von MergeSort



Effizient?

Sei $V_{\mathrm{MS}}(n)$ die maximale Anzahl von Vergleichen, die MergeSort zum Sortieren von n Zahlen benötigt.

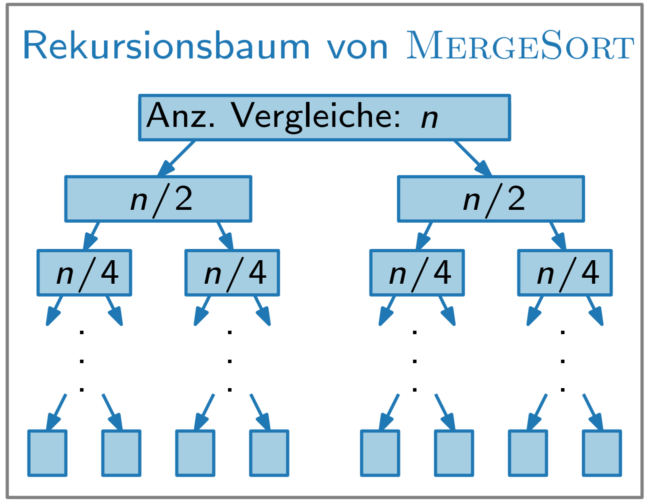
Dann gilt

$$V_{\mathrm{MS}}(n) = \left.\begin{cases} 0 & \text{falls } n = 1, \\ 2V_{\mathrm{MS}}(n/2) + n & \text{sonst.} \end{cases}\right\} = \textbf{?}$$

falls n Zweierpotenz

Laufzeit von MergeSort

$\textsc{MergeSort}(\text{int}[]\ A, \text{int}\ \ell = 1, \text{int}\ r = A.length)$
if $\ell < r$ **then**
$\quad m = \lfloor (\ell + r)/2 \rfloor$ } **teile**
$\quad \textsc{MergeSort}(A, \ell, m)$
$\quad \textsc{MergeSort}(A, m+1, r)$ } **herrsche**
$\quad \textsc{Merge}(A, \ell, m, r)$ } **kombiniere**



Effizient? Sei $V_{\text{MS}}(n)$ die maximale Anzahl von Vergleichen, die MergeSort zum Sortieren von n Zahlen benötigt.

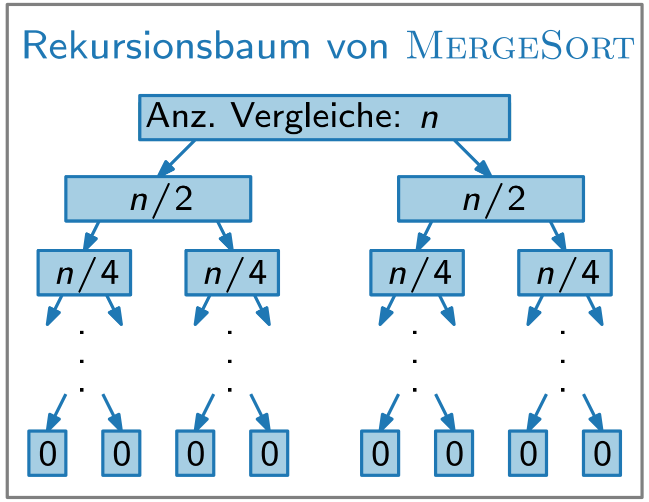
Dann gilt

$$V_{\text{MS}}(n) = \left.\begin{cases} 0 & \text{falls } n = 1, \\ 2V_{\text{MS}}(n/2) + n & \text{sonst.} \end{cases}\right\} = \textbf{?}$$

falls n Zweierpotenz

Laufzeit von MergeSort

MergeSort(int[] A, int $\ell = 1$, int $r = A.length$)
if $\ell < r$ **then**
  $m = \lfloor (\ell + r)/2 \rfloor$ } **teile**
  MergeSort(A, ℓ, m)
  MergeSort($A, m+1, r$) } **herrsche**
  Merge(A, ℓ, m, r) } **kombiniere**



Effizient?

Sei $V_{\mathrm{MS}}(n)$ die maximale Anzahl von Vergleichen, die MergeSort zum Sortieren von n Zahlen benötigt.

Dann gilt

$$V_{\mathrm{MS}}(n) = \begin{cases} 0 & \text{falls } n = 1, \\ 2V_{\mathrm{MS}}(n/2) + n & \text{sonst.} \end{cases} = \mathbf{?}$$

falls n Zweierpotenz

Laufzeit von MergeSort

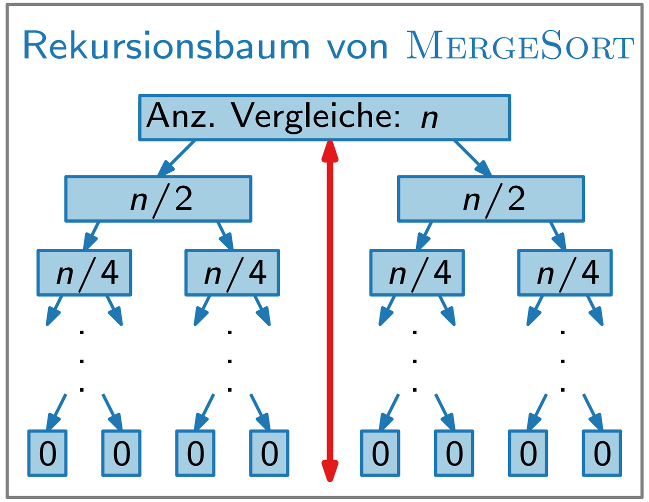
$\textsc{MergeSort}(\text{int}[]\ A,\ \text{int}\ \ell = 1,\ \text{int}\ r = A.\mathit{length})$

if $\ell < r$ **then**

$m = \lfloor (\ell + r)/2 \rfloor$ } **teile**

$\textsc{MergeSort}(A, \ell, m)$
$\textsc{MergeSort}(A, m+1, r)$ } **herrsche**

$\textsc{Merge}(A, \ell, m, r)$ } **kombiniere**



Effizient?

Sei $V_{\mathrm{MS}}(n)$ die maximale Anzahl von Vergleichen, die MergeSort zum Sortieren von n Zahlen benötigt.

Dann gilt

$$V_{\mathrm{MS}}(n) = \begin{cases} 0 & \text{falls } n = 1, \\ 2V_{\mathrm{MS}}(n/2) + n & \text{sonst.} \end{cases} = \mathbf{?}$$

falls n Zweierpotenz

Laufzeit von MergeSort

$\textsc{MergeSort}(\text{int}[\,]\ A,\ \text{int}\ \ell = 1,\ \text{int}\ r = A.\mathit{length})$

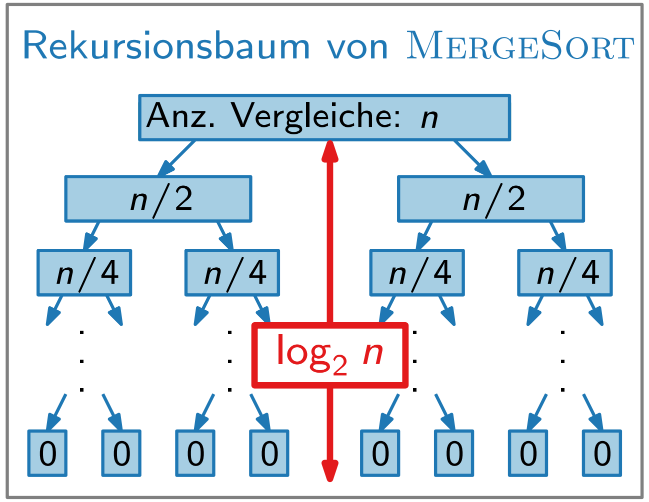
if $\ell < r$ **then**

$m = \lfloor (\ell + r)/2 \rfloor$ } **teile**

$\textsc{MergeSort}(A, \ell, m)$

$\textsc{MergeSort}(A, m+1, r)$ } **herrsche**

$\textsc{Merge}(A, \ell, m, r)$ } **kombiniere**



Effizient?

Sei $V_{\mathrm{MS}}(n)$ die maximale Anzahl von Vergleichen, die MergeSort zum Sortieren von n Zahlen benötigt.

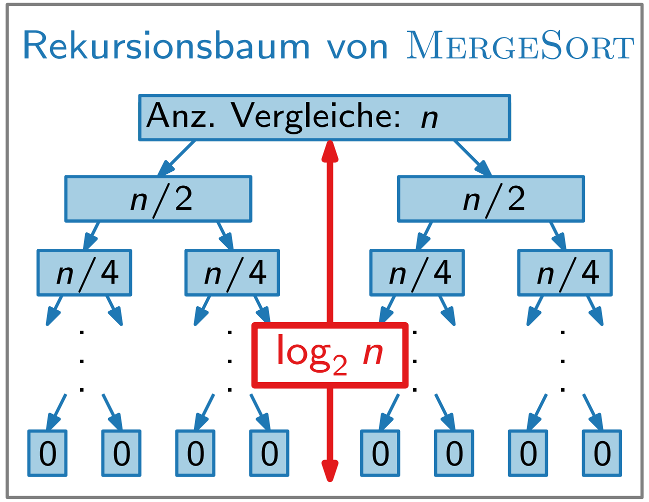
Dann gilt

$$V_{\mathrm{MS}}(n) = \left.\begin{cases} 0 & \text{falls } n = 1, \\ 2V_{\mathrm{MS}}(n/2) + n & \text{sonst.} \end{cases}\right\} = \ ?$$

falls n Zweierpotenz

Laufzeit von MergeSort

$\textsc{MergeSort}(\text{int}[]\ A, \text{int}\ \ell = 1, \text{int}\ r = A.\mathit{length})$
if $\ell < r$ **then**
 $m = \lfloor(\ell + r)/2\rfloor$ } **teile**
 $\textsc{MergeSort}(A, \ell, m)$
 $\textsc{MergeSort}(A, m+1, r)$ } **herrsche**
 $\textsc{Merge}(A, \ell, m, r)$ } **kombiniere**



Effizient? Sei $V_{\mathrm{MS}}(n)$ die maximale Anzahl von Vergleichen, die MergeSort zum Sortieren von n Zahlen benötigt.

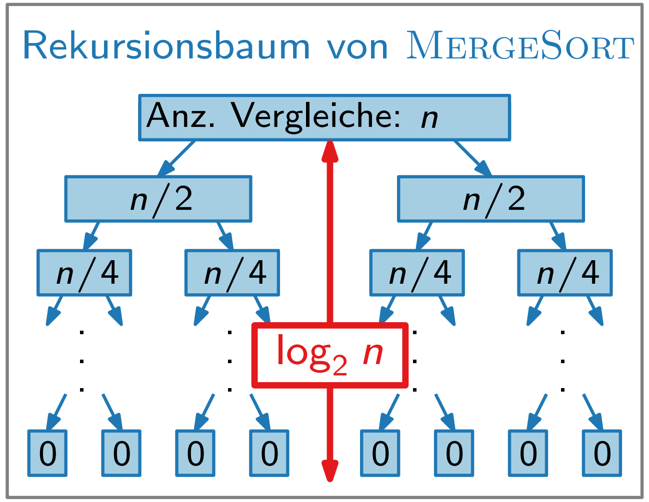
Dann gilt

$$V_{\mathrm{MS}}(n) = \begin{cases} 0 & \text{falls } n = 1, \\ 2V_{\mathrm{MS}}(n/2) + n & \text{sonst.} \end{cases} = n \cdot \log_2 n$$

falls n Zweierpotenz

Laufzeit von MergeSort

MergeSort(int[] A, int $\ell = 1$, int $r = A.length$)
if $\ell < r$ **then**
  $m = \lfloor (\ell + r)/2 \rfloor$ } **teile**
  MergeSort(A, ℓ, m)
  MergeSort($A, m+1, r$) } **herrsche**
  Merge(A, ℓ, m, r) } **kombiniere**



Effizient? Sei $V_{\mathrm{MS}}(n)$ die maximale Anzahl von Vergleichen, die MergeSort zum Sortieren von n Zahlen benötigt.

Dann gilt

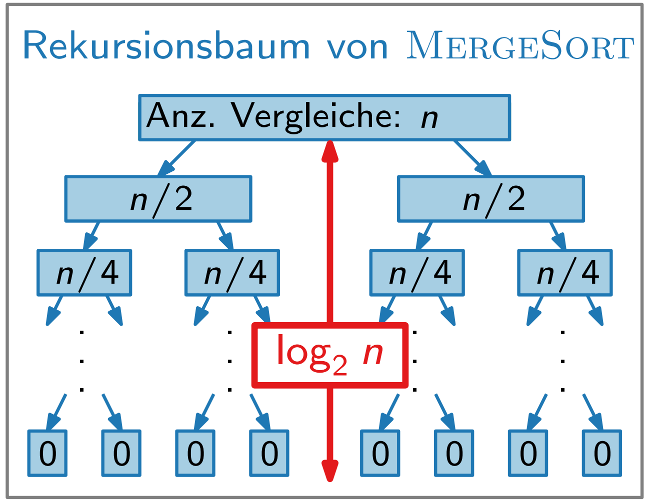
$$V_{\mathrm{MS}}(n) = \begin{cases} 0 & \text{falls } n = 1, \\ 2V_{\mathrm{MS}}(n/2) + n & \text{sonst.} \end{cases} = n \cdot \log_2 n$$

falls n Zweierpotenz

Beweis.

Laufzeit von MergeSort

MergeSort(int[] A, int $\ell = 1$, int $r = A.length$)
if $\ell < r$ **then**
  $m = \lfloor (\ell + r)/2 \rfloor$ } **teile**
  MergeSort(A, ℓ, m)
  MergeSort($A, m+1, r$) } **herrsche**
  Merge(A, ℓ, m, r) } **kombiniere**



Effizient? Sei $V_{\mathrm{MS}}(n)$ die maximale Anzahl von Vergleichen, die MergeSort zum Sortieren von n Zahlen benötigt.

Dann gilt

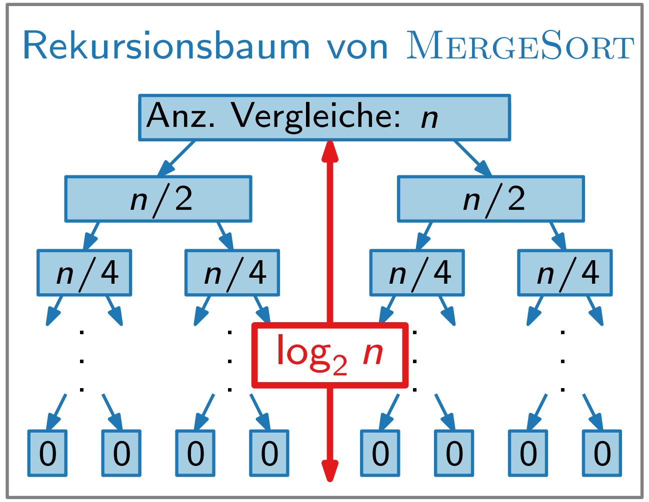
$$V_{\mathrm{MS}}(n) = \left\{ \begin{array}{ll} 0 & \text{falls } n = 1, \\ 2V_{\mathrm{MS}}(n/2) + n & \text{sonst.} \end{array} \right\} = n \cdot \log_2 n$$

falls n Zweierpotenz

Beweis. Per Induktion über n.

Laufzeit von MergeSort

MergeSort(int[] A, int $\ell = 1$, int $r = A.length$)
if $\ell < r$ **then**
  $m = \lfloor (\ell + r)/2 \rfloor$ } **teile**
  MergeSort(A, ℓ, m)
  MergeSort($A, m+1, r$) } **herrsche**
  Merge(A, ℓ, m, r) } **kombiniere**



Effizient? Sei $V_{\mathrm{MS}}(n)$ die maximale Anzahl von Vergleichen, die MergeSort zum Sortieren von n Zahlen benötigt.

Dann gilt

$$V_{\mathrm{MS}}(n) = \begin{cases} 0 & \text{falls } n = 1, \\ 2V_{\mathrm{MS}}(n/2) + n & \text{sonst.} \end{cases} = n \cdot \log_2 n$$

falls n Zweierpotenz

Beweis. Per Induktion über n.

Induktionsanfang:

Laufzeit von MergeSort

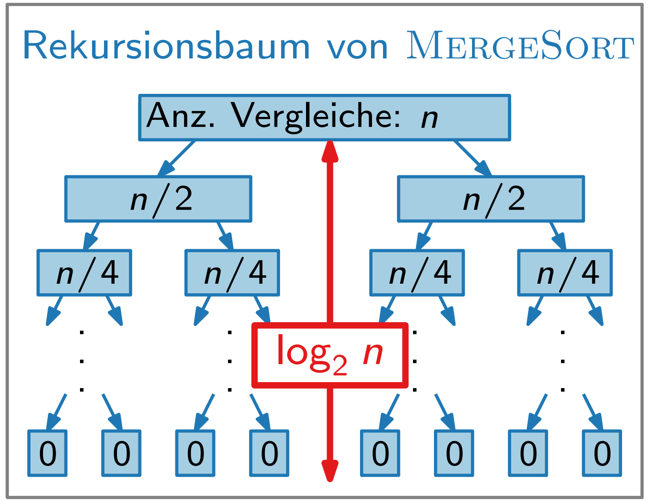
$\textsc{MergeSort}(\text{int}[\,]\ A,\ \text{int}\ \ell = 1,\ \text{int}\ r = A.length)$

if $\ell < r$ **then**

$m = \lfloor (\ell + r)/2 \rfloor$ } **teile**

$\textsc{MergeSort}(A, \ell, m)$
$\textsc{MergeSort}(A, m+1, r)$ } **herrsche**

$\textsc{Merge}(A, \ell, m, r)$ } **kombiniere**



Effizient? Sei $V_{\text{MS}}(n)$ die maximale Anzahl von Vergleichen, die MergeSort zum Sortieren von n Zahlen benötigt.

Dann gilt

$$V_{\text{MS}}(n) = \begin{cases} 0 & \text{falls } n = 1, \\ 2V_{\text{MS}}(n/2) + n & \text{sonst.} \end{cases} = n \cdot \log_2 n$$

falls n Zweierpotenz

Beweis. Per Induktion über n.

Induktionsanfang: $V_{\text{MS}}(1) =$

Laufzeit von MergeSort

MergeSort(int[] A, int $\ell = 1$, int $r = A.length$)

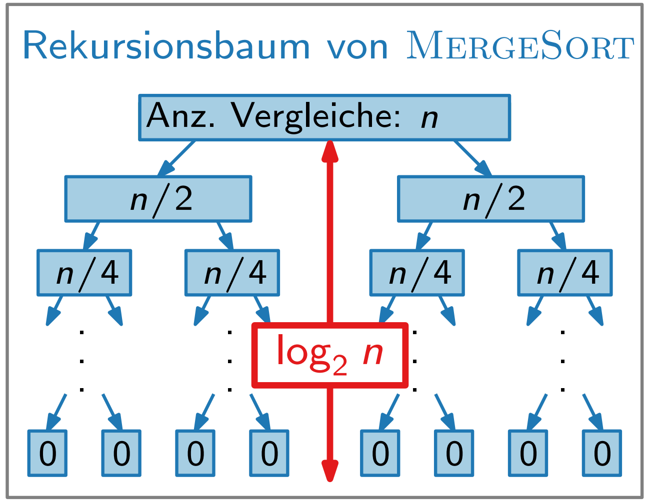
if $\ell < r$ **then**

$m = \lfloor (\ell + r)/2 \rfloor$ } **teile**

MergeSort(A, ℓ, m)

MergeSort($A, m+1, r$) } **herrsche**

Merge(A, ℓ, m, r) } **kombiniere**



Effizient? Sei $V_{\mathrm{MS}}(n)$ die maximale Anzahl von Vergleichen, die MergeSort zum Sortieren von n Zahlen benötigt.

Dann gilt

$$V_{\mathrm{MS}}(n) = \begin{cases} 0 & \text{falls } n = 1, \\ 2V_{\mathrm{MS}}(n/2) + n & \text{sonst.} \end{cases} = n \cdot \log_2 n$$

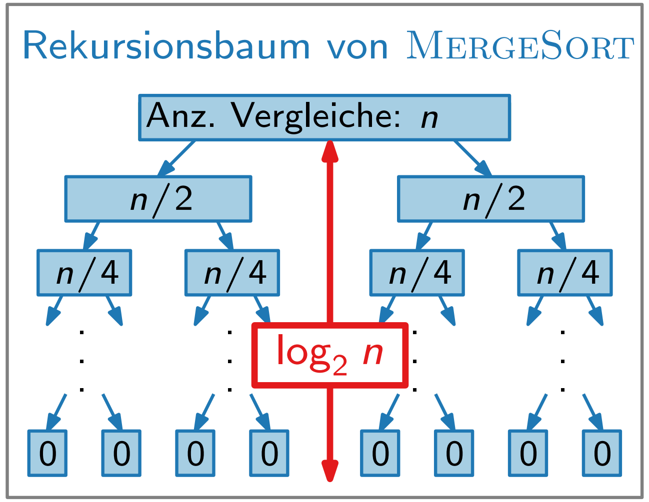
falls n Zweierpotenz

Beweis. Per Induktion über n.

Induktionsanfang: $V_{\mathrm{MS}}(1) = \; 1 \cdot \log_2 1 =$

Laufzeit von MergeSort

MergeSort(int[] A, int $\ell = 1$, int $r = A.length$)
if $\ell < r$ **then**
  $m = \lfloor (\ell + r)/2 \rfloor$ } **teile**
  MergeSort(A, ℓ, m)
  MergeSort($A, m+1, r$) } **herrsche**
  Merge(A, ℓ, m, r) } **kombiniere**



Effizient? Sei $V_{\mathrm{MS}}(n)$ die maximale Anzahl von Vergleichen, die MergeSort zum Sortieren von n Zahlen benötigt.

Dann gilt

$$V_{\mathrm{MS}}(n) = \left.\begin{cases} 0 & \text{falls } n = 1, \\ 2V_{\mathrm{MS}}(n/2) + n & \text{sonst.} \end{cases}\right\} = n \cdot \log_2 n$$

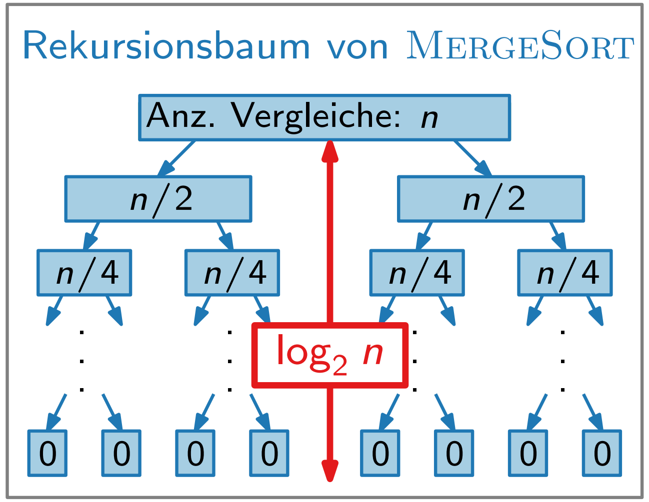
falls n Zweierpotenz

Beweis. Per Induktion über n.

