Datenstrukturen, Algorithmen und Programmierung 2 (DAP2)

Korrektheitsbeweise

Ein erstes nichttriviales Beispiel

Algorithmus Max-Search(Array A)

1. max $\leftarrow$ 1
2. **for** j $\leftarrow$ 2 **to** length[A] **do**
3.   **if** A[j] > A[max] **then** max $\leftarrow$ j
4. **return** max

Definition (Schleifeninvariante)

Eine Schleifeninvariante ist eine i.a. von der Anzahl i der Schleifendurchläufe abhängige Aussage A(i), die zu Beginn des i-ten Schleifendurchlauf gilt. Mit A(1) beziehen wir uns also auf den Zustand zu Beginn des ersten Durchlaufs. Dieser wird auch als Initialisierung bezeichnet.

Korrektheitsbeweise

Ein erstes nichttriviales Beispiel

Algorithmus Max-Search(Array A)

1. max $\leftarrow$ 1
2. **for** j $\leftarrow$ 2 **to** length[A] **do**
3.   **if** A[j] > A[max] **then** max $\leftarrow$ j
4. **return** max

Lemma 1

Die **for**-Schleife in Algorithmus Max-Search erfüllt folgende Schleifeninvariante:

(Inv.) A[max] ist ein größtes Element aus A[1..j-1].

Korrektheitsbeweise

Satz 2

Algorithmus Max-Search berechnet den Index eines größten Element aus einem Feld A.

Beweis

Der Schleifenaustritt aus der for-Schleife (Zeile 2) erfolgt für j=length[A]+1. Nach Lemma 1 gilt die Invariante insbesondere beim Schleifenaustritt und somit, dass A[max] ein größtes Element aus A[1..length[A]] ist. Der **return**-Befehl gibt mit max daher den Index eines größten Elementes aus A zurück.

Korrektheitsbeweise

Notation: Invarianten

Algorithmus Max-Search(Array A)

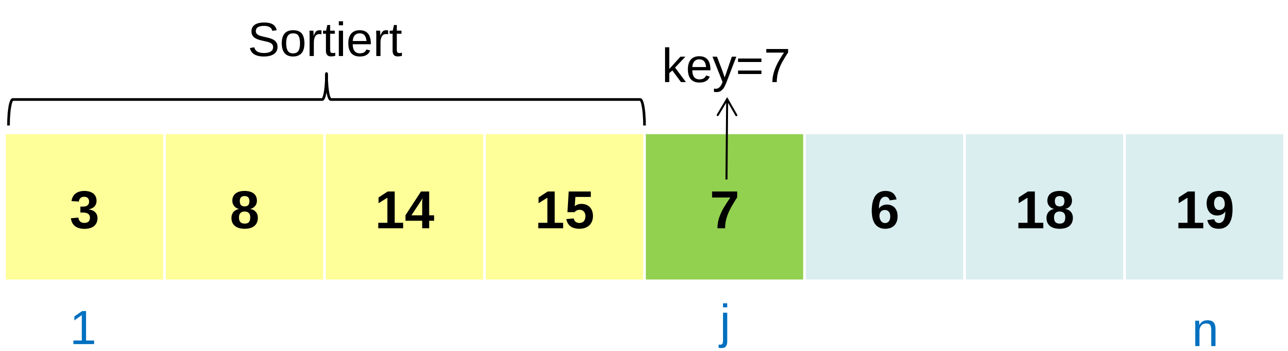
1. max $\leftarrow$ 1
2. **for** j $\leftarrow$ 2 **to** length[A] **do**
3. **if** A[j] > A[max] **then** max $\leftarrow$ j
4. **return** max

- Kommentare (Invariante):
- Initialisierung: max=1, j=2, A[max] ist
- Maximum von A[1..1].
- Invariante: A[max] ist Maximum von A[1..j-1]
- Austritt: A[max] ist Maximum von
- A[1..length[A]]

Insertion Sort

```
InsertionSort(Array A)
for j ← 2 to length[A] 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
```

- Eingabegröße n
- length[A] = n
- verschiebe alle Elemente aus
- A[1…j-1], die größer als key
- sind eine Stelle nach rechts
- Speichere key in Lücke