Induktionsanfang: $V_{\mathrm{MS}}(1) = 1 \cdot \log_2 1 = 0$ ✓

Laufzeit von MergeSort

MergeSort(int[] A, int $\ell = 1$, int $r = A.length$)
if $\ell < r$ **then**
  $m = \lfloor (\ell + r)/2 \rfloor$ } **teile**
  MergeSort(A, ℓ, m)
  MergeSort($A, m+1, r$) } **herrsche**
  Merge(A, ℓ, m, r) } **kombiniere**



Effizient? Sei $V_{\mathrm{MS}}(n)$ die maximale Anzahl von Vergleichen, die MergeSort zum Sortieren von n Zahlen benötigt.

Dann gilt

$$V_{\mathrm{MS}}(n) = \left.\begin{cases} 0 & \text{falls } n = 1, \\ 2V_{\mathrm{MS}}(n/2) + n & \text{sonst.} \end{cases}\right\} = n \cdot \log_2 n$$

falls n Zweierpotenz

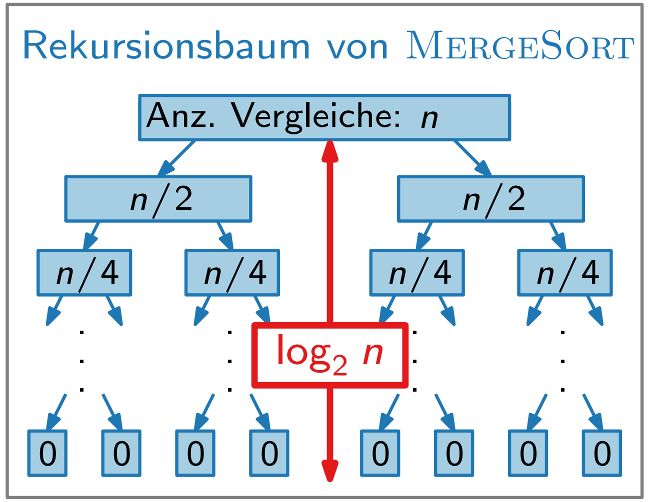
Beweis. Per Induktion über n.

Induktionsanfang: $V_{\mathrm{MS}}(1) = 1 \cdot \log_2 1 = 0$ ✓

Induktionsschritt: $V_{\mathrm{MS}}(n) \underset{n>1}{=}$

Laufzeit von MergeSort

MergeSort(int[] A, int $\ell = 1$, int $r = A.length$)
if $\ell < r$ **then**
  $m = \lfloor(\ell + r)/2\rfloor$ } **teile**
  MergeSort(A, ℓ, m)
  MergeSort($A, m+1, r$) } **herrsche**
  Merge(A, ℓ, m, r) } **kombiniere**



Effizient? Sei $V_{\mathrm{MS}}(n)$ die maximale Anzahl von Vergleichen, die MergeSort zum Sortieren von n Zahlen benötigt.

Dann gilt

$$V_{\mathrm{MS}}(n) = \left.\begin{cases} 0 & \text{falls } n = 1, \\ 2V_{\mathrm{MS}}(n/2) + n & \text{sonst.} \end{cases}\right\} = n \cdot \log_2 n$$

falls n Zweierpotenz

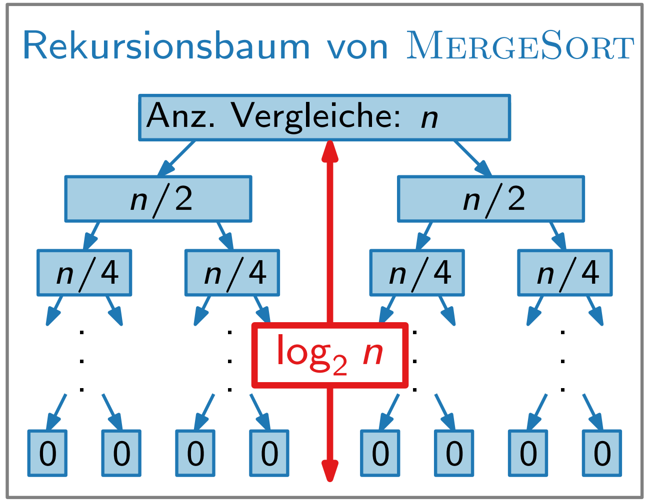
Beweis. Per Induktion über n.

Induktionsanfang: $V_{\mathrm{MS}}(1) = 1 \cdot \log_2 1 = 0$ ✓

Induktionsschritt: $V_{\mathrm{MS}}(n) \underset{n>1}{=} 2V_{\mathrm{MS}}(\frac{n}{2}) + n =$

Laufzeit von MergeSort

MergeSort(int[] A, int $\ell = 1$, int $r = A.length$)
if $\ell < r$ **then**
  $m = \lfloor(\ell + r)/2\rfloor$ } **teile**
  MergeSort(A, ℓ, m)
  MergeSort($A, m+1, r$) } **herrsche**
  Merge(A, ℓ, m, r) } **kombiniere**



Effizient? Sei $V_{\mathrm{MS}}(n)$ die maximale Anzahl von Vergleichen, die MergeSort zum Sortieren von n Zahlen benötigt.

Dann gilt

$$V_{\mathrm{MS}}(n) = \left.\begin{cases} 0 & \text{falls } n = 1, \\ 2V_{\mathrm{MS}}(n/2) + n & \text{sonst.} \end{cases}\right\} = n \cdot \log_2 n$$

falls n Zweierpotenz

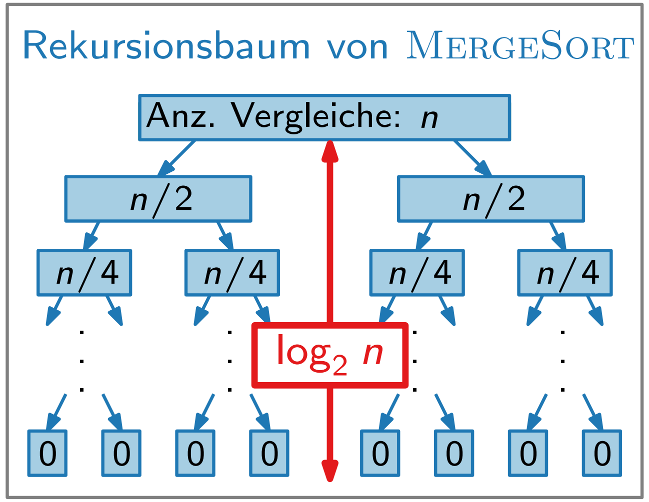
Beweis. Per Induktion über n.

Induktionsanfang: $V_{\mathrm{MS}}(1) = 1 \cdot \log_2 1 = 0$ ✓

Induktionsschritt: $V_{\mathrm{MS}}(n) \underset{n>1}{=} 2V_{\mathrm{MS}}(\frac{n}{2}) + n = 2 \cdot \frac{n}{2} \log_2 \frac{n}{2} + n =$

Laufzeit von MergeSort

$\textsc{MergeSort}(\text{int}[]\ A,\ \text{int}\ \ell = 1,\ \text{int}\ r = A.\mathit{length})$
if $\ell < r$ **then**
 $m = \lfloor (\ell + r)/2 \rfloor$ } **teile**
 $\textsc{MergeSort}(A, \ell, m)$
 $\textsc{MergeSort}(A, m+1, r)$ } **herrsche**
 $\textsc{Merge}(A, \ell, m, r)$ } **kombiniere**



Effizient? Sei $V_{\text{MS}}(n)$ die maximale Anzahl von Vergleichen, die MergeSort zum Sortieren von n Zahlen benötigt.

Dann gilt

$$V_{\text{MS}}(n) = \left.\begin{cases} 0 & \text{falls } n = 1, \\ 2V_{\text{MS}}(n/2) + n & \text{sonst.} \end{cases}\right\} = n \cdot \log_2 n$$

falls n Zweierpotenz

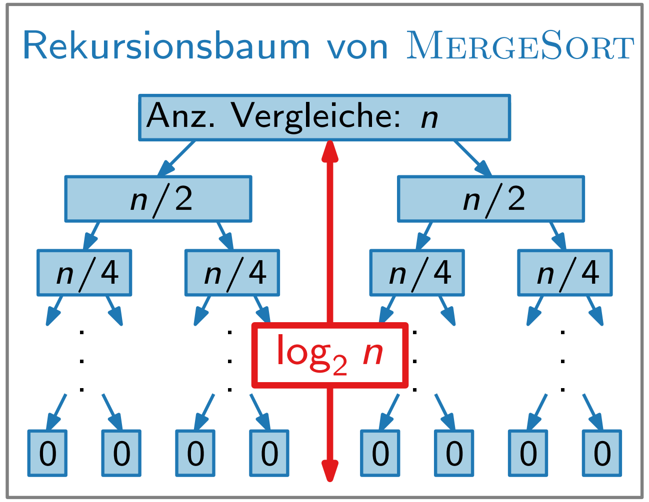
Beweis. Per Induktion über n.

Induktionsanfang: $V_{\text{MS}}(1) = 1 \cdot \log_2 1 = 0$ ✓

Induktionsschritt: $V_{\text{MS}}(n) \underset{n>1}{=} 2V_{\text{MS}}(\frac{n}{2}) + n = 2 \cdot \frac{n}{2} \log_2 \frac{n}{2} + n = n \cdot (\log_2 n - \log_2 2) + n$

Laufzeit von MergeSort

MergeSort(int[] A, int $\ell = 1$, int $r = A.length$)
if $\ell < r$ **then**
  $m = \lfloor(\ell + r)/2\rfloor$ } **teile**
  MergeSort(A, ℓ, m)
  MergeSort($A, m+1, r$) } **herrsche**
  Merge(A, ℓ, m, r) } **kombiniere**



Effizient? Sei $V_{\mathrm{MS}}(n)$ die maximale Anzahl von Vergleichen, die MergeSort zum Sortieren von n Zahlen benötigt.

Dann gilt

$$V_{\mathrm{MS}}(n) = \left.\begin{cases} 0 & \text{falls } n = 1, \\ 2V_{\mathrm{MS}}(n/2) + n & \text{sonst.} \end{cases}\right\} = n \cdot \log_2 n$$

falls n Zweierpotenz

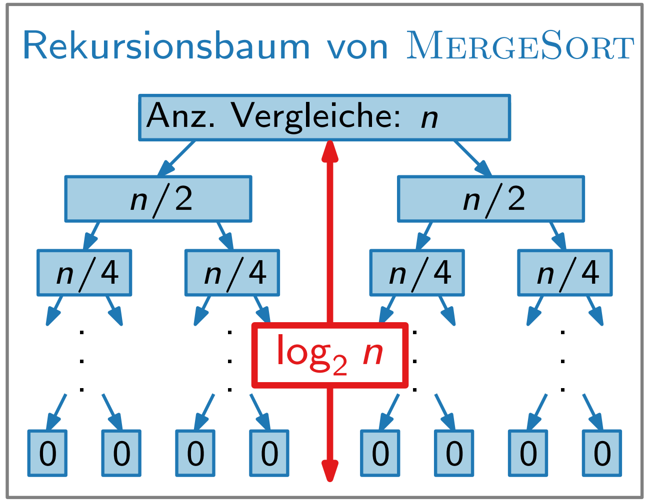
Beweis. Per Induktion über n.

Induktionsanfang: $V_{\mathrm{MS}}(1) = 1 \cdot \log_2 1 = 0$ ✓

Induktionsschritt: $V_{\mathrm{MS}}(n) \underset{n>1}{=} 2V_{\mathrm{MS}}(\frac{n}{2}) + n = 2 \cdot \frac{n}{2} \log_2 \frac{n}{2} + n = n \cdot (\log_2 n - \overbrace{\log_2 2}^{1}) + n$

Laufzeit von MergeSort

MergeSort(int[] A, int $\ell = 1$, int $r = A.length$)
if $\ell < r$ **then**
$m = \lfloor (\ell + r)/2 \rfloor$ } **teile**
MergeSort(A, ℓ, m)
MergeSort($A, m+1, r$) } **herrsche**
Merge(A, ℓ, m, r) } **kombiniere**



Effizient? Sei $V_{MS}(n)$ die maximale Anzahl von Vergleichen, die MergeSort zum Sortieren von n Zahlen benötigt.

Dann gilt

$$V_{MS}(n) = \left.\begin{cases} 0 & \text{falls } n = 1, \\ 2V_{MS}(n/2) + n & \text{sonst.} \end{cases}\right\} = n \cdot \log_2 n$$

falls n Zweierpotenz

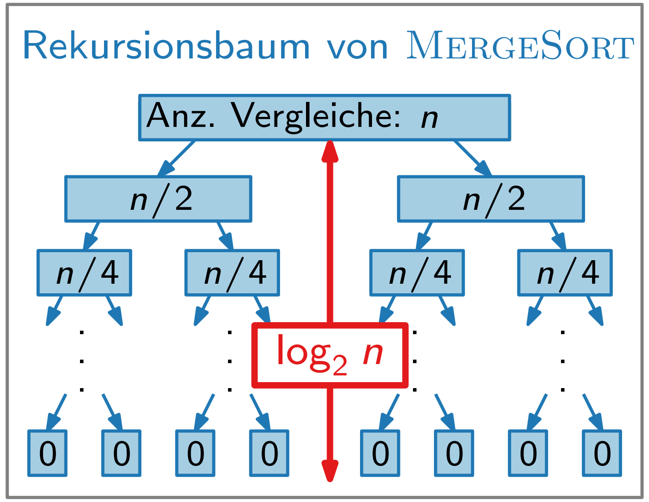
Beweis. Per Induktion über n.

Induktionsanfang: $V_{MS}(1) = 1 \cdot \log_2 1 = 0$ ✓

Induktionsschritt: $V_{MS}(n) \underset{n>1}{=} 2V_{MS}(\frac{n}{2}) + n = 2 \cdot \frac{n}{2} \log_2 \frac{n}{2} + n = n \cdot (\log_2 n - \overbrace{\log_2 2}^{1}) + n = n \log_2 n$

Laufzeit von MergeSort

MergeSort(int[] A, int $\ell = 1$, int $r = A.length$)
if $\ell < r$ **then**
$m = \lfloor (\ell + r)/2 \rfloor$ } **teile**
MergeSort(A, ℓ, m)
MergeSort($A, m+1, r$) } **herrsche**
Merge(A, ℓ, m, r) } **kombiniere**



Effizient? ✓

Sei $V_{\mathrm{MS}}(n)$ die maximale Anzahl von Vergleichen, die MergeSort zum Sortieren von n Zahlen benötigt.

Dann gilt

$$V_{\mathrm{MS}}(n) = \begin{cases} 0 & \text{falls } n = 1, \\ 2V_{\mathrm{MS}}(n/2) + n & \text{sonst.} \end{cases} = n \cdot \log_2 n$$

falls n Zweierpotenz

Beweis. Per Induktion über n.

Induktionsanfang: $V_{\mathrm{MS}}(1) = 1 \cdot \log_2 1 = 0$ ✓

Induktionsschritt: $V_{\mathrm{MS}}(n) \underset{n>1}{=} 2V_{\mathrm{MS}}(\frac{n}{2}) + n = 2 \cdot \frac{n}{2} \log_2 \frac{n}{2} + n = n \cdot (\log_2 n - \overbrace{\log_2 2}^{1}) + n = n \log_2 n$ ✓

Vergleich Laufzeiten

	Bester Fall	Schlechtester Fall
InsertionSort	$n - 1$	$\frac{n(n-1)}{2}$
SelectionSort	$\frac{n(n-1)}{2}$	$\frac{n(n-1)}{2}$
BubbleSort	$n - 1$	$\frac{n(n-1)}{2}$
BogoSort	$n - 1$	∞

Geht das besser?

Vergleich Laufzeiten

	Bester Fall	Schlechtester Fall
InsertionSort	$n-1$	$\frac{n(n-1)}{2}$
SelectionSort	$\frac{n(n-1)}{2}$	$\frac{n(n-1)}{2}$
BubbleSort	$n-1$	$\frac{n(n-1)}{2}$
BogoSort	$n-1$	∞
MergeSort		

Geht das besser?

Vergleich Laufzeiten

	Bester Fall	Schlechtester Fall
InsertionSort	$n-1$	$\frac{n(n-1)}{2}$
SelectionSort	$\frac{n(n-1)}{2}$	$\frac{n(n-1)}{2}$
BubbleSort	$n-1$	$\frac{n(n-1)}{2}$
BogoSort	$n-1$	∞
MergeSort		$n \log_2 n$

Geht das besser?

Vergleich Laufzeiten

	Bester Fall	Schlechtester Fall
InsertionSort	$n-1$	$\frac{n(n-1)}{2}$
SelectionSort	$\frac{n(n-1)}{2}$	$\frac{n(n-1)}{2}$
BubbleSort	$n-1$	$\frac{n(n-1)}{2}$
BogoSort	$n-1$	∞
MergeSort	$n \log_2 n$	$n \log_2 n$

Geht das besser?

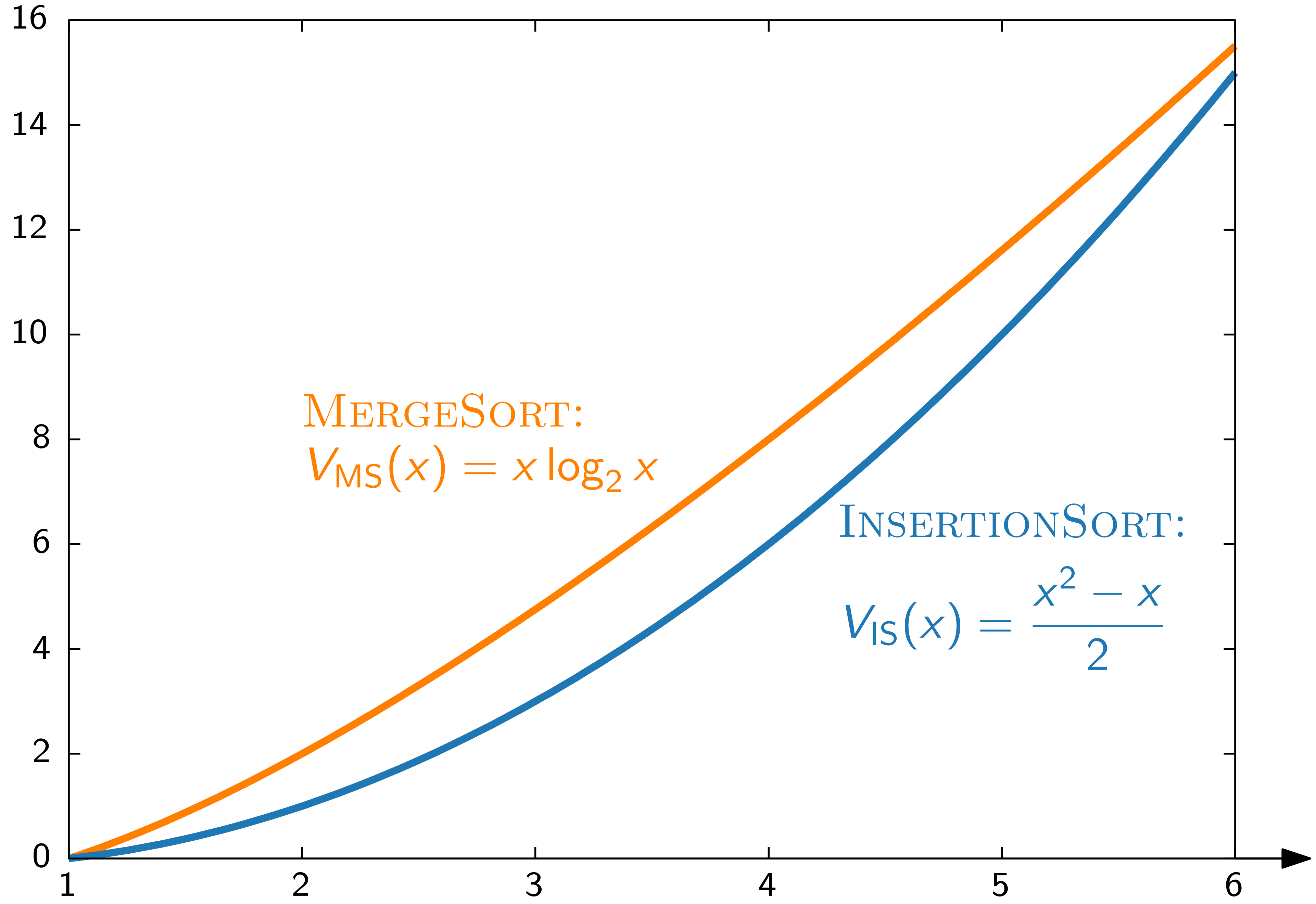
Vergleich Laufzeiten

	Bester Fall	Schlechtester Fall
InsertionSort	$n-1$	$\frac{n(n-1)}{2}$
SelectionSort	$\frac{n(n-1)}{2}$	$\frac{n(n-1)}{2}$
BubbleSort	$n-1$	$\frac{n(n-1)}{2}$
BogoSort	$n-1$	∞
MergeSort	$n \log_2 n$	$n \log_2 n$

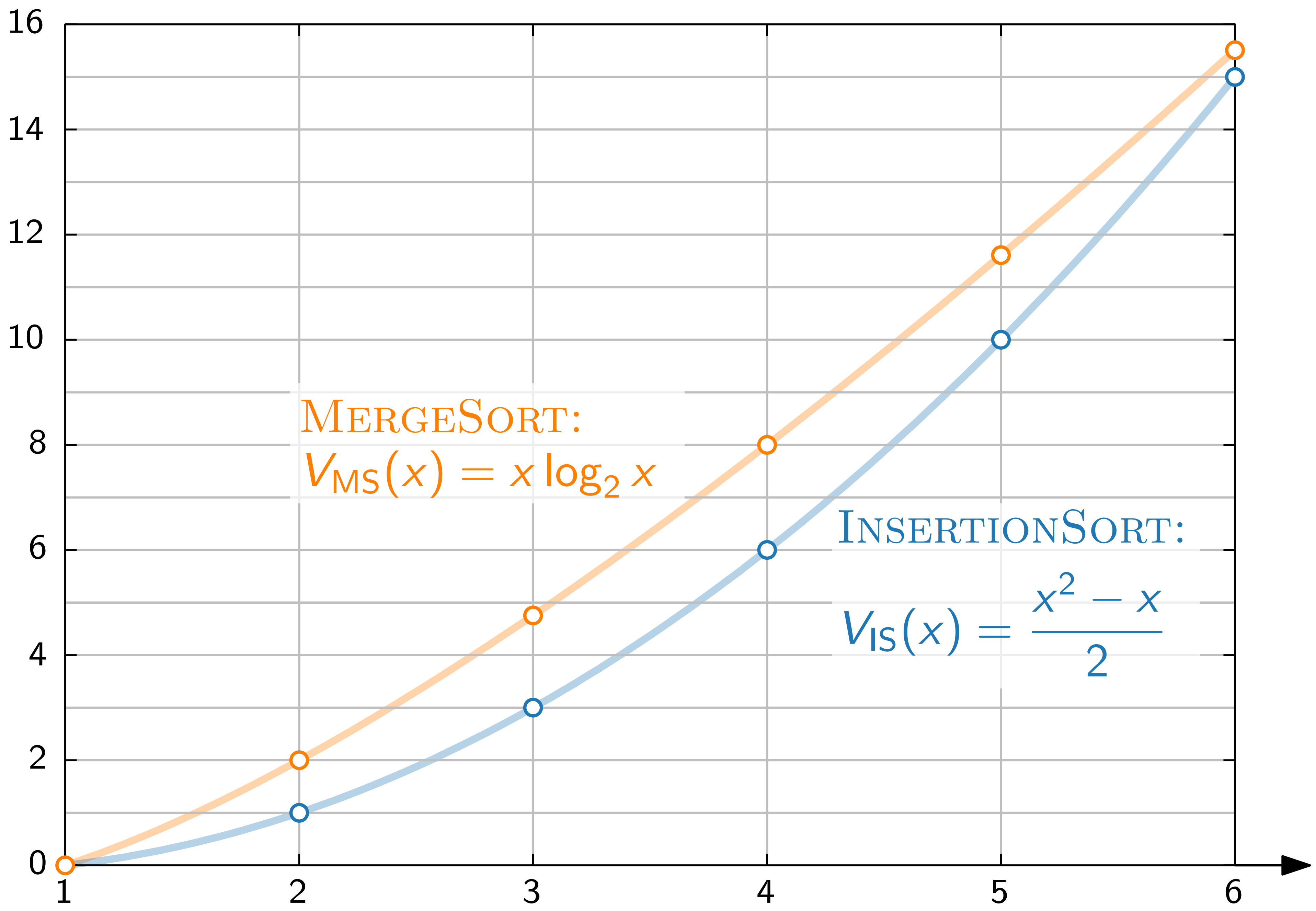
Geht das besser?

Ist das besser?

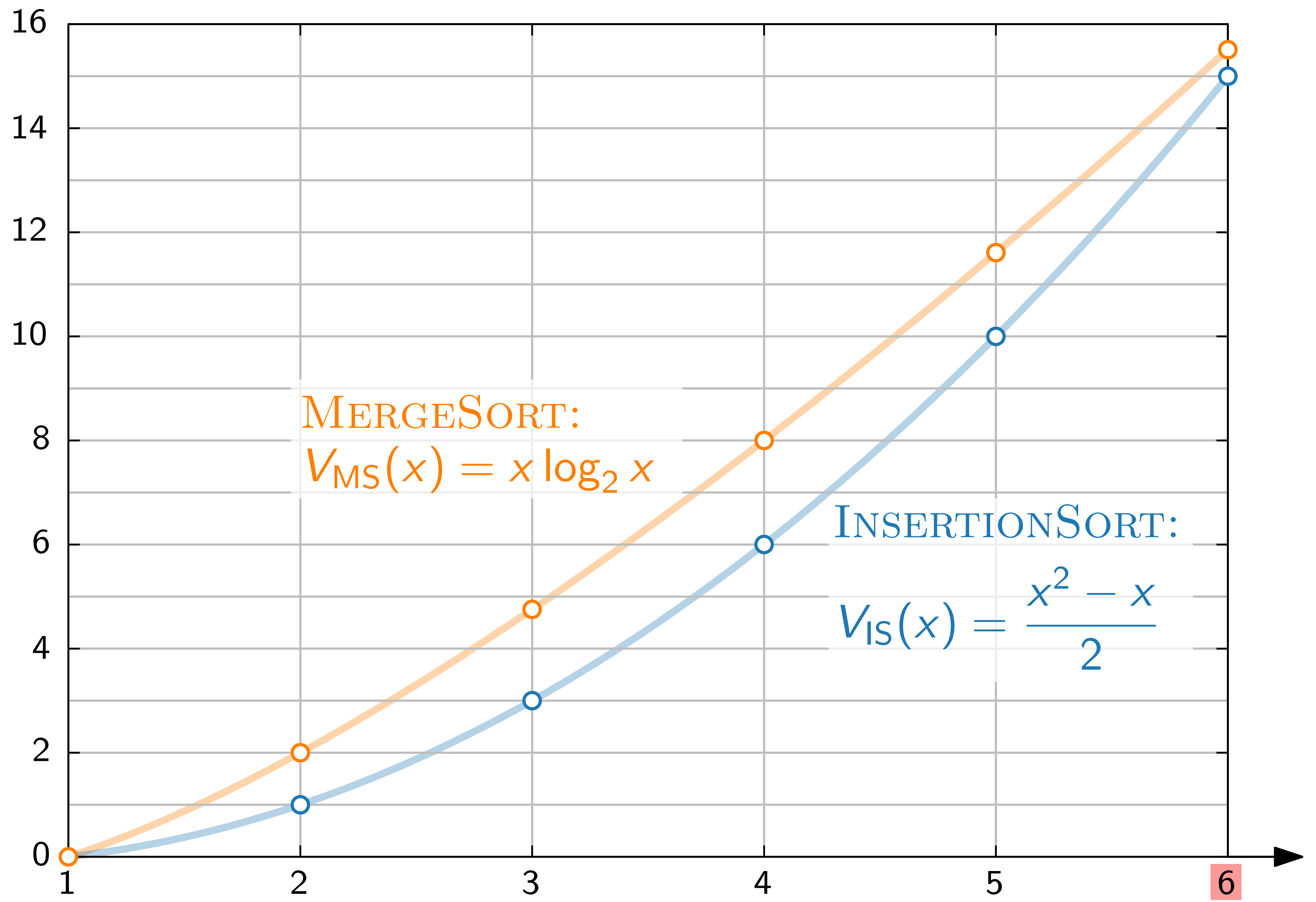
Vergleich InsertionSort vs. MergeSort



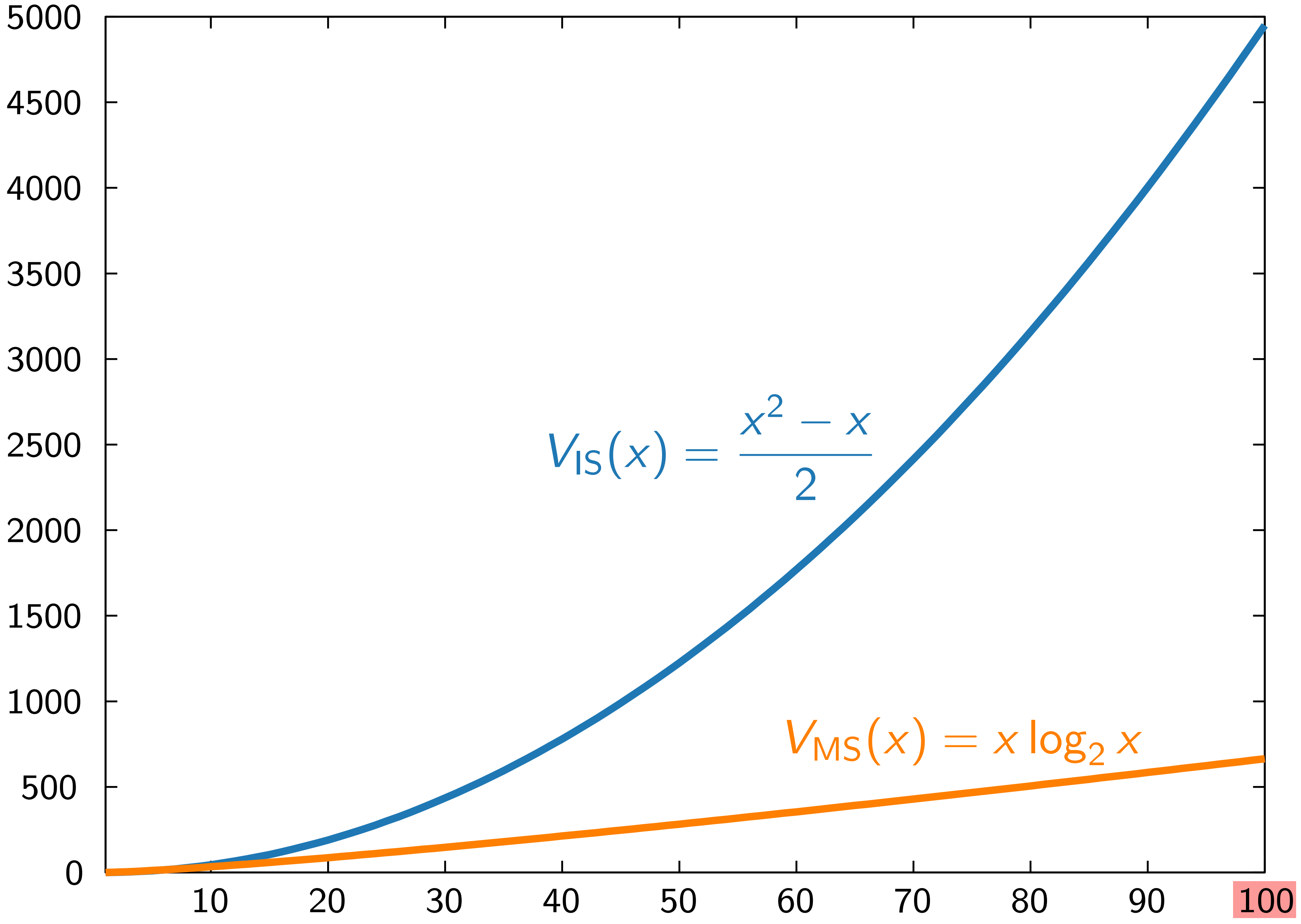
Vergleich InsertionSort vs. MergeSort



Vergleich InsertionSort vs. MergeSort



Vergleich InsertionSort vs. MergeSort



Tatsächliche Laufzeit

Was ist schneller?

Tatsächliche Laufzeit

Was ist schneller?

$\textsc{AlgA}(n)$
  $n = n + 1$
  $n = n + 1$
  $n = n + 1$
  $n = n + 1$
  $n = n + 1$
  return n

Tatsächliche Laufzeit

Was ist schneller?

$\textsc{AlgA}(n)$		$\textsc{AlgB}(n)$
$n = n + 1$	oder	$n = n + 5$
$n = n + 1$		**return** n
$n = n + 1$		
$n = n + 1$		
$n = n + 1$		
return n		

Tatsächliche Laufzeit

Was ist schneller?

$\textsc{AlgA}(n)$
  $n = n + 1$
  $n = n + 1$
  $n = n + 1$
  $n = n + 1$
  $n = n + 1$
  return n

oder

$\textsc{AlgB}(n)$
  $n = n + 5$
  return n

$a = a + 1$ oder $a = a - 1$

Tatsächliche Laufzeit

Was ist schneller?

$\textsc{AlgA}(n)$
$n = n + 1$
$n = n + 1$
$n = n + 1$
$n = n + 1$
$n = n + 1$
return n

oder

$\textsc{AlgB}(n)$
$n = n + 5$
return n

$a = a + 1$ oder $a = a - 1$

$b = b + 1$ oder $b = b + 100$

Tatsächliche Laufzeit

Was ist schneller?

$\textsc{AlgA}(n)$
  $n = n + 1$
  $n = n + 1$
  $n = n + 1$
  $n = n + 1$
  $n = n + 1$
  return n

oder

$\textsc{AlgB}(n)$
  $n = n + 5$
  return n

$a = a + 1$ oder $a = a - 1$

$b = b + 1$ oder $b = b + 100$

$c = c + 10$ oder $c = c \cdot 10$

Tatsächliche Laufzeit

Was ist schneller?

$\textsc{AlgA}(n)$ $n = n + 1$ $n = n + 1$ $n = n + 1$ $n = n + 1$ $n = n + 1$ **return** n	oder	$\textsc{AlgB}(n)$ $n = n + 5$ **return** n

$a = a + 1$	oder	$a = a - 1$
$b = b + 1$	oder	$b = b + 100$
$c = c + 10$	oder	$c = c \cdot 10$
$d = A[0]$	oder	$d = A[100]$

Tatsächliche Laufzeit

Was ist schneller?

$\textsc{AlgA}(n)$
$n = n + 1$
$n = n + 1$
$n = n + 1$
$n = n + 1$
$n = n + 1$
return n

oder

$\textsc{AlgB}(n)$
$n = n + 5$
return n

$a = a + 1$ oder $a = a - 1$

$b = b + 1$ oder $b = b + 100$

$c = c + 10$ oder $c = c \cdot 10$

$d = A[0]$ oder $d = A[100]$

Hängt von vielen Faktoren ab!

Tatsächliche Laufzeit

Was ist schneller?

$\textsc{AlgA}(n)$ $n = n + 1$ $n = n + 1$ $n = n + 1$ $n = n + 1$ $n = n + 1$ **return** n	oder	$\textsc{AlgB}(n)$ $n = n + 5$ **return** n		$a = a + 1$ $b = b + 1$ $c = c + 10$ $d = A[0]$	oder oder oder oder	$a = a - 1$ $b = b + 100$ $c = c \cdot 10$ $d = A[100]$

Hängt von vielen Faktoren ab!

- Realisierung der Elementaroperationen

Tatsächliche Laufzeit

Was ist schneller?

$\textsc{AlgA}(n)$
$n = n + 1$
$n = n + 1$
$n = n + 1$
$n = n + 1$
$n = n + 1$
return n

oder

$\textsc{AlgB}(n)$
$n = n + 5$
return n

$a = a + 1$ oder $a = a - 1$

$b = b + 1$ oder $b = b + 100$

$c = c + 10$ oder $c = c \cdot 10$

$d = A[0]$ oder $d = A[100]$

Hängt von vielen Faktoren ab!

- Realisierung der Elementaroperationen
- Größe der Variablen

Tatsächliche Laufzeit

Was ist schneller?

$\textsc{AlgA}(n)$
  $n = n + 1$
  $n = n + 1$
  $n = n + 1$
  $n = n + 1$
  $n = n + 1$
  return n

oder

$\textsc{AlgB}(n)$
  $n = n + 5$
  return n

$a = a + 1$ oder $a = a - 1$

$b = b + 1$ oder $b = b + 100$

$c = c + 10$ oder $c = c \cdot 10$

$d = A[0]$ oder $d = A[100]$

Hängt von vielen Faktoren ab!

- Realisierung der Elementaroperationen
- Größe der Variablen
- Position im Speicher

Tatsächliche Laufzeit

Was ist schneller?

$\textsc{AlgA}(n)$ $n = n + 1$ $n = n + 1$ $n = n + 1$ $n = n + 1$ $n = n + 1$ **return** n	oder	$\textsc{AlgB}(n)$ $n = n + 5$ **return** n

$a = a + 1$	oder	$a = a - 1$
$b = b + 1$	oder	$b = b + 100$
$c = c + 10$	oder	$c = c \cdot 10$
$d = A[0]$	oder	$d = A[100]$

Hängt von vielen Faktoren ab!

- Realisierung der Elementaroperationen
- Größe der Variablen
- Position im Speicher
- Position des Lesekopfs

Tatsächliche Laufzeit

Was ist schneller?

$\textsc{AlgA}(n)$ $n = n + 1$ $n = n + 1$ $n = n + 1$ $n = n + 1$ $n = n + 1$ **return** n	oder	$\textsc{AlgB}(n)$ $n = n + 5$ **return** n

$a = a + 1$	oder	$a = a - 1$
$b = b + 1$	oder	$b = b + 100$
$c = c + 10$	oder	$c = c \cdot 10$
$d = A[0]$	oder	$d = A[100]$

Hängt von vielen Faktoren ab!

- Realisierung der Elementaroperationen
- Größe der Variablen
- Position im Speicher
- Position des Lesekopfs

Wollen ein Maschinenmodell, das einem Rechner ähnelt, aber möglichst einfach ist!

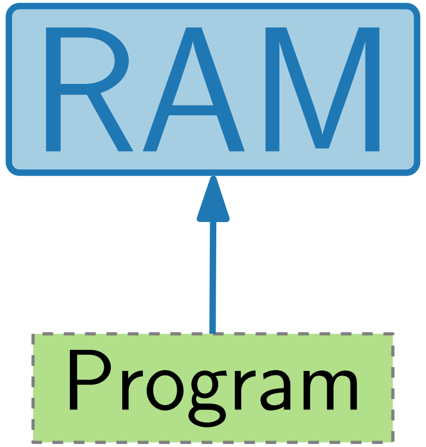
Das RAM-Modell

Das RAM-Modell

Die **Random Access Machine (RAM)** besteht aus:

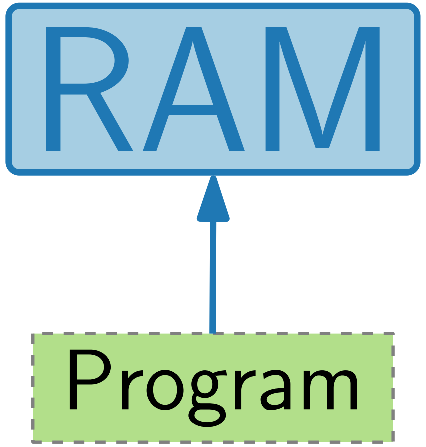
Das RAM-Modell

Die **Random Access Machine (RAM)** besteht aus:



Das RAM-Modell

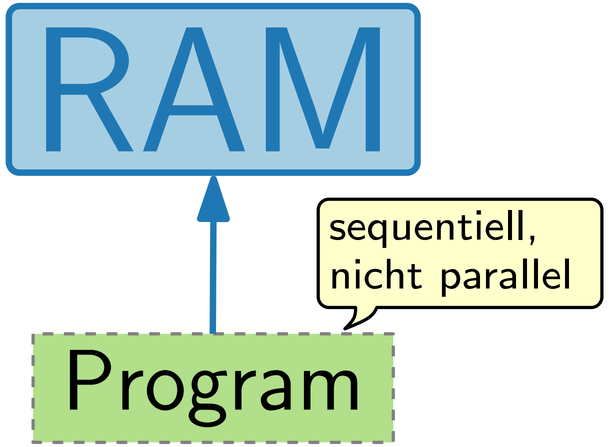
Die **Random Access Machine (RAM)** besteht aus:




```
INSERTIONSORT(int[] A)
  for j = 2 to A.length do
      key = A[j]
      i = j − 1
      while i > 0 and A[i] > key do
          A[i + 1] = A[i]
          i = i − 1
      A[i + 1] = key
```

Das RAM-Modell

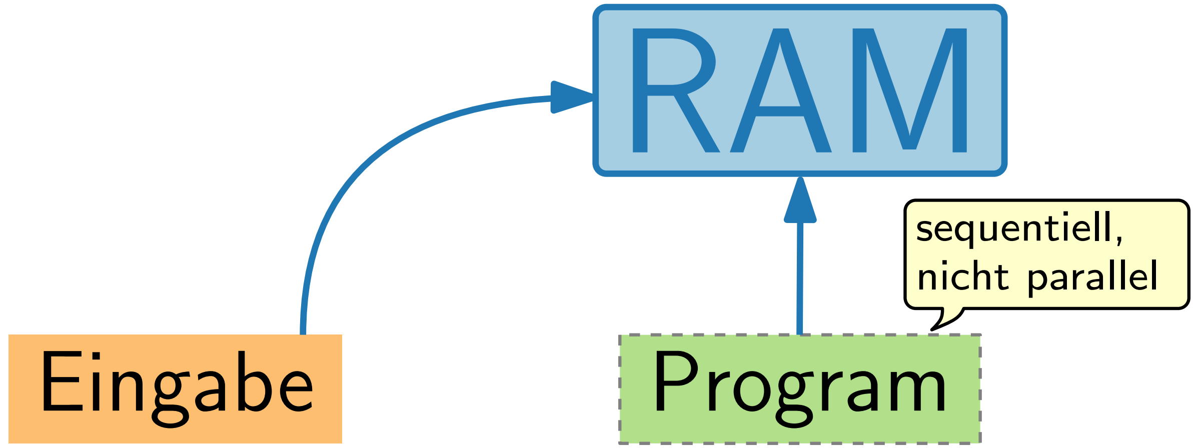
Die **Random Access Machine (RAM)** besteht aus:



InsertionSort(int[] A)
  for $j = 2$ **to** $A.length$ **do**
    $key = A[j]$
    $i = j - 1$
    while $i > 0$ **and** $A[i] > key$ **do**
      $A[i + 1] = A[i]$
      $i = i - 1$
    $A[i + 1] = key$

Das RAM-Modell

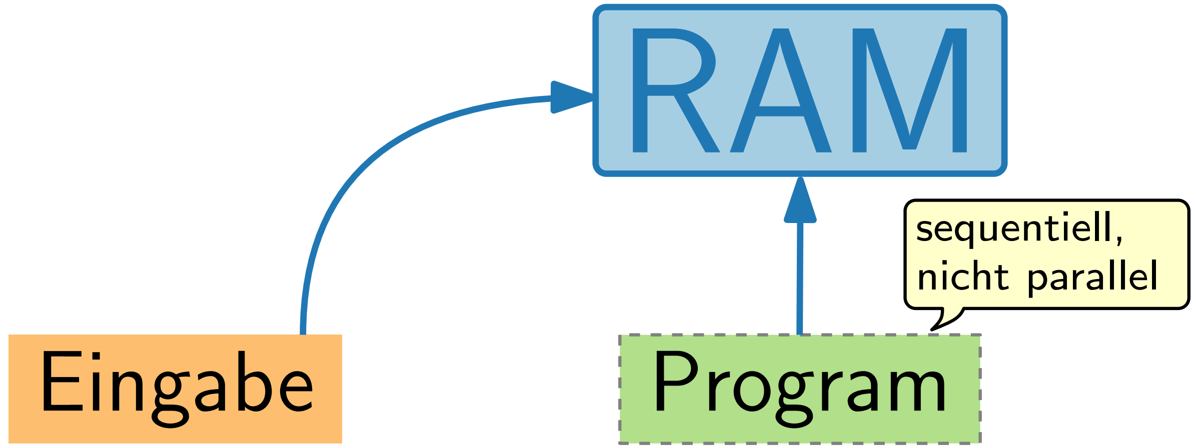
Die **Random Access Machine (RAM)** besteht aus:



INSERTIONSORT(int[] A)
  for $j = 2$ **to** $A.length$ **do**
    $key = A[j]$
    $i = j - 1$
    while $i > 0$ **and** $A[i] > key$ **do**
      $A[i+1] = A[i]$
      $i = i - 1$
    $A[i+1] = key$

Das RAM-Modell

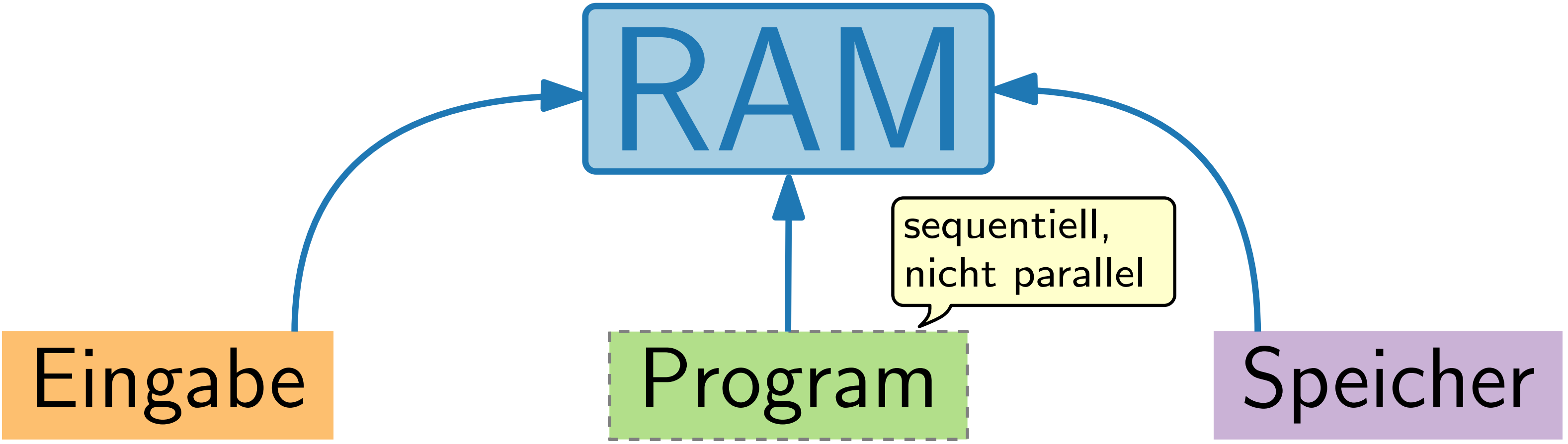
Die **Random Access Machine (RAM)** besteht aus:




```
InsertionSort(int[] A)
  for j = 2 to A.length do
      key = A[j]
      i = j − 1
      while i > 0 and A[i] > key do
          A[i + 1] = A[i]
          i = i − 1
      A[i + 1] = key
```