Invarianten InsertionSort

```
InsertionSort(Array A)
for j ← 2 to length[A] 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
```

- Initialisierung: j=2, A[1..1] ist sortiert
- Invariante: A[1..j-1] ist sortiert
- Austritt: A[1..length[A]] ist sortiert

Invarianten InsertionSort

InsertionSort(Array A)

1. **for** j ← 2 **to** length[A] **do**
2. key ← A[j]
3. i ← j-1
4. **while** i>0 and A[i]>key **do**
5. A[i+1] ← A[i]
6. i ← i-1
7. A[i+1] ← key

- Initialisierung: j=2, A[1..1] ist sortiert
- Invariante: A[1..j-1] ist sortiert
  - Initialisierung: i=j-1, A[1..j] \ A[j] ist
  - sortiert
  - Invariante: A[1..j] \A[i+1] ist sortiert
  - Austritt: A[1..j] \A[i+1] ist sortiert und
  - A[i] ≤ key < A[i+2] (wenn i+1=j,dann
  - gilt die letzte Ungl. nicht unbedingt)
  - oder i=0 und key < A[2]
- Austritt: A[1..length[A]] ist sortiert

Korrektheitsbeweise - Rekursionen

Algorithmus Sum(A,n)

1. **If** n=1 **then** return A[1]
2. **else**
3. W= Sum(A,n-1)
4. **return** A[n] + W

Problem

Wir können nicht genau sagen, wie häufig eine Rekursion ausgeführt wird.

Korrektheitsbeweise - Rekursionen

Algorithmus Sum(A,n)

1. **If** n=1 **then** return A[1]
2. **else**
3. W= Sum(A,n-1)
4. **return** A[n] + W

Abhilfe

Rekursion ist das Gegenstück zu Induktion. Man kann daher die Korrektheit leicht per Induktion zeigen.

Korrektheitsbeweise - Rekursionen

Algorithmus Sum(A,n)
1. **If** n=1 **then** return A[1]
2. **else**
3. W= Sum(A,n-1)
4. **return** A[n] + W

Beweis (Induktion über n):

(I.A.) Ist n=1, so gibt der Algorithmus in Zeile 1 den Wert A[1] zurück. Dies ist korrekt.

Satz 3

Algorithmus Sum(A,n) berechnet die Summe der ersten n Einträge eines Feldes A.

Korrektheitsbeweise - Rekursionen

Algorithmus Sum(A,n)

1. **If** n=1 **then** return A[1]
2. **else**
3. W= Sum(A,n-1)
4. **return** A[n] + W

Beweis (Induktion über n):

(I.A.) Ist n=1, so gibt der Algorithmus in Zeile 1 den Wert A[1] zurück. Dies ist korrekt.

(I.V.) Für n-1>0 berechnet Sum(A,n-1) die Summe der ersten n-1 Einträge von A.

Satz 3

Algorithmus Sum(A,n) berechnet die Summe der ersten n Einträge eines Feldes A.

Korrektheitsbeweise - Rekursionen

Hier: Induktionsschritt von n-1 nach n. Dies ist leichter, weil es den Beweis der Rekursion im Algorithmus anpasst.

Algorithmus Sum(A,n)

1. **If** n=1 **then** return A[1]
2. **else**
3.   W= Sum(A,n-1)
4.   **return** A[n] + W

<u>Beweis (Induktion[illegible]</u>

(I.A.) Ist n=1, so gibt [illegible]gorithmus in Zeile 1 den Wert A[1] zurück. Dies ist korrekt.

(I.V.) Für n-1>0 berechnet Sum(A,n-1) die Summe der ersten n-1 Einträge von A.

Satz 3

Algorithmus Sum(A,n) berechnet die Summe der ersten n Einträge eines Feldes A.

Korrektheitsbeweise - Rekursionen

Algorithmus Sum(A,n)

1. **If** n=1 **then** return A[1]
2. **else**
3. W= Sum(A,n-1)
4. **return** A[n] + W

Beweis (Induktion über n):

(I.A.) Ist n=1, so gibt der Algorithmus in Zeile 1 den Wert A[1] zurück. Dies ist korrekt.