Das RAM-Modell

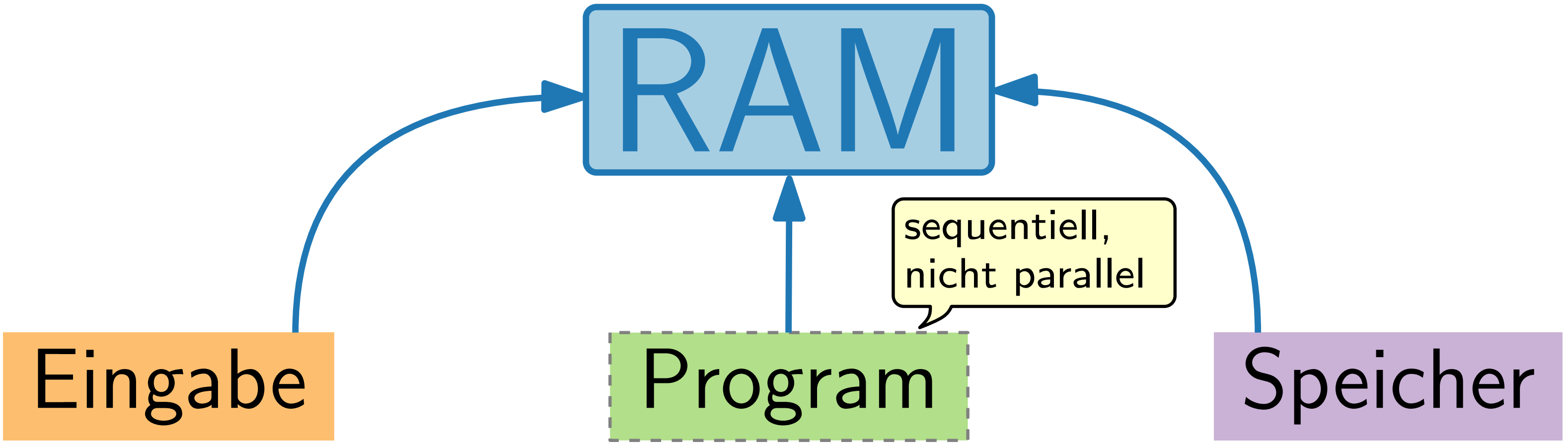
Die **Random Access Machine (RAM)** besteht aus:



INSERTIONSORT(int[] A)
for $j = 2$ **to** $A.length$ **do**
    $key = A[j]$
    $i = j - 1$
    while $i > 0$ **and** $A[i] > key$ **do**
        $A[i+1] = A[i]$
        $i = i - 1$
    $A[i+1] = key$

Das RAM-Modell

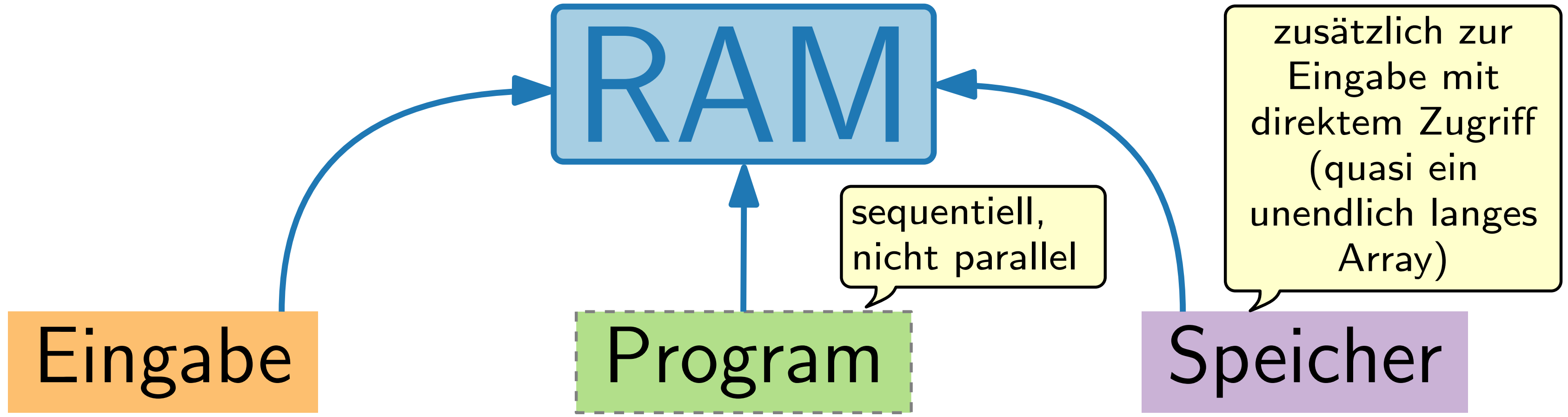
Die **Random Access Machine (RAM)** besteht aus:




```
InsertionSort(int[] A)
  for j = 2 to A.length do
      key = A[j]
      i = j − 1
      while i > 0 and A[i] > key do
          A[i + 1] = A[i]
          i = i − 1
      A[i + 1] = key
```

Das RAM-Modell

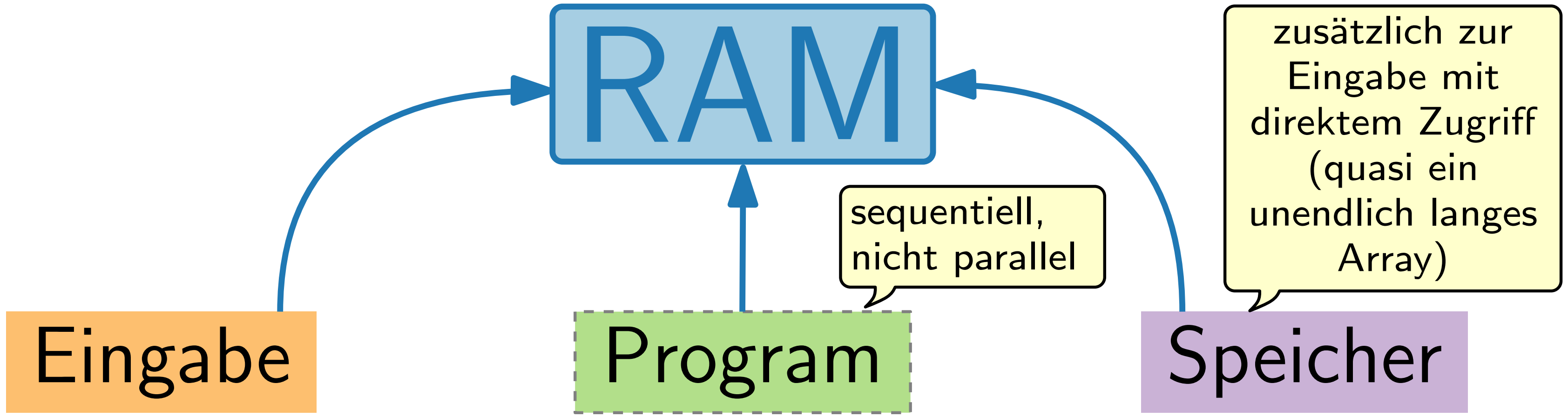
Die **Random Access Machine (RAM)** besteht aus:



InsertionSort(int[] A)
  for $j = 2$ **to** $A.length$ **do**
    $key = A[j]$
    $i = j - 1$
    while $i > 0$ **and** $A[i] > key$ **do**
      $A[i+1] = A[i]$
      $i = i - 1$
    $A[i+1] = key$

Das RAM-Modell

Die **Random Access Machine (RAM)** besteht aus:

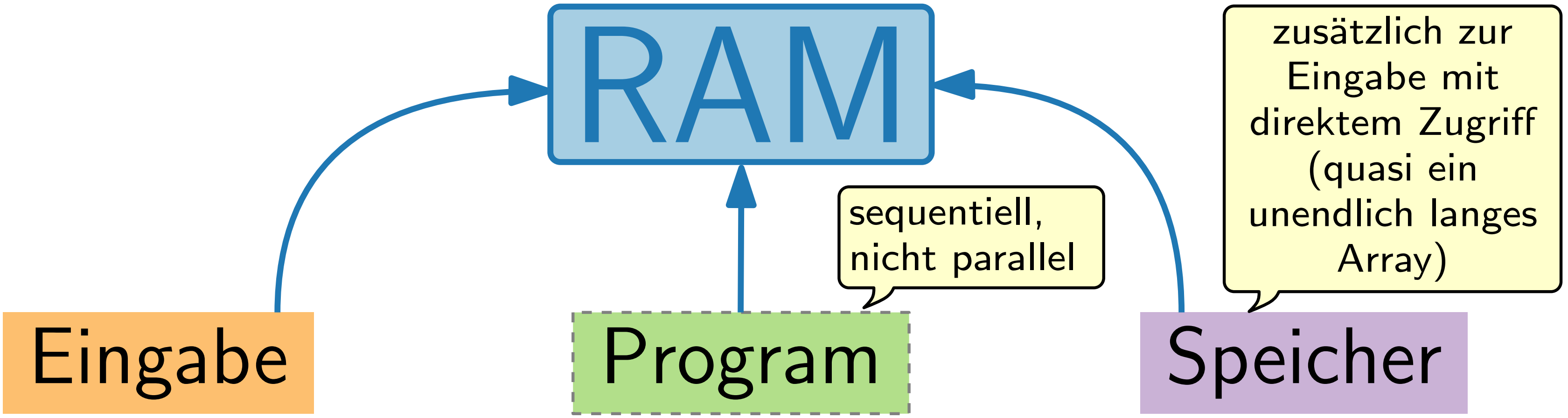


- Lesen

```
InsertionSort(int[] A)
  for j = 2 to A.length do
    key = A[j]
    i = j − 1
    while i > 0 and A[i] > key do
      A[i + 1] = A[i]
      i = i − 1
    A[i + 1] = key
```

Das RAM-Modell

Die **Random Access Machine (RAM)** besteht aus:

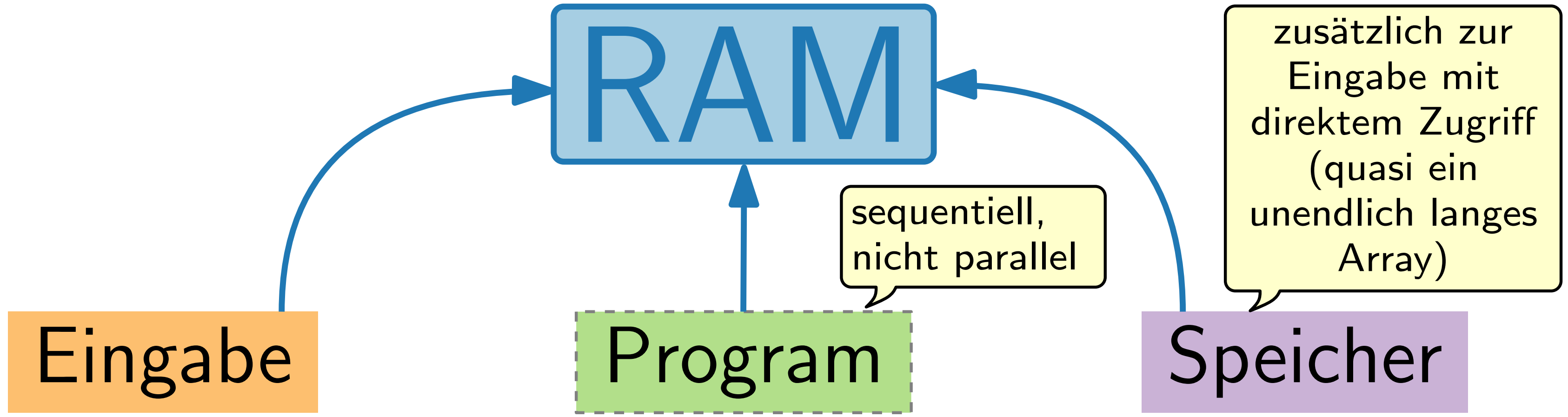


- Lesen
- Schreiben

```
InsertionSort(int[] A)
  for j = 2 to A.length do
    key = A[j]
    i = j − 1
    while i > 0 and A[i] > key do
      A[i + 1] = A[i]
      i = i − 1
    A[i + 1] = key
```

Das RAM-Modell

Die **Random Access Machine (RAM)** besteht aus:



Eingabe:

- Lesen
- Schreiben

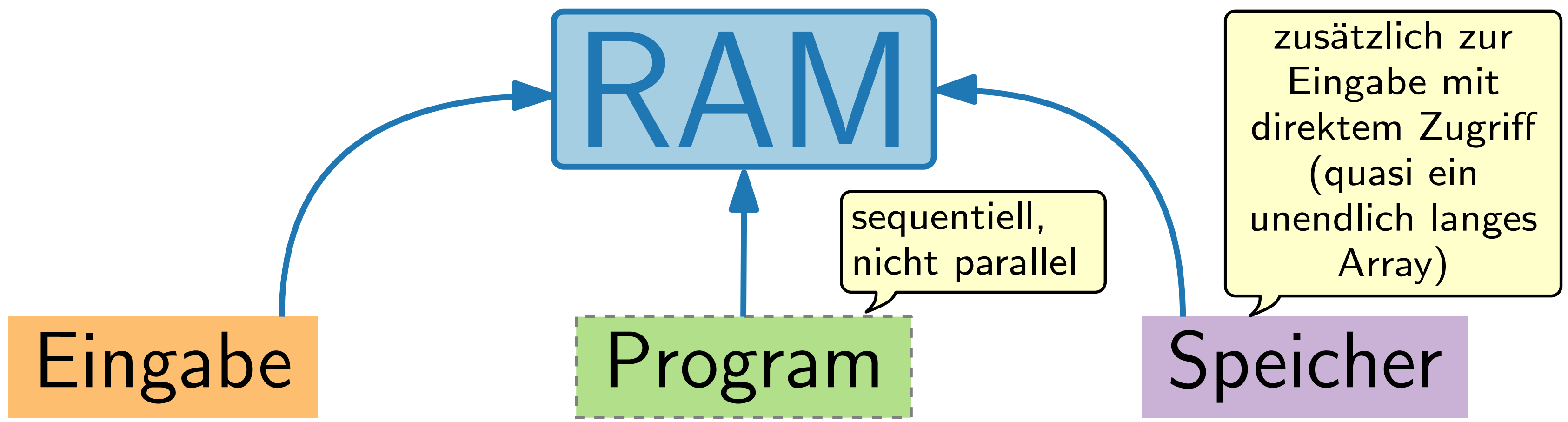
Speicher:

- Lesen
- Schreiben

```
InsertionSort(int[] A)
  for j = 2 to A.length do
    key = A[j]
    i = j - 1
    while i > 0 and A[i] > key do
      A[i + 1] = A[i]
      i = i - 1
    A[i + 1] = key
```

Das RAM-Modell

Die **Random Access Machine (RAM)** besteht aus:



Eingabe:
- Lesen
- Schreiben

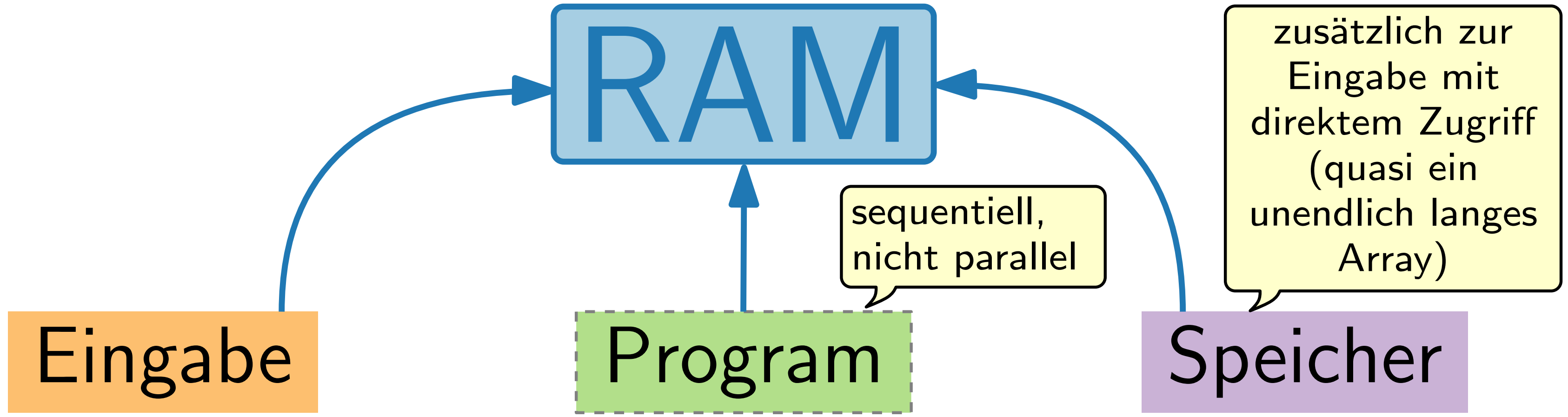
Program:
- Elementaroperationen:

Speicher:
- Lesen
- Schreiben

```
InsertionSort(int[] A)
  for j = 2 to A.length do
    key = A[j]
    i = j − 1
    while i > 0 and A[i] > key do
      A[i + 1] = A[i]
      i = i − 1
    A[i + 1] = key
```

Das RAM-Modell

Die **Random Access Machine (RAM)** besteht aus:

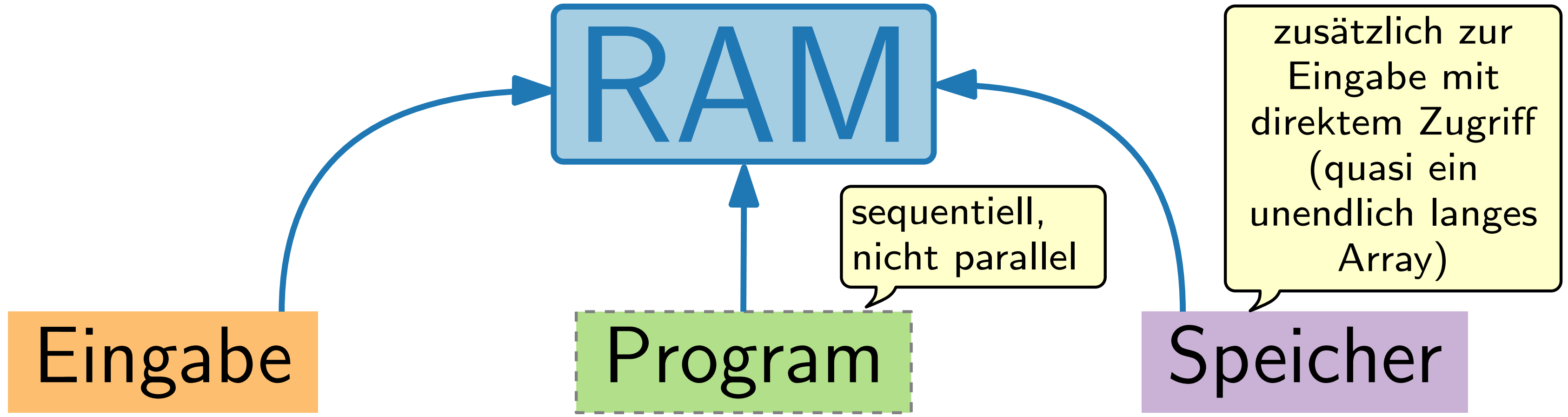


- **Eingabe**
  - Lesen
  - Schreiben
- **Program**
  - Elementaroperationen:
    - $+, -, \cdot, /, \%$
- **Speicher**
  - Lesen
  - Schreiben

```
InsertionSort(int[] A)
  for j = 2 to A.length do
    key = A[j]
    i = j - 1
    while i > 0 and A[i] > key do
      A[i + 1] = A[i]
      i = i - 1
    A[i + 1] = key
```

Das RAM-Modell

Die **Random Access Machine (RAM)** besteht aus:



- Lesen
- Schreiben

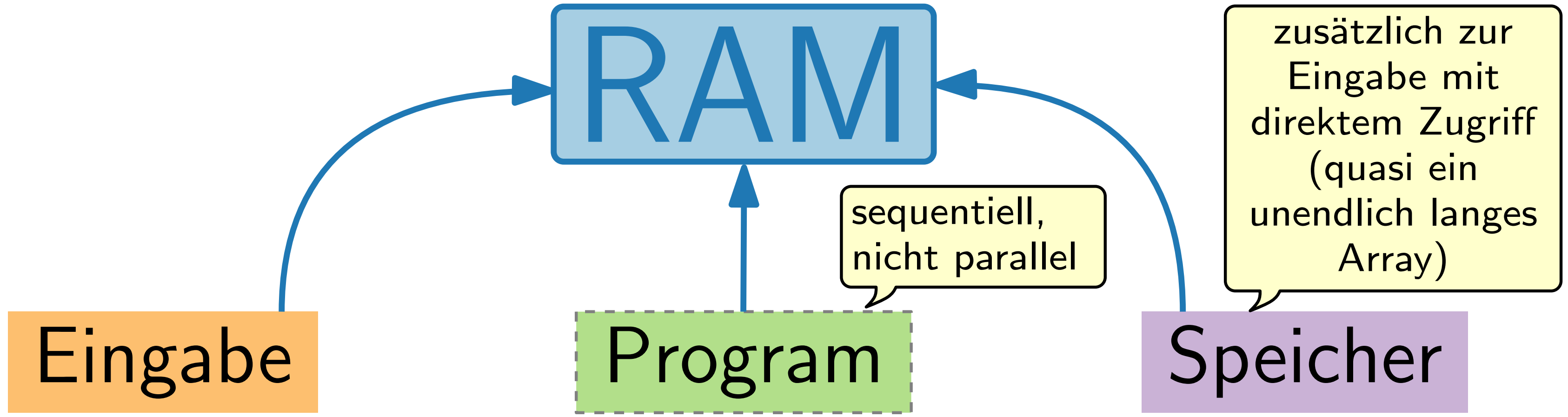
- Elementaroperationen:
  - $+, -, \cdot, /, \%$
  - $\min, \max, \lceil x \rceil, \lfloor x \rfloor$

- Lesen
- Schreiben

```
InsertionSort(int[] A)
  for j = 2 to A.length do
    key = A[j]
    i = j − 1
    while i > 0 and A[i] > key do
      A[i + 1] = A[i]
      i = i − 1
    A[i + 1] = key
```

Das RAM-Modell

Die **Random Access Machine (RAM)** besteht aus:



Eingabe
- Lesen
- Schreiben

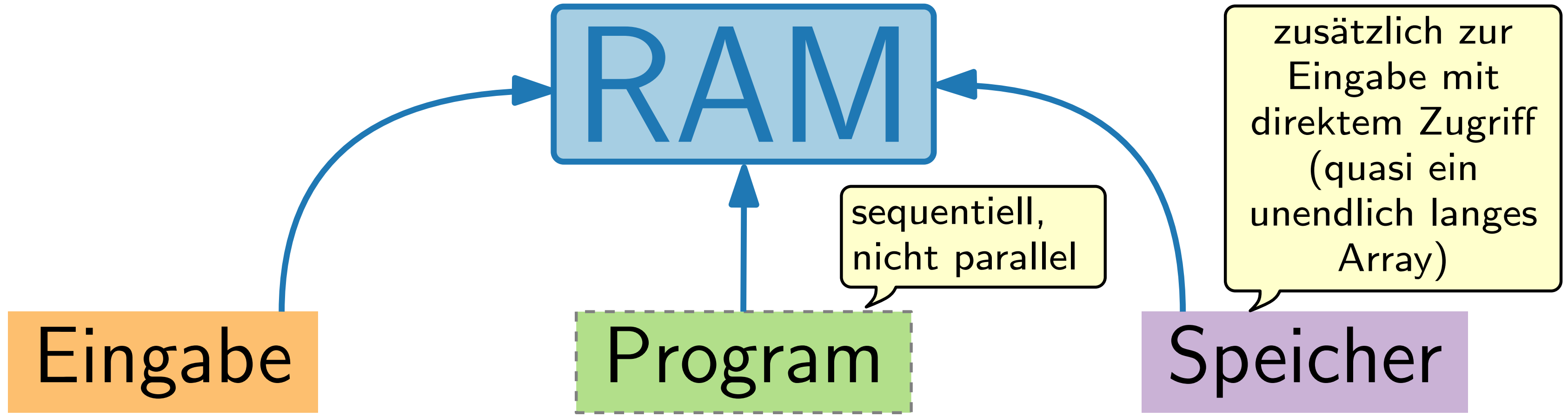
Program
- Elementaroperationen:
  - $+, -, \cdot, /, \%$
  - min, max, $\lceil x \rceil$, $\lfloor x \rfloor$
- Vergleiche:

Speicher
- Lesen
- Schreiben

```
InsertionSort(int[] A)
  for j = 2 to A.length do
    key = A[j]
    i = j − 1
    while i > 0 and A[i] > key do
      A[i + 1] = A[i]
      i = i − 1
    A[i + 1] = key
```

Das RAM-Modell

Die **Random Access Machine (RAM)** besteht aus:



Eingabe

- Lesen
- Schreiben

Program

- Elementaroperationen:
  - $+, -, \cdot, /, \%$
  - $\min, \max, \lceil x \rceil, \lfloor x \rfloor$
- Vergleiche:
  - $<, \leq, =, \neq, \geq, >$

Speicher

- Lesen
- Schreiben

```
InsertionSort(int[] A)
  for j = 2 to A.length do
    key = A[j]
    i = j − 1
    while i > 0 and A[i] > key do
      A[i + 1] = A[i]
      i = i − 1
    A[i + 1] = key
```

Das RAM-Modell

Die **Random Access Machine (RAM)** besteht aus:

RAM

sequentiell, nicht parallel

zusätzlich zur Eingabe mit direktem Zugriff (quasi ein unendlich langes Array)

Eingabe
- Lesen
- Schreiben

Program
- Elementaroperationen:
  - $+, -, \cdot, /, \%$
  - $\min, \max, \lceil x \rceil, \lfloor x \rfloor$
- Vergleiche:
  - $<, \leq, =, \neq, \geq, >$

Speicher
- Lesen
- Schreiben

Je 1 Zeiteinheit

```
InsertionSort(int[] A)
  for j = 2 to A.length do
    key = A[j]
    i = j − 1
    while i > 0 and A[i] > key do
      A[i + 1] = A[i]
      i = i − 1
    A[i + 1] = key
```

Laufzeit von InsertionSort

InsertionSort(int[] A)
  for $j = 2$ **to** $A.\mathit{length}$ **do**
    $\mathit{key} = A[j]$
    $i = j - 1$
    while $i > 0$ **and** $A[i] > \mathit{key}$ **do**
      $A[i + 1] = A[i]$
      $i = i - 1$
    $A[i + 1] = \mathit{key}$

Laufzeit von InsertionSort

```
InsertionSort(int[] A)
  for j = 2 to A.length do
      key = A[j]
      i = j − 1
      while i > 0 and A[i] > key do
          A[i + 1] = A[i]
          i = i − 1
      A[i + 1] = key
```

Wie lang braucht dieser Algorithmus?
Wir zählen **alle Rechenschritte**.

Laufzeit von InsertionSort

```
InsertionSort(int[] A)
  for j = 2 to A.length do
      key = A[j]
      i = j − 1
      while i > 0 and A[i] > key do
          A[i + 1] = A[i]
          i = i − 1
      A[i + 1] = key
```

Wie lang braucht dieser Algorithmus?
Wir zählen **alle Rechenschritte**.

Laufzeit von InsertionSort

```
InsertionSort(int[] A)
  for j = 2 to A.length do          // 1 Zuweisung + 1 Addition + 1 Vergleich
      key = A[j]
      i = j − 1
      while i > 0 and A[i] > key do
          A[i + 1] = A[i]
          i = i − 1
      A[i + 1] = key
```

Wie lang braucht dieser Algorithmus?
Wir zählen **alle Rechenschritte**.

Laufzeit von InsertionSort

```
InsertionSort(int[] A)
  for j = 2 to A.length do
      key = A[j]                    // 1 Zuweisung + 1 Addition + 1 Vergleich
      i = j − 1
      while i > 0 and A[i] > key do
          A[i + 1] = A[i]
          i = i − 1
      A[i + 1] = key
```

Wie lang braucht dieser Algorithmus?
Wir zählen **alle Rechenschritte**.

Laufzeit von InsertionSort

```
InsertionSort(int[] A)
  for j = 2 to A.length do                 // 1 Zuweisung + 1 Addition + 1 Vergleich
      key = A[j]                           // 1 Zuweisung + 1 Feldzugriff
      i = j − 1
      while i > 0 and A[i] > key do
          A[i + 1] = A[i]
          i = i − 1
      A[i + 1] = key
```

Wie lang braucht dieser Algorithmus?
Wir zählen **alle Rechenschritte**.

Laufzeit von InsertionSort

```
InsertionSort(int[] A)
  for j = 2 to A.length do          // 1 Zuweisung + 1 Addition + 1 Vergleich
      key = A[j]                    // 1 Zuweisung + 1 Feldzugriff
      i = j − 1
      while i > 0 and A[i] > key do
          A[i + 1] = A[i]
          i = i − 1
      A[i + 1] = key
```

Wie lang braucht dieser Algorithmus?
Wir zählen **alle Rechenschritte**.

Laufzeit von InsertionSort

```
InsertionSort(int[] A)
  for j = 2 to A.length do            // 1 Zuweisung + 1 Addition + 1 Vergleich
      key = A[j]                      // 1 Zuweisung + 1 Feldzugriff
      i = j − 1                       // 1 Zuweisung + 1 Subtraktion
      while i > 0 and A[i] > key do
          A[i + 1] = A[i]
          i = i − 1
      A[i + 1] = key
```

Wie lang braucht dieser Algorithmus?
Wir zählen **alle Rechenschritte**.

Laufzeit von InsertionSort

```
InsertionSort(int[] A)
  for j = 2 to A.length do                // 1 Zuweisung + 1 Addition + 1 Vergleich
      key = A[j]                          // 1 Zuweisung + 1 Feldzugriff
      i = j − 1                           // 1 Zuweisung + 1 Subtraktion
      while i > 0 and A[i] > key do
          A[i + 1] = A[i]
          i = i − 1
      A[i + 1] = key
```

Wie lang braucht dieser Algorithmus?
Wir zählen **alle Rechenschritte**.

Laufzeit von InsertionSort

```
InsertionSort(int[] A)
  for j = 2 to A.length do                  // 1 Zuweisung + 1 Addition + 1 Vergleich
      key = A[j]                            // 1 Zuweisung + 1 Feldzugriff
      i = j − 1                             // 1 Zuweisung + 1 Subtraktion
      while i > 0 and A[i] > key do         // 2 Vergleiche + 1 Feldzugriff
          A[i + 1] = A[i]
          i = i − 1
      A[i + 1] = key
```

Wie lang braucht dieser Algorithmus?
Wir zählen **alle Rechenschritte**.

Laufzeit von InsertionSort

```
InsertionSort(int[] A)
  for j = 2 to A.length do            // 1 Zuweisung + 1 Addition + 1 Vergleich
      key = A[j]                      // 1 Zuweisung + 1 Feldzugriff
      i = j − 1                       // 1 Zuweisung + 1 Subtraktion
      while i > 0 and A[i] > key do   // 2 Vergleiche + 1 Feldzugriff
          A[i + 1] = A[i]
          i = i − 1
      A[i + 1] = key
```

Wie lang braucht dieser Algorithmus?
Wir zählen **alle Rechenschritte**.

Laufzeit von InsertionSort

```
InsertionSort(int[] A)
  for j = 2 to A.length do                 // 1 Zuweisung + 1 Addition + 1 Vergleich
      key = A[j]                           // 1 Zuweisung + 1 Feldzugriff
      i = j − 1                            // 1 Zuweisung + 1 Subtraktion
      while i > 0 and A[i] > key do        // 2 Vergleiche + 1 Feldzugriff
          A[i + 1] = A[i]                  // 1 Zuweisung + 1 Addition + 2 Feldzugriffe
          i = i − 1
      A[i + 1] = key
```

Wie lang braucht dieser Algorithmus?
Wir zählen **alle Rechenschritte**.

Laufzeit von InsertionSort

```
InsertionSort(int[] A)
  for j = 2 to A.length do            // 1 Zuweisung + 1 Addition + 1 Vergleich
      key = A[j]                      // 1 Zuweisung + 1 Feldzugriff
      i = j − 1                       // 1 Zuweisung + 1 Subtraktion
      while i > 0 and A[i] > key do   // 2 Vergleiche + 1 Feldzugriff
          A[i + 1] = A[i]             // 1 Zuweisung + 1 Addition + 2 Feldzugriffe
          i = i − 1
      A[i + 1] = key
```

Wie lang braucht dieser Algorithmus?
Wir zählen **alle Rechenschritte**.

Laufzeit von InsertionSort

```
InsertionSort(int[] A)
  for j = 2 to A.length do              // 1 Zuweisung + 1 Addition + 1 Vergleich
      key = A[j]                        // 1 Zuweisung + 1 Feldzugriff
      i = j − 1                         // 1 Zuweisung + 1 Subtraktion
      while i > 0 and A[i] > key do     // 2 Vergleiche + 1 Feldzugriff
          A[i + 1] = A[i]               // 1 Zuweisung + 1 Addition + 2 Feldzugriffe
          i = i − 1                     // 1 Zuweisung + 1 Subtraktion
      A[i + 1] = key
```

Wie lang braucht dieser Algorithmus?
Wir zählen **alle Rechenschritte**.

Laufzeit von InsertionSort

```
InsertionSort(int[] A)
  for j = 2 to A.length do                // 1 Zuweisung + 1 Addition + 1 Vergleich
      key = A[j]                          // 1 Zuweisung + 1 Feldzugriff
      i = j − 1                           // 1 Zuweisung + 1 Subtraktion
      while i > 0 and A[i] > key do       // 2 Vergleiche + 1 Feldzugriff
          A[i + 1] = A[i]                 // 1 Zuweisung + 1 Addition + 2 Feldzugriffe
          i = i − 1                       // 1 Zuweisung + 1 Subtraktion
      A[i + 1] = key
```

Wie lang braucht dieser Algorithmus?

Wir zählen **alle Rechenschritte**.

Laufzeit von InsertionSort

```
InsertionSort(int[] A)
  for j = 2 to A.length do              // 1 Zuweisung + 1 Addition + 1 Vergleich
      key = A[j]                        // 1 Zuweisung + 1 Feldzugriff
      i = j − 1                         // 1 Zuweisung + 1 Subtraktion
      while i > 0 and A[i] > key do     // 2 Vergleiche + 1 Feldzugriff
          A[i + 1] = A[i]               // 1 Zuweisung + 1 Addition + 2 Feldzugriffe
          i = i − 1                     // 1 Zuweisung + 1 Subtraktion
      A[i + 1] = key                    // 1 Zuweisung + 1 Addition + 1 Feldzugriff
```

Wie lang braucht dieser Algorithmus?
Wir zählen **alle Rechenschritte**.

Laufzeit von InsertionSort

```
InsertionSort(int[] A)
  for j = 2 to A.length do              // 1 Zuweisung + 1 Addition + 1 Vergleich
      key = A[j]                        // 1 Zuweisung + 1 Feldzugriff
      i = j − 1                         // 1 Zuweisung + 1 Subtraktion
      while i > 0 and A[i] > key do     // 2 Vergleiche + 1 Feldzugriff
          A[i + 1] = A[i]               // 1 Zuweisung + 1 Addition + 2 Feldzugriffe
          i = i − 1                     // 1 Zuweisung + 1 Subtraktion
      A[i + 1] = key                    // 1 Zuweisung + 1 Addition + 1 Feldzugriff
```

Wie lang braucht dieser Algorithmus?
Wir zählen **alle Rechenschritte**.

Schlechtester Fall.
Array ist absteigend sortiert.

9	8	7	6	5	4	3	2	1

$\Rightarrow A[i] > key$ **immer** erfüllt

Laufzeit von InsertionSort

```
InsertionSort(int[] A)
  for j = 2 to A.length do              // 1 Zuweisung + 1 Addition + 1 Vergleich
    key = A[j]                          // 1 Zuweisung + 1 Feldzugriff
    i = j − 1                           // 1 Zuweisung + 1 Subtraktion
    while i > 0 and A[i] > key do       // 2 Vergleiche + 1 Feldzugriff
      A[i + 1] = A[i]                   // 1 Zuweisung + 1 Addition + 2 Feldzugriffe
      i = i − 1                         // 1 Zuweisung + 1 Subtraktion
    A[i + 1] = key                      // 1 Zuweisung + 1 Addition + 1 Feldzugriff
```

Wie lang braucht dieser Algorithmus?
Wir zählen **alle Rechenschritte**.

Schlechtester Fall.
Array ist absteigend sortiert.

9	8	7	6	5	4	3	2	1

$\Rightarrow A[i] > key$ **immer** erfüllt

$\Rightarrow$ für jedes j gibt es Rechenschr.

Laufzeit von InsertionSort

```
InsertionSort(int[] A)
  for j = 2 to A.length do          // 1 Zuweisung + 1 Addition + 1 Vergleich
      key = A[j]                    // 1 Zuweisung + 1 Feldzugriff
      i = j − 1                     // 1 Zuweisung + 1 Subtraktion
      while i > 0 and A[i] > key do // 2 Vergleiche + 1 Feldzugriff
          A[i + 1] = A[i]           // 1 Zuweisung + 1 Addition + 2 Feldzugriffe
          i = i − 1                 // 1 Zuweisung + 1 Subtraktion
      A[i + 1] = key                // 1 Zuweisung + 1 Addition + 1 Feldzugriff
```

$\boxed{13}$

Wie lang braucht dieser Algorithmus?
Wir zählen **alle Rechenschritte**.

Schlechtester Fall.
Array ist absteigend sortiert.

9	8	7	6	5	4	3	2	1

$\Rightarrow$ $A[i] > key$ **immer** erfüllt

$\Rightarrow$ für jedes j gibt es Rechenschr.

Laufzeit von InsertionSort

InsertionSort(int[] A)
for $j = 2$ **to** $A.length$ **do** // 1 Zuweisung + 1 Addition + 1 Vergleich
  $key = A[j]$ // 1 Zuweisung + 1 Feldzugriff
  $i = j - 1$ // 1 Zuweisung + 1 Subtraktion
  while $i > 0$ **and** $A[i] > key$ **do** // 2 Vergleiche + 1 Feldzugriff
    $A[i+1] = A[i]$ // 1 Zuweisung + 1 Addition + 2 Feldzugriffe
    $i = i - 1$ // 1 Zuweisung + 1 Subtraktion
  $A[i+1] = key$ // 1 Zuweisung + 1 Addition + 1 Feldzugriff

$\boxed{13}$

Wie lang braucht dieser Algorithmus?
Wir zählen **alle Rechenschritte**.

Schlechtester Fall.
Array ist absteigend sortiert.

9	8	7	6	5	4	3	2	1

$\Rightarrow$ $A[i] > key$ **immer** erfüllt
$\Rightarrow$ für jedes j gibt es 13 + Rechenschr.

Laufzeit von InsertionSort

```
InsertionSort(int[] A)
  for j = 2 to A.length do                    // 1 Zuweisung + 1 Addition + 1 Vergleich
      key = A[j]                              // 1 Zuweisung + 1 Feldzugriff
      i = j − 1                               // 1 Zuweisung + 1 Subtraktion
      while i > 0 and A[i] > key do           // 2 Vergleiche + 1 Feldzugriff
          A[i + 1] = A[i]                     // 1 Zuweisung + 1 Addition + 2 Feldzugriffe
          i = i − 1                           // 1 Zuweisung + 1 Subtraktion
      A[i + 1] = key                          // 1 Zuweisung + 1 Addition + 1 Feldzugriff
```

13

9

Wie lang braucht dieser Algorithmus?
Wir zählen **alle Rechenschritte**.

Schlechtester Fall.
Array ist absteigend sortiert.

9	8	7	6	5	4	3	2	1

$\Rightarrow A[i] > key$ **immer** erfüllt

$\Rightarrow$ für jedes j gibt es 13 $+$ Rechenschr.

Laufzeit von InsertionSort

```
InsertionSort(int[] A)
  for j = 2 to A.length do          // 1 Zuweisung + 1 Addition + 1 Vergleich
    key = A[j]                      // 1 Zuweisung + 1 Feldzugriff
    i = j − 1                       // 1 Zuweisung + 1 Subtraktion
    while i > 0 and A[i] > key do   // 2 Vergleiche + 1 Feldzugriff
      A[i + 1] = A[i]               // 1 Zuweisung + 1 Addition + 2 Feldzugriffe
      i = i − 1                     // 1 Zuweisung + 1 Subtraktion
    A[i + 1] = key                  // 1 Zuweisung + 1 Addition + 1 Feldzugriff
```

13

9

Wie lang braucht dieser Algorithmus?
Wir zählen **alle Rechenschritte**.

Schlechtester Fall.
Array ist absteigend sortiert.

9	8	7	6	5	4	3	2	1

$\Rightarrow A[i] > key$ **immer** erfüllt

$\Rightarrow$ für jedes j gibt es $13 + 9(j-1)$ Rechenschr.

Laufzeit von InsertionSort

```
InsertionSort(int[] A)
  for j = 2 to A.length do              // 1 Zuweisung + 1 Addition + 1 Vergleich
      key = A[j]                        // 1 Zuweisung + 1 Feldzugriff
      i = j − 1                         // 1 Zuweisung + 1 Subtraktion
      while i > 0 and A[i] > key do     // 2 Vergleiche + 1 Feldzugriff
          A[i + 1] = A[i]               // 1 Zuweisung + 1 Addition + 2 Feldzugriffe
          i = i − 1                     // 1 Zuweisung + 1 Subtraktion
      A[i + 1] = key                    // 1 Zuweisung + 1 Addition + 1 Feldzugriff
```

$\boxed{13}$ $\boxed{9}$

Wie lang braucht dieser Algorithmus?
Wir zählen **alle Rechenschritte**.

Schlechtester Fall.
Array ist absteigend sortiert.

9	8	7	6	5	4	3	2	1

$\Rightarrow A[i] > key$ **immer** erfüllt

$\Rightarrow$ für jedes j gibt es $13 + 9(j-1)$ Rechenschr.

$\Rightarrow$ Laufzeit $\sum_{j=2}^{n}(13 + 9(j-1))$

Laufzeit von InsertionSort

```
InsertionSort(int[] A)
  for j = 2 to A.length do              // 1 Zuweisung + 1 Addition + 1 Vergleich
    key = A[j]                          // 1 Zuweisung + 1 Feldzugriff
    i = j − 1                           // 1 Zuweisung + 1 Subtraktion
    while i > 0 and A[i] > key do       // 2 Vergleiche + 1 Feldzugriff
      A[i + 1] = A[i]                   // 1 Zuweisung + 1 Addition + 2 Feldzugriffe
      i = i − 1                         // 1 Zuweisung + 1 Subtraktion
    A[i + 1] = key                      // 1 Zuweisung + 1 Addition + 1 Feldzugriff
```

$\boxed{13}$ $\boxed{9}$

Wie lang braucht dieser Algorithmus?
Wir zählen **alle Rechenschritte**.

Schlechtester Fall.
Array ist absteigend sortiert.

9	8	7	6	5	4	3	2	1

$\Rightarrow A[i] > key$ **immer** erfüllt

$\Rightarrow$ für jedes j gibt es $13 + 9(j-1)$ Rechenschr.

$\Rightarrow$ Laufzeit $\sum_{j=2}^{n}(13 + 9(j-1)) = \sum_{j=1}^{n-1}(13 + 9j)$

Laufzeit von InsertionSort

```
InsertionSort(int[] A)
  for j = 2 to A.length do                 // 1 Zuweisung + 1 Addition + 1 Vergleich
    key = A[j]                             // 1 Zuweisung + 1 Feldzugriff
    i = j − 1                              // 1 Zuweisung + 1 Subtraktion
    while i > 0 and A[i] > key do          // 2 Vergleiche + 1 Feldzugriff
      A[i + 1] = A[i]                      // 1 Zuweisung + 1 Addition + 2 Feldzugriffe
      i = i − 1                            // 1 Zuweisung + 1 Subtraktion
    A[i + 1] = key                         // 1 Zuweisung + 1 Addition + 1 Feldzugriff
```

$\boxed{13}$ $\boxed{9}$

Wie lang braucht dieser Algorithmus?
Wir zählen **alle Rechenschritte**.

Schlechtester Fall.
Array ist absteigend sortiert.

9	8	7	6	5	4	3	2	1

$\Rightarrow$ $A[i] > key$ **immer** erfüllt

$\Rightarrow$ für jedes j gibt es $13 + 9(j-1)$ Rechenschr.

$\Rightarrow$ Laufzeit $\sum_{j=2}^{n}(13 + 9(j-1)) = \sum_{j=1}^{n-1}(13 + 9j) = 13(n-1) + 9\sum_{j=1}^{n-1} j$

Laufzeit von InsertionSort

```
InsertionSort(int[] A)
  for j = 2 to A.length do              // 1 Zuweisung + 1 Addition + 1 Vergleich
    key = A[j]                          // 1 Zuweisung + 1 Feldzugriff
    i = j − 1                           // 1 Zuweisung + 1 Subtraktion
    while i > 0 and A[i] > key do       // 2 Vergleiche + 1 Feldzugriff
      A[i + 1] = A[i]                   // 1 Zuweisung + 1 Addition + 2 Feldzugriffe
      i = i − 1                         // 1 Zuweisung + 1 Subtraktion
    A[i + 1] = key                      // 1 Zuweisung + 1 Addition + 1 Feldzugriff
```

13

9

Wie lang braucht dieser Algorithmus?
Wir zählen **alle Rechenschritte**.

Schlechtester Fall.
Array ist absteigend sortiert.

9	8	7	6	5	4	3	2	1

$\Rightarrow A[i] > key$ **immer** erfüllt

$\Rightarrow$ für jedes j gibt es $13 + 9(j-1)$ Rechenschr.

$\Rightarrow$ Laufzeit $\sum_{j=2}^{n}(13 + 9(j-1)) = \sum_{j=1}^{n-1}(13 + 9j) = 13(n-1) + 9\sum_{j=1}^{n-1} j$

1) $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$ **arithmetische Reihe**

Laufzeit von InsertionSort

```
InsertionSort(int[] A)
  for j = 2 to A.length do                  // 1 Zuweisung + 1 Addition + 1 Vergleich
    key = A[j]                              // 1 Zuweisung + 1 Feldzugriff
    i = j − 1                               // 1 Zuweisung + 1 Subtraktion
    while i > 0 and A[i] > key do           // 2 Vergleiche + 1 Feldzugriff
      A[i + 1] = A[i]                       // 1 Zuweisung + 1 Addition + 2 Feldzugriffe
      i = i − 1                             // 1 Zuweisung + 1 Subtraktion
    A[i + 1] = key                          // 1 Zuweisung + 1 Addition + 1 Feldzugriff
```

$\boxed{13}$ $\boxed{9}$

Wie lang braucht dieser Algorithmus?
Wir zählen **alle Rechenschritte**.

Schlechtester Fall.
Array ist absteigend sortiert.

9	8	7	6	5	4	3	2	1

$\Rightarrow A[i] > key$ **immer** erfüllt

$\Rightarrow$ für jedes j gibt es $13 + 9(j-1)$ Rechenschr.

$\Rightarrow$ Laufzeit $\sum_{j=2}^{n}(13 + 9(j-1)) = \sum_{j=1}^{n-1}(13 + 9j) = 13(n-1) + 9\sum_{j=1}^{n-1} j$

$= 13(n-1) + 9 \cdot \frac{n(n-1)}{2}$

1) $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$ **arithmetische Reihe**

Laufzeit von InsertionSort

```
InsertionSort(int[] A)
  for j = 2 to A.length do            // 1 Zuweisung + 1 Addition + 1 Vergleich
      key = A[j]                      // 1 Zuweisung + 1 Feldzugriff
      i = j − 1                       // 1 Zuweisung + 1 Subtraktion
      while i > 0 and A[i] > key do   // 2 Vergleiche + 1 Feldzugriff
          A[i + 1] = A[i]             // 1 Zuweisung + 1 Addition + 2 Feldzugriffe
          i = i − 1                   // 1 Zuweisung + 1 Subtraktion
      A[i + 1] = key                  // 1 Zuweisung + 1 Addition + 1 Feldzugriff
```

13

9

Wie lang braucht dieser Algorithmus?
Wir zählen **alle Rechenschritte**.

Schlechtester Fall.
Array ist absteigend sortiert.

9	8	7	6	5	4	3	2	1

$\Rightarrow$ $A[i] > key$ **immer** erfüllt

$\Rightarrow$ für jedes j gibt es $13 + 9(j-1)$ Rechenschr.

$\Rightarrow$ Laufzeit $\sum_{j=2}^{n}(13 + 9(j-1)) = \sum_{j=1}^{n-1}(13 + 9j) = 13(n-1) + 9\sum_{j=1}^{n-1} j$

$= 13(n-1) + 9 \cdot \frac{n(n-1)}{2} = \frac{(9n+26)(n-1)}{2}$

1) $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$ **arithmetische Reihe**

Laufzeit von InsertionSort

```
InsertionSort(int[] A)
  for j = 2 to A.length do               // 1 Zuweisung + 1 Addition + 1 Vergleich
      key = A[j]                         // 1 Zuweisung + 1 Feldzugriff
      i = j − 1                          // 1 Zuweisung + 1 Subtraktion
      while i > 0 and A[i] > key do      // 2 Vergleiche + 1 Feldzugriff
          A[i + 1] = A[i]                // 1 Zuweisung + 1 Addition + 2 Feldzugriffe
          i = i − 1                      // 1 Zuweisung + 1 Subtraktion
      A[i + 1] = key                     // 1 Zuweisung + 1 Addition + 1 Feldzugriff
```

13

9

Wie lang braucht dieser Algorithmus?
Wir zählen **alle Rechenschritte**.

Schlechtester Fall.
Array ist absteigend sortiert.

9	8	7	6	5	4	3	2	1

$\Rightarrow$ $A[i] > key$ **immer** erfüllt

$\Rightarrow$ für jedes j gibt es $13 + 9(j-1)$ Rechenschr.

$\Rightarrow$ Laufzeit $\sum_{j=2}^{n}(13 + 9(j-1)) = \sum_{j=1}^{n-1}(13 + 9j) = 13(n-1) + 9\sum_{j=1}^{n-1} j$

$= 13(n-1) + 9 \cdot \frac{n(n-1)}{2} = \frac{(9n+26)(n-1)}{2}$ ☹

1) $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$ **arithmetische Reihe**

Ein Klassifikationsschema für Funktionen

Definition.

Sei $g\colon \mathbb{N} \to \mathbb{R}$ eine Funktion.

Ein Klassifikationsschema für Funktionen

Definition.

Sei $g\colon \mathbb{N} \to \mathbb{R}$ eine Funktion.

Menge der **natürlichen Zahlen** $0, 1, 2, \ldots$

Ein Klassifikationsschema für Funktionen

Definition.

Menge der **reellen** Zahlen, z.B. -7, 3, $\frac{2}{9}$, $\sqrt{2}$, e, π^2.

Sei $g \colon \mathbb{N} \to \mathbb{R}$ eine Funktion.

Menge der **natürlichen Zahlen** $0, 1, 2, \ldots$

Ein Klassifikationsschema für Funktionen

Definition.

Sei $g \colon \mathbb{N} \to \mathbb{R}$ eine Funktion. Dann ist „Groß-Oh von g"

$$\mathcal{O}(g) = \left\{ f \colon \mathbb{N} \to \mathbb{R} \;\middle|\; \qquad \right\}$$

die Klasse der Fkt., die **höchstens** so schnell wachsen wie g.

Ein Klassifikationsschema für Funktionen

Definition.

Sei $g\colon \mathbb{N} \to \mathbb{R}$ eine Funktion. Dann ist „Groß-Oh von g“

$$\mathcal{O}(g) = \left\{ f\colon \mathbb{N} \to \mathbb{R} \;\middle|\; \begin{array}{c} \text{es gibt positive Konstanten } c \text{ und } n_0, \\ \text{so dass für alle } n \geq n_0 \text{ gilt:} \\ f(n) \leq c \cdot g(n) \end{array} \right\}$$

die Klasse der Fkt., die **höchstens** so schnell wachsen wie g.

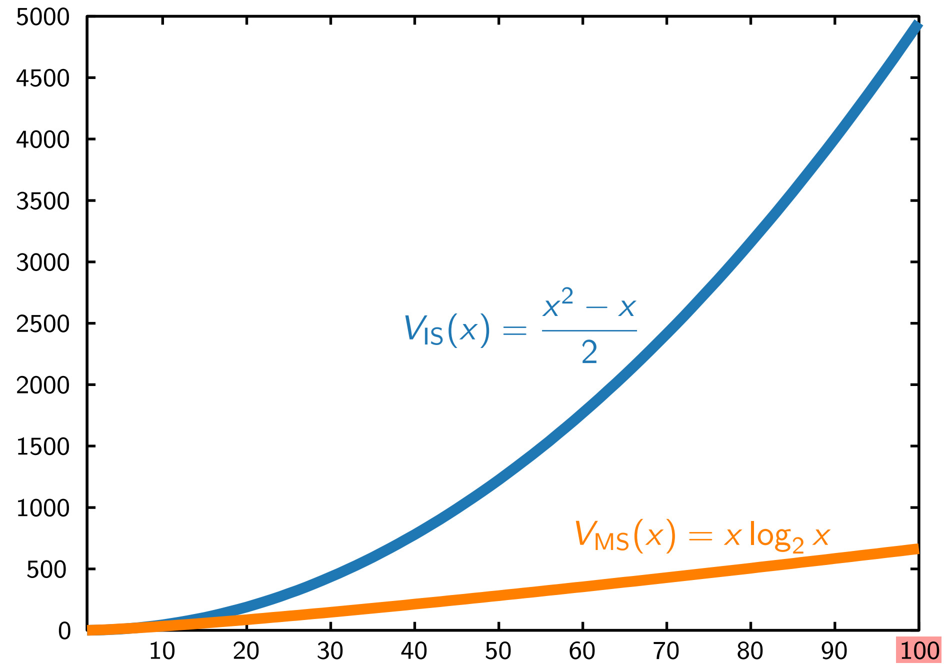
Ein Klassifikationsschema für Funktionen

Definition.

Sei $g\colon \mathbb{N} \to \mathbb{R}$ eine Funktion. Dann ist „Groß-Oh von g“

$$\mathcal{O}(g) = \left\{ f\colon \mathbb{N} \to \mathbb{R} \;\middle|\; \begin{array}{c} \text{es gibt positive Konstanten } c \text{ und } n_0, \\ \text{so dass für alle } n \geq n_0 \text{ gilt:} \\ f(n) \leq c \cdot g(n) \end{array} \right\}$$

die Klasse der Fkt., die **höchstens** so schnell wachsen wie g.



Ein Klassifikationsschema für Funktionen

Definition.

Sei $g\colon \mathbb{N} \to \mathbb{R}$ eine Funktion. Dann ist „Groß-Oh von g“

$$\mathcal{O}(g) = \left\{ f\colon \mathbb{N} \to \mathbb{R} \;\middle|\; \begin{array}{c} \text{es gibt positive Konstanten } c \text{ und } n_0, \\ \text{so dass für alle } n \geq n_0 \text{ gilt:} \\ f(n) \leq c \cdot g(n) \end{array} \right\}$$

die Klasse der Fkt., die **höchstens** so schnell wachsen wie g.

Beispiel. $f(n) = 2n^2 + 4n - 20$

Ein Klassifikationsschema für Funktionen

Definition.

Sei $g\colon \mathbb{N} \to \mathbb{R}$ eine Funktion. Dann ist „Groß-Oh von g“

$$\mathcal{O}(g) = \left\{ f\colon \mathbb{N} \to \mathbb{R} \;\middle|\; \begin{array}{c} \text{es gibt positive Konstanten } c \text{ und } n_0, \\ \text{so dass für alle } n \geq n_0 \text{ gilt:} \\ f(n) \leq c \cdot g(n) \end{array} \right\}$$

die Klasse der Fkt., die **höchstens** so schnell wachsen wie g.

Beispiel. $f(n) = 2n^2 + 4n - 20$

Behauptung: $f \in \mathcal{O}(n^2)$

Ein Klassifikationsschema für Funktionen

Definition.

Sei $g\colon \mathbb{N} \to \mathbb{R}$ eine Funktion. Dann ist „Groß-Oh von g“

$$\mathcal{O}(g) = \left\{ f\colon \mathbb{N} \to \mathbb{R} \;\middle|\; \begin{array}{c} \text{es gibt positive Konstanten } c \text{ und } n_0, \\ \text{so dass für alle } n \geq n_0 \text{ gilt:} \\ f(n) \leq c \cdot g(n) \end{array} \right\}$$

die Klasse der Fkt., die **höchstens** so schnell wachsen wie g.

Beispiel. $f(n) = 2n^2 + 4n - 20$

Behauptung: $f \in \mathcal{O}(n^2)$; m.a.W. f wächst höchstens quadratisch.

Ein Klassifikationsschema für Funktionen

Definition.

Sei $g\colon \mathbb{N} \to \mathbb{R}$ eine Funktion. Dann ist „Groß-Oh von g“

$$\mathcal{O}(g) = \left\{ f\colon \mathbb{N} \to \mathbb{R} \;\middle|\; \begin{array}{c} \text{es gibt positive Konstanten } c \text{ und } n_0, \\ \text{so dass für alle } n \geq n_0 \text{ gilt:} \\ f(n) \leq c \cdot g(n) \end{array} \right\}$$

die Klasse der Fkt., die **höchstens** so schnell wachsen wie g.

Beispiel. $f(n) = 2n^2 + 4n - 20$

Behauptung: $f \in \mathcal{O}(n^2)$; m.a.W. f wächst höchstens quadratisch.

Beweis. **Wähle** positive c und n_0,

Ein Klassifikationsschema für Funktionen

Definition.

Sei $g\colon \mathbb{N} \to \mathbb{R}$ eine Funktion. Dann ist „Groß-Oh von g“

$$\mathcal{O}(g) = \left\{ f\colon \mathbb{N} \to \mathbb{R} \;\middle|\; \begin{array}{c} \text{es gibt positive Konstanten } c \text{ und } n_0, \\ \text{so dass für alle } n \geq n_0 \text{ gilt:} \\ f(n) \leq c \cdot g(n) \end{array} \right\}$$

die Klasse der Fkt., die **höchstens** so schnell wachsen wie g.

Beispiel. $f(n) = 2n^2 + 4n - 20$

Behauptung: $f \in \mathcal{O}(n^2)$; m.a.W. f wächst höchstens quadratisch.

Beweis. **Wähle** positive c und n_0, so dass für alle $n \geq n_0$ gilt: $f(n) \leq c \cdot n^2$.

Ein Klassifikationsschema für Funktionen

Definition.

Sei $g\colon \mathbb{N} \to \mathbb{R}$ eine Funktion. Dann ist „Groß-Oh von g“

$$\mathcal{O}(g) = \left\{ f\colon \mathbb{N} \to \mathbb{R} \;\middle|\; \begin{array}{c} \text{es gibt positive Konstanten } c \text{ und } n_0, \\ \text{so dass für alle } n \geq n_0 \text{ gilt:} \\ f(n) \leq c \cdot g(n) \end{array} \right\}$$

die Klasse der Fkt., die **höchstens** so schnell wachsen wie g.

Beispiel. $f(n) = 2n^2 + 4n - 20$

Behauptung: $f \in \mathcal{O}(n^2)$; m.a.W. f wächst höchstens quadratisch.

Beweis. **Wähle** positive c und n_0, so dass für alle $n \geq n_0$ gilt: $f(n) \leq c \cdot n^2$.

Ein Klassifikationsschema für Funktionen

Definition.

Sei $g\colon \mathbb{N} \to \mathbb{R}$ eine Funktion. Dann ist „Groß-Oh von g“

$$\mathcal{O}(g) = \left\{ f\colon \mathbb{N} \to \mathbb{R} \;\middle|\; \begin{array}{c} \text{es gibt positive Konstanten } c \text{ und } n_0, \\ \text{so dass für alle } n \geq n_0 \text{ gilt:} \\ f(n) \leq c \cdot g(n) \end{array} \right\}$$

die Klasse der Fkt., die **höchstens** so schnell wachsen wie g.

Beispiel. $f(n) = 2n^2 + 4n - 20$

Behauptung: $f \in \mathcal{O}(n^2)$; m.a.W. f wächst höchstens quadratisch.

Beweis. **Wähle** positive c und n_0, so dass für alle $n \geq n_0$ gilt: $f(n) \leq c \cdot n^2$.

$$f(n) = 2n^2 + 4n - 20 \quad \leq$$

Ein Klassifikationsschema für Funktionen

Definition.

Sei $g\colon \mathbb{N} \to \mathbb{R}$ eine Funktion. Dann ist „Groß-Oh von g“

$$\mathcal{O}(g) = \left\{ f\colon \mathbb{N} \to \mathbb{R} \;\middle|\; \begin{array}{c} \text{es gibt positive Konstanten } c \text{ und } n_0, \\ \text{so dass für alle } n \geq n_0 \text{ gilt:} \\ f(n) \leq c \cdot g(n) \end{array} \right\}$$

die Klasse der Fkt., die **höchstens** so schnell wachsen wie g.

Beispiel. $f(n) = 2n^2 + 4n - 20$

Behauptung: $f \in \mathcal{O}(n^2)$; m.a.W. f wächst höchstens quadratisch.

Beweis. **Wähle** positive c und n_0, so dass für alle $n \geq n_0$ gilt: $f(n) \leq c \cdot n^2$.

$$f(n) = 2n^2 + 4n - 20 \quad \leq \quad 6n^2$$

Ein Klassifikationsschema für Funktionen

Definition.

Sei $g\colon \mathbb{N} \to \mathbb{R}$ eine Funktion. Dann ist „Groß-Oh von g“

$$\mathcal{O}(g) = \left\{ f\colon \mathbb{N} \to \mathbb{R} \;\middle|\; \begin{array}{c} \text{es gibt positive Konstanten } c \text{ und } n_0, \\ \text{so dass für alle } n \geq n_0 \text{ gilt:} \\ f(n) \leq c \cdot g(n) \end{array} \right\}$$

die Klasse der Fkt., die **höchstens** so schnell wachsen wie g.

Beispiel. $f(n) = 2n^2 + 4n - 20$

Behauptung: $f \in \mathcal{O}(n^2)$; m.a.W. f wächst höchstens quadratisch.

Beweis. **Wähle** positive c und n_0, so dass für alle $n \geq n_0$ gilt: $f(n) \leq c \cdot n^2$.

da $4n \leq 4n^2$

$$f(n) = 2n^2 + 4n - 20 \leq 6n^2$$

Ein Klassifikationsschema für Funktionen

Definition.

Sei $g \colon \mathbb{N} \to \mathbb{R}$ eine Funktion. Dann ist „Groß-Oh von g“

$$\mathcal{O}(g) = \left\{ f \colon \mathbb{N} \to \mathbb{R} \;\middle|\; \begin{array}{c} \text{es gibt positive Konstanten } c \text{ und } n_0, \\ \text{so dass für alle } n \geq n_0 \text{ gilt:} \\ f(n) \leq c \cdot g(n) \end{array} \right\}$$

die Klasse der Fkt., die **höchstens** so schnell wachsen wie g.

Beispiel. $f(n) = 2n^2 + 4n - 20$

Behauptung: $f \in \mathcal{O}(n^2)$; m.a.W. f wächst höchstens quadratisch.

Beweis. **Wähle** positive c und n_0, so dass für alle $n \geq n_0$ gilt: $f(n) \leq c \cdot n^2$.

$$f(n) = 2n^2 + 4n - 20 \underbrace{\leq}_{\text{da } 4n \leq 4n^2} 6n^2 \Rightarrow$$

Ein Klassifikationsschema für Funktionen

Definition.
Sei $g \colon \mathbb{N} \to \mathbb{R}$ eine Funktion. Dann ist „Groß-Oh von g“

$$\mathcal{O}(g) = \left\{ f \colon \mathbb{N} \to \mathbb{R} \;\middle|\; \begin{array}{c} \text{es gibt positive Konstanten } c \text{ und } n_0, \\ \text{so dass für alle } n \geq n_0 \text{ gilt:} \\ f(n) \leq c \cdot g(n) \end{array} \right\}$$

die Klasse der Fkt., die **höchstens** so schnell wachsen wie g.

Beispiel. $f(n) = 2n^2 + 4n - 20$

Behauptung: $f \in \mathcal{O}(n^2)$; m.a.W. f wächst höchstens quadratisch.

Beweis. **Wähle** positive c und n_0, so dass für alle $n \geq n_0$ gilt: $f(n) \leq c \cdot n^2$.

da $4n \leq 4n^2$

$$f(n) = 2n^2 + 4n - 20 \leq 6n^2 \Rightarrow \text{wähle } c = 6.$$

Ein Klassifikationsschema für Funktionen

Definition.
Sei $g\colon \mathbb{N} \to \mathbb{R}$ eine Funktion. Dann ist „Groß-Oh von g“

$$\mathcal{O}(g) = \left\{ f\colon \mathbb{N} \to \mathbb{R} \;\middle|\; \begin{array}{c} \text{es gibt positive Konstanten } c \text{ und } n_0, \\ \text{so dass für alle } n \geq n_0 \text{ gilt:} \\ f(n) \leq c \cdot g(n) \end{array} \right\}$$

die Klasse der Fkt., die **höchstens** so schnell wachsen wie g.

Beispiel. $f(n) = 2n^2 + 4n - 20$

Behauptung: $f \in \mathcal{O}(n^2)$; m.a.W. f wächst höchstens quadratisch.

Beweis. **Wähle** positive c und n_0, so dass für alle $n \geq n_0$ gilt: $f(n) \leq c \cdot n^2$.

da $4n \leq 4n^2$

$$f(n) = 2n^2 + 4n - 20 \leq 6n^2 \Rightarrow \text{wähle } c = 6.$$

Welches n_0?

Ein Klassifikationsschema für Funktionen

Definition.
Sei $g \colon \mathbb{N} \to \mathbb{R}$ eine Funktion. Dann ist „Groß-Oh von g"

$$\mathcal{O}(g) = \left\{ f \colon \mathbb{N} \to \mathbb{R} \;\middle|\; \begin{array}{c} \text{es gibt positive Konstanten } c \text{ und } n_0, \\ \text{so dass für alle } n \geq n_0 \text{ gilt:} \\ f(n) \leq c \cdot g(n) \end{array} \right\}$$

die Klasse der Fkt., die **höchstens** so schnell wachsen wie g.

Beispiel. $f(n) = 2n^2 + 4n - 20$

Behauptung: $f \in \mathcal{O}(n^2)$; m.a.W. f wächst höchstens quadratisch.

Beweis. **Wähle** positive c und n_0, so dass für alle $n \geq n_0$ gilt: $f(n) \leq c \cdot n^2$.

da $4n \leq 4n^2$

$$f(n) = 2n^2 + 4n - 20 \leq 6n^2 \Rightarrow \text{wähle } c = 6.$$

Welches n_0? Aussage gilt für jedes $n \geq 0$.

Ein Klassifikationsschema für Funktionen

Definition.

Sei $g \colon \mathbb{N} \to \mathbb{R}$ eine Funktion. Dann ist „Groß-Oh von g“

$$\mathcal{O}(g) = \left\{ f \colon \mathbb{N} \to \mathbb{R} \;\middle|\; \begin{array}{c} \text{es gibt positive Konstanten } c \text{ und } n_0, \\ \text{so dass für alle } n \geq n_0 \text{ gilt:} \\ f(n) \leq c \cdot g(n) \end{array} \right\}$$

die Klasse der Fkt., die **höchstens** so schnell wachsen wie g.

Beispiel. $f(n) = 2n^2 + 4n - 20$

Behauptung: $f \in \mathcal{O}(n^2)$; m.a.W. f wächst höchstens quadratisch.

Beweis. **Wähle** positive c und n_0, so dass für alle $n \geq n_0$ gilt: $f(n) \leq c \cdot n^2$.

da $4n \leq 4n^2$

$f(n) = 2n^2 + 4n - 20 \leq 6n^2 \Rightarrow$ wähle $c = 6$.

Welches n_0? Aussage gilt für jedes $n \geq 0$. Nimm z.B. $n_0 = 1$. $\square$

Ein Klassifikationsschema für Funktionen (I)

Definition.

Sei $g \colon \mathbb{N} \to \mathbb{R}$ eine Funktion. Dann ist „Groß-Oh von g“

$$\mathcal{O}(g) = \left\{ f \colon \mathbb{N} \to \mathbb{R} \;\middle|\; \begin{array}{c} \text{es gibt positive Konstanten } c \text{ und } n_0, \\ \text{so dass für alle } n \geq n_0 \text{ gilt:} \\ f(n) \leq c \cdot g(n) \end{array} \right\}$$

die Klasse der Fkt., die **höchstens** so schnell wachsen wie g.

Beispiel. $f(n) = 2n^2 + 4n - 20$

Behauptung: $f \notin \mathcal{O}(n)$

Ein Klassifikationsschema für Funktionen (I)

Definition.

Sei $g\colon \mathbb{N} \to \mathbb{R}$ eine Funktion. Dann ist „Groß-Oh von g“

$$\mathcal{O}(g) = \left\{ f\colon \mathbb{N} \to \mathbb{R} \;\middle|\; \begin{array}{c} \text{es gibt positive Konstanten } c \text{ und } n_0, \\ \text{so dass für alle } n \geq n_0 \text{ gilt:} \\ f(n) \leq c \cdot g(n) \end{array} \right\}$$

die Klasse der Fkt., die **höchstens** so schnell wachsen wie g.

Beispiel. $f(n) = 2n^2 + 4n - 20$

Behauptung: $f \notin \mathcal{O}(n)$; m.a.W. f wächst schneller als linear.

Ein Klassifikationsschema für Funktionen (I)

Definition.

Sei $g \colon \mathbb{N} \to \mathbb{R}$ eine Funktion. Dann ist „Groß-Oh von g“

$$\mathcal{O}(g) = \left\{ f \colon \mathbb{N} \to \mathbb{R} \;\middle|\; \begin{array}{c} \text{es gibt positive Konstanten } c \text{ und } n_0, \\ \text{so dass für alle } n \geq n_0 \text{ gilt:} \\ f(n) \leq c \cdot g(n) \end{array} \right\}$$

die Klasse der Fkt., die **höchstens** so schnell wachsen wie g.

Beispiel. $f(n) = 2n^2 + 4n - 20$

Behauptung: $f \notin \mathcal{O}(n)$; m.a.W. f wächst schneller als linear.

Beweis. Zeige:

Ein Klassifikationsschema für Funktionen (I)

Definition.

Sei $g \colon \mathbb{N} \to \mathbb{R}$ eine Funktion. Dann ist „Groß-Oh von g"

$$\mathcal{O}(g) = \left\{ f \colon \mathbb{N} \to \mathbb{R} \;\middle|\; \boxed{\begin{array}{c}\text{es gibt positive Konstanten } c \text{ und } n_0,\\ \text{so dass für alle } n \geq n_0 \text{ gilt:}\\ f(n) \leq c \cdot g(n)\end{array}} \right\}$$

die Klasse der Fkt., die **höchstens** so schnell wachsen wie g.

Beispiel. $f(n) = 2n^2 + 4n - 20$

Behauptung: $f \notin \mathcal{O}(n)$; m.a.W. f wächst schneller als linear.

Beweis. Zeige:

Ein Klassifikationsschema für Funktionen (I)

Definition.

Sei $g\colon \mathbb{N} \to \mathbb{R}$ eine Funktion. Dann ist „Groß-Oh von g“

$$\mathcal{O}(g) = \left\{ f\colon \mathbb{N} \to \mathbb{R} \;\middle|\; \boxed{\begin{array}{c}\text{es gibt positive Konstanten } c \text{ und } n_0,\\ \text{so dass für alle } n \geq n_0 \text{ gilt:}\\ f(n) \leq c \cdot g(n)\end{array}} \right\}$$

die Klasse der Fkt., die **höchstens** so schnell wachsen wie g.

Beispiel. $f(n) = 2n^2 + 4n - 20$

negiere! ($\neg$)

Behauptung: $f \notin \mathcal{O}(n)$; m.a.W. f wächst schneller als linear.

Beweis. Zeige: $\boxed{}$

Ein Klassifikationsschema für Funktionen (I)

Definition.
Sei $g \colon \mathbb{N} \to \mathbb{R}$ eine Funktion. Dann ist „Groß-Oh von g“

$$\mathcal{O}(g) = \left\{ f \colon \mathbb{N} \to \mathbb{R} \;\middle|\; \begin{array}{c} \text{es gibt positive Konstanten } c \text{ und } n_0, \\ \text{so dass für alle } n \geq n_0 \text{ gilt:} \\ f(n) \leq c \cdot g(n) \end{array} \right\}$$

die Klasse der Fkt., die **höchstens** so schnell wachsen wie g.

Beispiel. $f(n) = 2n^2 + 4n - 20$

negiere! ($\neg$)

Behauptung: $f \notin \mathcal{O}(n)$; m.a.W. f wächst schneller als linear.

Beweis. Zeige:

Ein Klassifikationsschema für Funktionen (I)

Definition.

Sei $g\colon \mathbb{N} \to \mathbb{R}$ eine Funktion. Dann ist „Groß-Oh von g“

$$\mathcal{O}(g) = \left\{ f\colon \mathbb{N} \to \mathbb{R} \;\middle|\; \begin{array}{c} \text{es gibt positive Konstanten } c \text{ und } n_0, \\ \text{so dass für alle } n \geq n_0 \text{ gilt:} \\ f(n) \leq c \cdot g(n) \end{array} \right\}$$

die Klasse der Fkt., die **höchstens** so schnell wachsen wie g.

negiere! ($\neg$)

Beispiel. $f(n) = 2n^2 + 4n - 20$

Behauptung: $f \notin \mathcal{O}(n)$; m.a.W. f wächst schneller als linear.

Beweis. Zeige: für alle pos. Konst. c und n_0

Ein Klassifikationsschema für Funktionen (I)

Definition.

Sei $g\colon \mathbb{N} \to \mathbb{R}$ eine Funktion. Dann ist „Groß-Oh von g“

$$\mathcal{O}(g) = \left\{ f\colon \mathbb{N} \to \mathbb{R} \;\middle|\; \begin{array}{c} \text{es gibt positive Konstanten } c \text{ und } n_0, \\ \text{so dass für alle } n \geq n_0 \text{ gilt:} \\ f(n) \leq c \cdot g(n) \end{array} \right\}$$

die Klasse der Fkt., die **höchstens** so schnell wachsen wie g.

Beispiel. $f(n) = 2n^2 + 4n - 20$

negiere! ($\neg$)

Behauptung: $f \notin \mathcal{O}(n)$; m.a.W. f wächst schneller als linear.

Beweis. Zeige: für alle pos. Konst. c und n_0

Ein Klassifikationsschema für Funktionen (I)

Definition.

Sei $g \colon \mathbb{N} \to \mathbb{R}$ eine Funktion. Dann ist „Groß-Oh von g“

$$\mathcal{O}(g) = \left\{ f \colon \mathbb{N} \to \mathbb{R} \;\middle|\; \begin{array}{c} \text{es gibt positive Konstanten } c \text{ und } n_0, \\ \text{so dass für alle } n \geq n_0 \text{ gilt:} \\ f(n) \leq c \cdot g(n) \end{array} \right\}$$

die Klasse der Fkt., die **höchstens** so schnell wachsen wie g.

Beispiel. $f(n) = 2n^2 + 4n - 20$

negiere! ($\neg$)

Behauptung: $f \notin \mathcal{O}(n)$; m.a.W. f wächst schneller als linear.

Beweis. Zeige: für alle pos. Konst. c und n_0 gibt es ein $n \geq n_0$,

Ein Klassifikationsschema für Funktionen (I)

Definition.

Sei $g\colon \mathbb{N} \to \mathbb{R}$ eine Funktion. Dann ist „Groß-Oh von g“

$$\mathcal{O}(g) = \left\{ f\colon \mathbb{N} \to \mathbb{R} \;\middle|\; \begin{array}{c} \text{es gibt positive Konstanten } c \text{ und } n_0, \\ \text{so dass für alle } n \geq n_0 \text{ gilt:} \\ f(n) \leq c \cdot g(n) \end{array} \right\}$$

die Klasse der Fkt., die **höchstens** so schnell wachsen wie g.

negiere! ($\neg$)

Beispiel. $f(n) = 2n^2 + 4n - 20$

Behauptung: $f \notin \mathcal{O}(n)$; m.a.W. f wächst schneller als linear.

Beweis. Zeige: für alle pos. Konst. c und n_0 gibt es ein $n \geq n_0$,

Ein Klassifikationsschema für Funktionen (I)

Definition.

Sei $g\colon \mathbb{N} \to \mathbb{R}$ eine Funktion. Dann ist „Groß-Oh von g“

$$\mathcal{O}(g) = \left\{ f\colon \mathbb{N} \to \mathbb{R} \;\middle|\; \begin{array}{c} \text{es gibt positive Konstanten } c \text{ und } n_0, \\ \text{so dass für alle } n \geq n_0 \text{ gilt:} \\ f(n) \leq c \cdot g(n) \end{array} \right\}$$

die Klasse der Fkt., die **höchstens** so schnell wachsen wie g.

negiere! ($\neg$)

Beispiel. $f(n) = 2n^2 + 4n - 20$

Behauptung: $f \notin \mathcal{O}(n)$; m.a.W. f wächst schneller als linear.

Beweis. Zeige: für alle pos. Konst. c und n_0 gibt es ein $n \geq n_0$, so dass $f(n) > c \cdot n$.

Ein Klassifikationsschema für Funktionen (I)

Definition.

Sei $g\colon \mathbb{N} \to \mathbb{R}$ eine Funktion. Dann ist „Groß-Oh von g“

$$\mathcal{O}(g) = \left\{ f\colon \mathbb{N} \to \mathbb{R} \;\middle|\; \begin{array}{c} \text{es gibt positive Konstanten } c \text{ und } n_0, \\ \text{so dass für alle } n \geq n_0 \text{ gilt:} \\ f(n) \leq c \cdot g(n) \end{array} \right\}$$

die Klasse der Fkt., die **höchstens** so schnell wachsen wie g.

Beispiel. $f(n) = 2n^2 + 4n - 20$

Behauptung: $f \notin \mathcal{O}(n)$; m.a.W. f wächst schneller als linear.

Beweis. Zeige: für alle pos. Konst. c und n_0 gibt es ein $n \geq n_0$, so dass $\boxed{f(n) > c \cdot n.}$

Ein Klassifikationsschema für Funktionen (I)

Definition.

Sei $g\colon \mathbb{N} \to \mathbb{R}$ eine Funktion. Dann ist „Groß-Oh von g“

$$\mathcal{O}(g) = \left\{ f\colon \mathbb{N} \to \mathbb{R} \;\middle|\; \begin{array}{c} \text{es gibt positive Konstanten } c \text{ und } n_0, \\ \text{so dass für alle } n \geq n_0 \text{ gilt:} \\ f(n) \leq c \cdot g(n) \end{array} \right\}$$

die Klasse der Fkt., die **höchstens** so schnell wachsen wie g.

Beispiel. $f(n) = 2n^2 + 4n - 20$

Behauptung: $f \notin \mathcal{O}(n)$; m.a.W. f wächst schneller als linear.

Beweis. Zeige: für alle pos. Konst. c und n_0 gibt es ein $n \geq n_0$, so dass $\boxed{f(n) > c \cdot n.}$

Also: bestimme n in Abhängigkeit von c und n_0, so dass

Ein Klassifikationsschema für Funktionen (I)

Definition.

Sei $g : \mathbb{N} \to \mathbb{R}$ eine Funktion. Dann ist „Groß-Oh von g"

$$\mathcal{O}(g) = \left\{ f : \mathbb{N} \to \mathbb{R} \;\middle|\; \begin{array}{c} \text{es gibt positive Konstanten } c \text{ und } n_0, \\ \text{so dass für alle } n \geq n_0 \text{ gilt:} \\ f(n) \leq c \cdot g(n) \end{array} \right\}$$

die Klasse der Fkt., die **höchstens** so schnell wachsen wie g.

Beispiel. $f(n) = 2n^2 + 4n - 20$

Behauptung: $f \notin \mathcal{O}(n)$; m.a.W. f wächst schneller als linear.

Beweis. Zeige: für alle pos. Konst. c und n_0 gibt es ein $n \geq n_0$, so dass $\boxed{f(n) > c \cdot n.}$

Also: bestimme n in Abhängigkeit von c und n_0, so dass

$n \geq n_0$ und

Ein Klassifikationsschema für Funktionen (I)

Definition.
Sei $g\colon \mathbb{N} \to \mathbb{R}$ eine Funktion. Dann ist „Groß-Oh von g“

$$\mathcal{O}(g) = \left\{ f\colon \mathbb{N} \to \mathbb{R} \;\middle|\; \begin{array}{c} \text{es gibt positive Konstanten } c \text{ und } n_0, \\ \text{so dass für alle } n \geq n_0 \text{ gilt:} \\ f(n) \leq c \cdot g(n) \end{array} \right\}$$

die Klasse der Fkt., die **höchstens** so schnell wachsen wie g.

Beispiel. $f(n) = 2n^2 + 4n - 20$

Behauptung: $f \notin \mathcal{O}(n)$; m.a.W. f wächst schneller als linear.

Beweis. Zeige: für alle pos. Konst. c und n_0 gibt es ein $n \geq n_0$, so dass $\boxed{f(n) > c \cdot n.}$

Also: bestimme n in Abhängigkeit von c und n_0, so dass

$n \geq n_0$ und $f(n) = 2n^2 + 4n - 20 \;>\; c \cdot n$.

Fortsetzung des Beweises $f \notin \mathcal{O}(n)$

Bestimme n in Abhängigkeit von c und n_0, so dass $n \geq n_0$ und

$$f(n) = 2n^2 + 4n - 20 \; > \; c \cdot n.$$

Fortsetzung des Beweises $f \notin \mathcal{O}(n)$

Bestimme n in Abhängigkeit von c und n_0, so dass $n \geq n_0$ und

$$f(n) = 2n^2 + 4n - 20 \;>\; c \cdot n.$$

Problem: Die „-20“ stört.

Fortsetzung des Beweises $f \notin \mathcal{O}(n)$

Bestimme n in Abhängigkeit von c und n_0, so dass $n \geq n_0$ und

$$f(n) = 2n^2 + 4n - 20 \; > \; c \cdot n.$$

Problem: Die „-20“ stört.

Aber wenn $n \geq 5$, dann ...

Fortsetzung des Beweises $f \notin \mathcal{O}(n)$

Bestimme n in Abhängigkeit von c und n_0, so dass $n \geq n_0$ und

$$f(n) = 2n^2 + 4n - 20 \;>\; c \cdot n.$$

Problem: Die „-20“ stört.

Aber wenn $n \geq 5$, dann gilt $4n - 20 \geq 0$.

Fortsetzung des Beweises $f \notin \mathcal{O}(n)$

Bestimme n in Abhängigkeit von c und n_0, so dass $n \geq n_0$ und

$$f(n) = 2n^2 + 4n - 20 > c \cdot n.$$

Problem: Die „-20“ stört.

Aber wenn $n \geq 5$, dann gilt $4n - 20 \geq 0$.

Fortsetzung des Beweises $f \notin \mathcal{O}(n)$

Bestimme n in Abhängigkeit von c und n_0, so dass $n \geq n_0$ und

$$f(n) = 2n^2 + 4n - 20 > c \cdot n.$$

Problem: Die „-20“ stört.

Aber wenn $n \geq 5$, dann gilt $4n - 20 \geq 0$.

D.h. wenn $n \geq 5$ und $2n^2 > c \cdot n$, dann ...

Fortsetzung des Beweises $f \notin \mathcal{O}(n)$

Bestimme n in Abhängigkeit von c und n_0, so dass $n \geq n_0$ und

$$f(n) = 2n^2 \boxed{+ 4n - 20} > c \cdot n.$$

Problem: Die „-20“ stört.

Aber wenn $n \geq 5$, dann gilt $4n - 20 \geq 0$.

D.h. wenn $n \geq 5$ und $2n^2 > c \cdot n$, dann $f(n) > c \cdot n$.

Fortsetzung des Beweises $f \notin \mathcal{O}(n)$

Bestimme n in Abhängigkeit von c und n_0, so dass $n \geq n_0$ und

$$f(n) = 2n^2 + 4n - 20 > c \cdot n.$$

Problem: Die „-20“ stört.

Aber wenn $n \geq 5$, dann gilt $4n - 20 \geq 0$.

D.h. wenn $n \geq 5$ und $2n^2 > c \cdot n$, dann $f(n) > c \cdot n$.

$$\Updownarrow$$

Fortsetzung des Beweises $f \notin \mathcal{O}(n)$

Bestimme n in Abhängigkeit von c und n_0, so dass $n \geq n_0$ und

$$f(n) = 2n^2 + 4n - 20 > c \cdot n.$$

Problem: Die „-20“ stört.

Aber wenn $n \geq 5$, dann gilt $4n - 20 \geq 0$.

D.h. wenn $n \geq 5$ und $2n^2 > c \cdot n$, dann $f(n) > c \cdot n$.

$$\Updownarrow$$

$$\boxed{n > c/2}$$

Fortsetzung des Beweises $f \notin \mathcal{O}(n)$

Bestimme n in Abhängigkeit von c und n_0, so dass $n \geq n_0$ und

$$f(n) = 2n^2 + 4n - 20 > c \cdot n.$$

Problem: Die „-20" stört.

Aber wenn $n \geq 5$, dann gilt $4n - 20 \geq 0$.

D.h. wenn $n \geq 5$ und $2n^2 > c \cdot n$, dann $f(n) > c \cdot n$.

$$\Updownarrow$$

$$\boxed{n > c/2}$$

$$\Uparrow$$

Fortsetzung des Beweises $f \notin \mathcal{O}(n)$

Bestimme n in Abhängigkeit von c und n_0, so dass $n \geq n_0$ und

$$f(n) = 2n^2 + 4n - 20 > c \cdot n.$$

Problem: Die „-20“ stört.

Aber wenn $n \geq 5$, dann gilt $4n - 20 \geq 0$.

D.h. wenn $n \geq 5$ und $2n^2 > c \cdot n$, dann $f(n) > c \cdot n$.

$$\Updownarrow$$

$$\boxed{n > c/2}$$

$$\Uparrow$$

Wie wär's mit $n = c$?

Fortsetzung des Beweises $f \notin \mathcal{O}(n)$

Bestimme n in Abhängigkeit von c und n_0, so dass $n \geq n_0$ und

$$f(n) = 2n^2 + 4n - 20 > c \cdot n.$$

Problem: Die „-20" stört.

Aber wenn $n \geq 5$, dann gilt $4n - 20 \geq 0$.

D.h. wenn $n \geq 5$ und $2n^2 > c \cdot n$, dann $f(n) > c \cdot n$.

$$\Updownarrow$$

$$\boxed{n > c/2}$$

$$\Uparrow$$

Wie wär's mit $n = c$?

Gut, aber ...

Fortsetzung des Beweises $f \notin \mathcal{O}(n)$

Bestimme n in Abhängigkeit von c und n_0, so dass $n \geq n_0$ und

$$f(n) = 2n^2 + 4n - 20 > c \cdot n.$$

Problem: Die „-20“ stört.

Aber wenn $n \geq 5$, dann gilt $4n - 20 \geq 0$.

D.h. wenn $n \geq 5$ und $2n^2 > c \cdot n$, dann $f(n) > c \cdot n$.

$$\Updownarrow$$

$$\boxed{n > c/2}$$

$$\Uparrow$$

Wie wär's mit $n = c$?

Gut, aber ...

Fortsetzung des Beweises $f \notin \mathcal{O}(n)$

Bestimme n in Abhängigkeit von c und n_0, so dass $n \geq n_0$ und

$$f(n) = 2n^2 + 4n - 20 > c \cdot n.$$

Problem: Die „-20“ stört.

Aber wenn $n \geq 5$, dann gilt $4n - 20 \geq 0$.

D.h. wenn $n \geq 5$ und $2n^2 > c \cdot n$, dann $f(n) > c \cdot n$.

$$\Updownarrow$$

$$\boxed{n > c/2}$$

$$\Uparrow$$

Wie wär's mit $n = c$?

Gut, aber wir müssen sicherstellen, dass auch $n \geq 5$ und $n \geq n_0$ gilt.

Fortsetzung des Beweises $f \notin \mathcal{O}(n)$

Bestimme n in Abhängigkeit von c und n_0, so dass $n \geq n_0$ und

$$f(n) = 2n^2 + 4n - 20 > c \cdot n.$$

Problem: Die „-20“ stört.

Aber wenn $n \geq 5$, dann gilt $4n - 20 \geq 0$.

D.h. wenn $n \geq 5$ und $2n^2 > c \cdot n$, dann $f(n) > c \cdot n$.

$$\Updownarrow$$

$$\boxed{n > c/2}$$

$$\Uparrow$$

Wie wär's mit $n = c$?

Gut, aber wir müssen sicherstellen, dass auch $n \geq 5$ und $n \geq n_0$ gilt.

Also nehmen wir $n =$

Fortsetzung des Beweises $f \notin \mathcal{O}(n)$

Bestimme n in Abhängigkeit von c und n_0, so dass $n \geq n_0$ und

$$f(n) = 2n^2 + 4n - 20 > c \cdot n.$$

Problem: Die „-20“ stört.

Aber wenn $n \geq 5$, dann gilt $4n - 20 \geq 0$.

D.h. wenn $n \geq 5$ und $2n^2 > c \cdot n$, dann $f(n) > c \cdot n$.

$$\Updownarrow$$

$$\boxed{n > c/2}$$

$$\Uparrow$$

Wie wär's mit $n = c$?

Gut, aber wir müssen sicherstellen, dass auch $n \geq 5$ und $n \geq n_0$ gilt.

Also nehmen wir $n = \lceil \max(c, 5, n_0) \rceil$.

Fortsetzung des Beweises $f \notin \mathcal{O}(n)$

Bestimme n in Abhängigkeit von c und n_0, so dass $n \geq n_0$ und

$$f(n) = 2n^2 + 4n - 20 > c \cdot n.$$

Problem: Die „-20“ stört.

Aber wenn $n \geq 5$, dann gilt $4n - 20 \geq 0$.

D.h. wenn $n \geq 5$ und $2n^2 > c \cdot n$, dann $f(n) > c \cdot n$.

$$\Updownarrow$$

$$\boxed{n > c/2}$$

$$\Uparrow$$

Wie wär's mit $n = c$?

Gut, aber wir müssen sicherstellen, dass auch $n \geq 5$ und $n \geq n_0$ gilt.

Also nehmen wir $n = \lceil \max(c, 5, n_0) \rceil$.

Für dieses n gilt $n \geq n_0$ und $f(n) > c \cdot n$.

Fortsetzung des Beweises $f \notin \mathcal{O}(n)$

Bestimme n in Abhängigkeit von c und n_0, so dass $n \geq n_0$ und

$$f(n) = 2n^2 + 4n - 20 > c \cdot n.$$

Problem: Die „-20" stört.

Aber wenn $n \geq 5$, dann gilt $4n - 20 \geq 0$.

D.h. wenn $n \geq 5$ und $2n^2 > c \cdot n$, dann $f(n) > c \cdot n$.

$$\Updownarrow$$

$$\boxed{n > c/2}$$

$$\Uparrow$$

Wie wär's mit $n = c$?

Gut, aber wir müssen sicherstellen, dass auch $n \geq 5$ und $n \geq n_0$ gilt.

Also nehmen wir $n = \lceil \max(c, 5, n_0) \rceil$.

Für dieses n gilt $n \geq n_0$ und $f(n) > c \cdot n$.

Fortsetzung des Beweises $f \notin \mathcal{O}(n)$

Bestimme n in Abhängigkeit von c und n_0, so dass $n \geq n_0$ und

$$f(n) = 2n^2 + 4n - 20 > c \cdot n.$$

Problem: Die „-20“ stört.

Aber wenn $n \geq 5$, dann gilt $4n - 20 \geq 0$.

D.h. wenn $n \geq 5$ und $2n^2 > c \cdot n$, dann $f(n) > c \cdot n$.

$$\Updownarrow$$

$$\boxed{n > c/2}$$

$$\Uparrow$$

Wie wär's mit $n = c$?

Gut, aber wir müssen sicherstellen, dass auch $n \geq 5$ und $n \geq n_0$ gilt.

Also nehmen wir $n = \lceil \max(c, 5, n_0) \rceil$.

Für dieses n gilt $n \geq n_0$ und $f(n) > c \cdot n$. Also gilt $f \notin \mathcal{O}(n)$. $\square$

Ein Klassifikationsschema für Funktionen (II)

Definition.

Sei $g\colon \mathbb{N} \to \mathbb{R}$ eine Funktion. Dann ist „Groß-Oh von g"

$$\mathcal{O}(g) = \left\{ f\colon \mathbb{N} \to \mathbb{R} \;\middle|\; \begin{array}{c} \text{es gibt positive Konstanten } c \text{ und } n_0, \\ \text{so dass für alle } n \geq n_0 \text{ gilt:} \\ f(n) \leq c \cdot g(n) \end{array} \right\}$$

die Klasse der Fkt., die **höchstens** so schnell wachsen wie g.

Ein Klassifikationsschema für Funktionen (II)

Definition.

Sei $g\colon \mathbb{N} \to \mathbb{R}$ eine Funktion. Dann ist „Groß-Oh von g“

$$\mathcal{O}(g) = \left\{ f\colon \mathbb{N} \to \mathbb{R} \;\middle|\; \begin{array}{c} \text{es gibt positive Konstanten } c \text{ und } n_0, \\ \text{so dass für alle } n \geq n_0 \text{ gilt:} \\ f(n) \leq c \cdot g(n) \end{array} \right\}$$

die Klasse der Fkt., die **höchstens** so schnell wachsen wie g.

Ein Klassifikationsschema für Funktionen (II)

Definition.

Sei $g\colon \mathbb{N} \to \mathbb{R}$ eine Funktion. Dann ist „Groß-Oh von g“

$$\mathcal{O}(g) = \left\{ f\colon \mathbb{N} \to \mathbb{R} \;\middle|\; \begin{array}{c} \text{es gibt positive Konstanten } c \text{ und } n_0, \\ \text{so dass für alle } n \geq n_0 \text{ gilt:} \\ f(n) \leq c \cdot g(n) \end{array} \right\}$$

die Klasse der Fkt., die **höchstens** so schnell wachsen wie g.

Definition.

Sei $g\colon \mathbb{N} \to \mathbb{R}$ eine Funktion. Dann ist „Groß-Omega von g“

$$\Omega(g) = \left\{ f\colon \mathbb{N} \to \mathbb{R} \;\middle|\; \begin{array}{c} \text{es gibt positive Konstanten } c \text{ und } n_0, \\ \text{so dass für alle } n \geq n_0 \text{ gilt:} \\ f(n) \geq c \cdot g(n) \end{array} \right\}$$

die Klasse der Fkt., die **mindestens** so schnell wachsen wie g.

Ein Klassifikationsschema für Funktionen (II)

Definition.

Sei $g\colon \mathbb{N} \to \mathbb{R}$ eine Funktion. Dann ist „Groß-Omega von g“

$$\Omega(g) = \left\{ f\colon \mathbb{N} \to \mathbb{R} \;\middle|\; \begin{array}{c} \text{es gibt positive Konstanten } c \text{ und } n_0, \\ \text{so dass für alle } n \geq n_0 \text{ gilt:} \\ f(n) \geq c \cdot g(n) \end{array} \right\}$$

die Klasse der Fkt., die **mindestens** so schnell wachsen wie g.

Ein Klassifikationsschema für Funktionen (II)

Definition.

Sei $g\colon \mathbb{N} \to \mathbb{R}$ eine Funktion. Dann ist „Groß-Omega von g“

$$\Omega(g) = \left\{ f\colon \mathbb{N} \to \mathbb{R} \;\middle|\; \begin{array}{c} \text{es gibt positive Konstanten } c \text{ und } n_0, \\ \text{so dass für alle } n \geq n_0 \text{ gilt:} \\ f(n) \geq c \cdot g(n) \end{array} \right\}$$

die Klasse der Fkt., die **mindestens** so schnell wachsen wie g.

Beispiel. $f(n) = 2n^2 + 4n - 20$

Ein Klassifikationsschema für Funktionen (II)

Definition.

Sei $g\colon \mathbb{N} \to \mathbb{R}$ eine Funktion. Dann ist „Groß-Omega von g“

$$\Omega(g) = \left\{ f\colon \mathbb{N} \to \mathbb{R} \;\middle|\; \begin{array}{c} \text{es gibt positive Konstanten } c \text{ und } n_0, \\ \text{so dass für alle } n \geq n_0 \text{ gilt:} \\ f(n) \geq c \cdot g(n) \end{array} \right\}$$

die Klasse der Fkt., die **mindestens** so schnell wachsen wie g.

Beispiel. $f(n) = 2n^2 + 4n - 20$

Bewiesen: $f \notin \mathcal{O}(n)$, $f \in \mathcal{O}(n^2)$

Ein Klassifikationsschema für Funktionen (II)

Definition.

Sei $g\colon \mathbb{N} \to \mathbb{R}$ eine Funktion. Dann ist „Groß-Omega von g“

$$\Omega(g) = \left\{ f\colon \mathbb{N} \to \mathbb{R} \;\middle|\; \begin{array}{c} \text{es gibt positive Konstanten } c \text{ und } n_0, \\ \text{so dass für alle } n \geq n_0 \text{ gilt:} \\ f(n) \geq c \cdot g(n) \end{array} \right\}$$

die Klasse der Fkt., die **mindestens** so schnell wachsen wie g.

Beispiel. $f(n) = 2n^2 + 4n - 20$

Bewiesen: $f \notin \mathcal{O}(n)$, $f \in \mathcal{O}(n^2)$, $f \in \mathcal{O}(n^3)$

Ein Klassifikationsschema für Funktionen (II)

Definition.

Sei $g\colon \mathbb{N} \to \mathbb{R}$ eine Funktion. Dann ist „Groß-Omega von g“

$$\Omega(g) = \left\{ f\colon \mathbb{N} \to \mathbb{R} \;\middle|\; \begin{array}{c} \text{es gibt positive Konstanten } c \text{ und } n_0, \\ \text{so dass für alle } n \geq n_0 \text{ gilt:} \\ f(n) \geq c \cdot g(n) \end{array} \right\}$$

die Klasse der Fkt., die **mindestens** so schnell wachsen wie g.

Beispiel. $f(n) = 2n^2 + 4n - 20$

Bewiesen: $f \notin \mathcal{O}(n)$, $f \in \mathcal{O}(n^2)$, $f \in \mathcal{O}(n^3)$

Entsprechend:

Ein Klassifikationsschema für Funktionen (II)

Definition.

Sei $g : \mathbb{N} \to \mathbb{R}$ eine Funktion. Dann ist „Groß-Omega von g“

$$\Omega(g) = \left\{ f : \mathbb{N} \to \mathbb{R} \;\middle|\; \begin{array}{c} \text{es gibt positive Konstanten } c \text{ und } n_0, \\ \text{so dass für alle } n \geq n_0 \text{ gilt:} \\ f(n) \geq c \cdot g(n) \end{array} \right\}$$

die Klasse der Fkt., die **mindestens** so schnell wachsen wie g.

Beispiel. $f(n) = 2n^2 + 4n - 20$

Bewiesen: $f \notin \mathcal{O}(n)$, $f \in \mathcal{O}(n^2)$, $f \in \mathcal{O}(n^3)$

Entsprechend: $f \in \Omega(n)$

Ein Klassifikationsschema für Funktionen (II)

Definition.

Sei $g \colon \mathbb{N} \to \mathbb{R}$ eine Funktion. Dann ist „Groß-Omega von g“

$$\Omega(g) = \left\{ f \colon \mathbb{N} \to \mathbb{R} \;\middle|\; \begin{array}{c} \text{es gibt positive Konstanten } c \text{ und } n_0, \\ \text{so dass für alle } n \geq n_0 \text{ gilt:} \\ f(n) \geq c \cdot g(n) \end{array} \right\}$$

die Klasse der Fkt., die **mindestens** so schnell wachsen wie g.

Beispiel. $f(n) = 2n^2 + 4n - 20$

Bewiesen: $f \notin \mathcal{O}(n)$, $f \in \mathcal{O}(n^2)$, $f \in \mathcal{O}(n^3)$

Entsprechend: $f \in \Omega(n)$, $f \in \Omega(n^2)$

Ein Klassifikationsschema für Funktionen (II)

Definition.
Sei $g\colon \mathbb{N} \to \mathbb{R}$ eine Funktion. Dann ist „Groß-Omega von g“

$$\Omega(g) = \left\{ f\colon \mathbb{N} \to \mathbb{R} \;\middle|\; \begin{array}{c} \text{es gibt positive Konstanten } c \text{ und } n_0, \\ \text{so dass für alle } n \geq n_0 \text{ gilt:} \\ f(n) \geq c \cdot g(n) \end{array} \right\}$$

die Klasse der Fkt., die **mindestens** so schnell wachsen wie g.

Beispiel. $f(n) = 2n^2 + 4n - 20$

Bewiesen: $f \notin \mathcal{O}(n)$, $f \in \mathcal{O}(n^2)$, $f \in \mathcal{O}(n^3)$

Entsprechend: $f \in \Omega(n)$, $f \in \Omega(n^2)$, $f \notin \Omega(n^3)$

Ein Klassifikationsschema für Funktionen (II)

Definition.

Sei $g \colon \mathbb{N} \to \mathbb{R}$ eine Funktion. Dann ist „Groß-Omega von g“

$$\Omega(g) = \left\{ f \colon \mathbb{N} \to \mathbb{R} \;\middle|\; \begin{array}{c} \text{es gibt positive Konstanten } c \text{ und } n_0, \\ \text{so dass für alle } n \geq n_0 \text{ gilt:} \\ f(n) \geq c \cdot g(n) \end{array} \right\}$$

die Klasse der Fkt., die **mindestens** so schnell wachsen wie g.

Beispiel. $f(n) = 2n^2 + 4n - 20$

Bewiesen: $f \notin \mathcal{O}(n)$, $f \in \mathcal{O}(n^2)$, $f \in \mathcal{O}(n^3)$

Entsprechend: $f \in \Omega(n)$, $f \in \Omega(n^2)$, $f \notin \Omega(n^3)$

Ein Klassifikationsschema für Funktionen (II)

Definition.

Sei $g\colon \mathbb{N} \to \mathbb{R}$ eine Funktion. Dann ist „Groß-Omega von g"

$$\Omega(g) = \left\{ f\colon \mathbb{N} \to \mathbb{R} \;\middle|\; \begin{array}{c} \text{es gibt positive Konstanten } c \text{ und } n_0, \\ \text{so dass für alle } n \geq n_0 \text{ gilt:} \\ f(n) \geq c \cdot g(n) \end{array} \right\}$$

die Klasse der Fkt., die **mindestens** so schnell wachsen wie g.

Beispiel. $f(n) = 2n^2 + 4n - 20$

Bewiesen: $f \notin \mathcal{O}(n)$, $f \in \mathcal{O}(n^2)$, $f \in \mathcal{O}(n^3)$

Entsprechend: $f \in \Omega(n)$, $f \in \Omega(n^2)$, $f \notin \Omega(n^3)$

Zusammen:

Ein Klassifikationsschema für Funktionen (II)

Definition.

Sei $g\colon \mathbb{N} \to \mathbb{R}$ eine Funktion. Dann ist „Groß-Omega von g“

$$\Omega(g) = \left\{ f\colon \mathbb{N} \to \mathbb{R} \;\middle|\; \begin{array}{c} \text{es gibt positive Konstanten } c \text{ und } n_0, \\ \text{so dass für alle } n \geq n_0 \text{ gilt:} \\ f(n) \geq c \cdot g(n) \end{array} \right\}$$

die Klasse der Fkt., die **mindestens** so schnell wachsen wie g.

Beispiel. $f(n) = 2n^2 + 4n - 20$

Bewiesen: $f \notin \mathcal{O}(n)$, $f \in \mathcal{O}(n^2)$, $f \in \mathcal{O}(n^3)$

Entsprechend: $f \in \Omega(n)$, $f \in \Omega(n^2)$, $f \notin \Omega(n^3)$

Zusammen: $f \in \Theta(n^2)$

Ein Klassifikationsschema für Funktionen (II)

Definition.

Sei $g\colon \mathbb{N} \to \mathbb{R}$ eine Funktion. Dann ist „Groß-Omega von g“

$$\Omega(g) = \left\{ f\colon \mathbb{N} \to \mathbb{R} \;\middle|\; \begin{array}{c} \text{es gibt positive Konstanten } c \text{ und } n_0, \\ \text{so dass für alle } n \geq n_0 \text{ gilt:} \\ f(n) \geq c \cdot g(n) \end{array} \right\}$$

die Klasse der Fkt., die **mindestens** so schnell wachsen wie g.

Beispiel. $f(n) = 2n^2 + 4n - 20$

Bewiesen: $f \notin \mathcal{O}(n)$, $f \in \mathcal{O}(n^2)$, $f \in \mathcal{O}(n^3)$

Entsprechend: $f \in \Omega(n)$, $f \in \Omega(n^2)$, $f \notin \Omega(n^3)$

Zusammen: $f \in \Theta(n^2)$

d.h. es gibt pos. Konst. c_1, c_2, n_0, so dass für alle $n \geq n_0$ gilt:

Ein Klassifikationsschema für Funktionen (II)

Definition.
Sei $g\colon \mathbb{N} \to \mathbb{R}$ eine Funktion. Dann ist „Groß-Omega von g“

$$\Omega(g) = \left\{ f\colon \mathbb{N} \to \mathbb{R} \;\middle|\; \begin{array}{c} \text{es gibt positive Konstanten } c \text{ und } n_0, \\ \text{so dass für alle } n \geq n_0 \text{ gilt:} \\ f(n) \geq c \cdot g(n) \end{array} \right\}$$

die Klasse der Fkt., die **mindestens** so schnell wachsen wie g.

Beispiel. $f(n) = 2n^2 + 4n - 20$

Bewiesen: $f \notin \mathcal{O}(n)$, $f \in \mathcal{O}(n^2)$, $f \in \mathcal{O}(n^3)$

Entsprechend: $f \in \Omega(n)$, $f \in \Omega(n^2)$, $f \notin \Omega(n^3)$

Zusammen: $f \in \Theta(n^2)$

d.h. es gibt pos. Konst. c_1, c_2, n_0, so dass für alle $n \geq n_0$ gilt: $c_1 \cdot n^2 \leq f(n) \leq c_2 \cdot n^2$.

Ein Klassifikationsschema für Funktionen (II)

Definition.

Sei $g\colon \mathbb{N} \to \mathbb{R}$ eine Funktion. Dann ist „Groß-Theta von g“

$$\Theta(g) = \left\{ f\colon \mathbb{N} \to \mathbb{R} \;\middle|\; \begin{array}{c} \text{es gibt positive Konstanten } c_1,\ c_2 \text{ und } n_0, \\ \text{so dass für alle } n \geq n_0 \text{ gilt:} \\ c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \end{array} \right\}$$

die Klasse der Fkt., die **genau** so schnell wachsen wie g.

Beispiel. $f(n) = 2n^2 + 4n - 20$

Bewiesen: $f \notin \mathcal{O}(n)$, $f \in \mathcal{O}(n^2)$, $f \in \mathcal{O}(n^3)$

Entsprechend: $f \in \Omega(n)$, $f \in \Omega(n^2)$, $f \notin \Omega(n^3)$

Zusammen: $f \in \Theta(n^2)$

d.h. es gibt pos. Konst. c_1, c_2, n_0, so dass für alle $n \geq n_0$ gilt: $c_1 \cdot n^2 \leq f(n) \leq c_2 \cdot n^2$.

Das Klassifikationsschema – intuitiv

$f \in \mathcal{O}(n^2)$ **bedeutet**

Das Klassifikationsschema – intuitiv

$f \in \mathcal{O}(n^2)$ **bedeutet** f wächst **höchstens** quadratisch.

Das Klassifikationsschema – intuitiv

$f \in \mathcal{O}(n^2)$ **bedeutet** f wächst **höchstens** quadratisch.

$f \in \Omega(n^2)$

Das Klassifikationsschema – intuitiv

$f \in \mathcal{O}(n^2)$ **bedeutet** f wächst **höchstens** quadratisch.

$f \in \Omega(n^2)$ **mindestens**

Das Klassifikationsschema – intuitiv

$f \in \mathcal{O}(n^2)$ **bedeutet** f wächst **höchstens** quadratisch.

$f \in \Omega(n^2)$ **mindestens**

$f \in \Theta(n^2)$

Das Klassifikationsschema – intuitiv

$f \in \mathcal{O}(n^2)$ **bedeutet** f wächst **höchstens** quadratisch.

$f \in \Omega(n^2)$ **mindestens**

$f \in \Theta(n^2)$ **genau**

Das Klassifikationsschema – intuitiv

$f \in \mathcal{O}(n^2)$ **bedeutet** f wächst **höchstens** quadratisch.

$f \in \Omega(n^2)$ **mindestens**

$f \in \Theta(n^2)$ **genau**

$f \in o(n^2)$

neu!

Das Klassifikationsschema – intuitiv

$f \in \mathcal{O}(n^2)$ **bedeutet** f wächst **höchstens** quadratisch.

$f \in \Omega(n^2)$ **mindestens**

$f \in \Theta(n^2)$ **genau**

$f \in o(n^2)$ **echt langsamer als**

neu!

Das Klassifikationsschema – intuitiv

$f \in \mathcal{O}(n^2)$	**bedeutet** f wächst	**höchstens**	quadratisch.
$f \in \Omega(n^2)$		**mindestens**	
$f \in \Theta(n^2)$		**genau**	
$f \in o(n^2)$	neu!	**echt langsamer als**	
$f \in \omega(n^2)$			

Das Klassifikationsschema – intuitiv

$f \in \mathcal{O}(n^2)$	**bedeutet**	f wächst	**höchstens**	quadratisch.
$f \in \Omega(n^2)$			**mindestens**	
$f \in \Theta(n^2)$			**genau**	
$f \in o(n^2)$	neu!		**echt langsamer als**	
$f \in \omega(n^2)$			**echt schneller als**	

Das Klassifikationsschema – intuitiv

$f \in \mathcal{O}(n^2)$ **bedeutet** f wächst **höchstens** quadratisch.

$f \in \Omega(n^2)$ **mindestens**

$f \in \Theta(n^2)$ **genau**

$f \in o(n^2)$ **echt langsamer als**

$f \in \omega(n^2)$ **echt schneller als**

neu!

Genaue Definition für „klein-oh“ und „klein-omega“ siehe Kapitel 3 [CLRS].

Das Klassifikationsschema – intuitiv

$f \in \mathcal{O}(n^2)$ **bedeutet** f wächst **höchstens** quadratisch.

$f \in \Omega(n^2)$ **mindestens**

$f \in \Theta(n^2)$ **genau**

$f \in o(n^2)$ **echt langsamer als**

$f \in \omega(n^2)$ **echt schneller als**

neu!

Genaue Definition für „klein-oh“ und „klein-omega“ siehe Kapitel 3 [CLRS].

Übung.

Gegeben folgende Funktionen $\mathbb{N} \to \mathbb{R}$ mit $n \mapsto \ldots$:

$$n^2, \quad \log_2 n, \quad \sqrt{n \log_2 n}, \quad 1.01^n, \quad n^{\log_3 4}, \quad \log_2(n \cdot 2^n), \quad 4^{\log_3 n}.$$

Das Klassifikationsschema – intuitiv

$f \in \mathcal{O}(n^2)$ **bedeutet** f wächst **höchstens** quadratisch.

$f \in \Omega(n^2)$ **mindestens**

$f \in \Theta(n^2)$ **genau**

$f \in o(n^2)$ **echt langsamer als**

$f \in \omega(n^2)$ **echt schneller als**

neu!

Genaue Definition für „klein-oh“ und „klein-omega“ siehe Kapitel 3 [CLRS].

Übung.

Gegeben folgende Funktionen $\mathbb{N} \to \mathbb{R}$ mit $n \mapsto \ldots$:

$$n^2,\ \log_2 n,\ \sqrt{n \log_2 n},\ 1.01^n,\ n^{\log_3 4},\ \log_2(n \cdot 2^n),\ 4^{\log_3 n}.$$

Sortieren Sie nach Geschwindigkeit des **asymptotischen Wachstums**, also so, dass danach gilt: $\mathcal{O}(\ldots) \subseteq \mathcal{O}(\ldots) \subseteq \cdots \subseteq \mathcal{O}(\ldots)$.

Das Klassifikationsschema – intuitiv

$f \in \mathcal{O}(n^2)$ **bedeutet** f wächst **höchstens** quadratisch.

$f \in \Omega(n^2)$ **mindestens**

$f \in \Theta(n^2)$ **genau**

$f \in o(n^2)$ **echt langsamer als**

$f \in \omega(n^2)$ **echt schneller als**

neu!

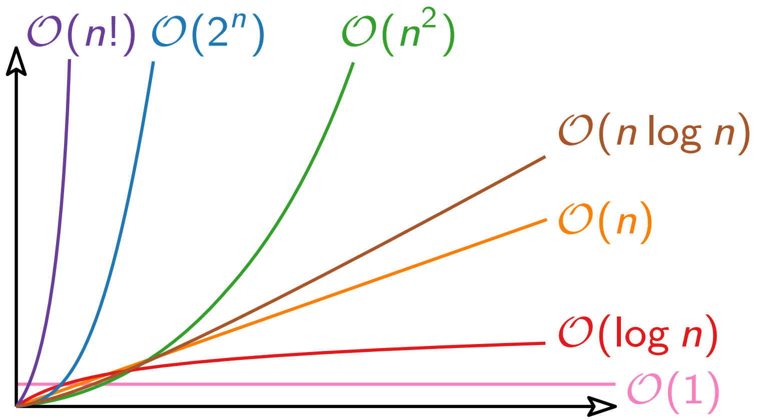
Genaue Definition für „klein-oh“ und „klein-omega“ siehe Kapitel 3 [CLRS].

Übung.

Gegeben folgende Funktionen $\mathbb{N} \to \mathbb{R}$ mit $n \mapsto \dots$:

$$n^2,\ \log_2 n,\ \sqrt{n \log_2 n},\ 1.01^n,\ n^{\log_3 4},\ \log_2(n \cdot 2^n),\ 4^{\log_3 n}.$$

Sortieren Sie nach Geschwindigkeit des **asymptotischen Wachstums**, also so, dass danach gilt: $\mathcal{O}(\dots) \subseteq \mathcal{O}(\dots) \subseteq \dots \subseteq \mathcal{O}(\dots)$.



Vergleich Laufzeiten

	Bester Fall	**Schlechtester Fall**
InsertionSort	$n-1$	$\frac{n(n-1)}{2}$
SelectionSort	$\frac{n(n-1)}{2}$	$\frac{n(n-1)}{2}$
BubbleSort	$n-1$	$\frac{n(n-1)}{2}$
BogoSort	$n-1$	∞
MergeSort	$n \log_2 n$	$n \log_2 n$

Geht das besser?

Ist das besser?

Vergleich Laufzeiten

	Bester Fall	Schlechtester Fall
InsertionSort	$\Theta(n)$	$\Theta(n^2)$
SelectionSort	$\Theta(n^2)$	$\Theta(n^2)$
BubbleSort	$\Theta(n)$	$\Theta(n^2)$
BogoSort	$\Theta(n)$	∞
MergeSort	$\Theta(n \log n)$	$\Theta(n \log n)$

Geht das besser?

Ist das besser?

Vergleich Laufzeiten

	Bester Fall	Schlechtester Fall
InsertionSort	$\Theta(n)$	$\Theta(n^2)$
SelectionSort	$\Theta(n^2)$	$\Theta(n^2)$
BubbleSort	$\Theta(n)$	$\Theta(n^2)$
BogoSort	$\Theta(n)$	∞
MergeSort	$\Theta(n \log n)$	$\Theta(n \log n)$

Geht das besser?

Ist das besser? Ja!