(I.V.) Für n-1>0 berechnet Sum(A,n-1) die Summe der ersten n-1 Einträge von A.

(I.S.) Wir betrachten den Aufruf von Sum(A,n).

Satz 3

Algorithmus Sum(A,n) berechnet die Summe der ersten n Einträge eines Feldes A.

Korrektheitsbeweise - Rekursionen

Algorithmus Sum(A,n)

1. **If** n=1 **then** return A[1]
2. **else**
3. W= Sum(A,n-1)
4. **return** A[n] + W

Beweis (Induktion über n):
(I.A.) Ist n=1, so gibt der Algorithmus in Zeile 1 den Wert A[1] zurück. Dies ist korrekt.
(I.V.) Für n-1>0 berechnet Sum(A,n-1) die Summe der ersten n-1 Einträge von A.
(I.S.) Wir betrachten den Aufruf von Sum(A,n). Da n>1 ist, wird der else-Fall der ersten if-Anweisung aufgerufen. Dort wird W auf Sum(A,n-1) gesetzt.

Satz 3

Algorithmus Sum(A,n) berechnet die Summe der ersten n Einträge eines Feldes A.

Korrektheitsbeweise - Rekursionen

Algorithmus Sum(A,n)

1. **If** n=1 **then** return A[1]
2. **else**
3. W= Sum(A,n-1)
4. **return** A[n] + W

Beweis (Induktion über n):
(I.A.) Ist n=1, so gibt der Algorithmus in Zeile 1 den Wert A[1] zurück. Dies ist korrekt.
(I.V.) Für n-1>0 berechnet Sum(A,n-1) die Summe der ersten n-1 Einträge von A.
(I.S.) Wir betrachten den Aufruf von Sum(A,n). Da n>1 ist, wird der else-Fall der ersten if-Anweisung aufgerufen. Dort wird W auf Sum(A,n-1) gesetzt. Nach I.V. ist dies die Summe der ersten n-1 Einträge von A.

Satz 3

Algorithmus Sum(A,n) berechnet die Summe der ersten n Einträge eines Feldes A.

Korrektheitsbeweise - Rekursionen

Algorithmus Sum(A,n)

1. **If** n=1 **then** return A[1]
2. **else**
3. W= Sum(A,n-1)
4. **return** A[n] + W

Beweis (Induktion über n):
(I.A.) Ist n=1, so gibt der Algorithmus in Zeile 1 den Wert A[1] zurück. Dies ist korrekt.
(I.V.) Für n-1>0 berechnet Sum(A,n-1) die Summe der ersten n-1 Einträge von A.
(I.S.) Wir betrachten den Aufruf von Sum(A,n). Da n>1 ist, wird der else-Fall der ersten if-Anweisung aufgerufen. Dort wird W auf Sum(A,n-1) gesetzt. Nach I.V. ist dies die Summe der ersten n-1 Einträge von A. Nun wird in Zeile 4 A[n]+W, also die Summe der ersten n Einträge von A zurückgegeben.

Satz 3

Algorithmus Sum(A,n) berechnet die Summe der ersten n Einträge eines Feldes A.

Korrektheitsbeweise - Rekursionen

Algorithmus Sum(A,n)

1. **If** n=1 **then** return A[1]
2. **else**
3. W= Sum(A,n-1)
4. **return** A[n] + W

<u>Beweis (Induktion über n):</u>

(I.A.) Ist n=1, so gibt der Algorithmus in Zeile 1 den Wert A[1] zurück. Dies ist korrekt.

(I.V.) Für n-1>0 berechnet Sum(A,n-1) die Summe der ersten n-1 Einträge von A.

(I.S.) Wir betrachten den Aufruf von Sum(A,n). Da n>1 ist, wird der else-Fall der ersten if-Anweisung aufgerufen. Dort wird W auf Sum(A,n-1) gesetzt. Nach I.V. ist dies die Summe der ersten n-1 Einträge von A. Nun wird in Zeile 4 A[n]+W, also die Summe der ersten n Einträge von A zurückgegeben.

Satz 3

Algorithmus Sum(A,n) berechnet die Summe der ersten n Einträge eines Feldes A.

Zusammenfassung - Korrektheitsbeweise

- Grundannahme der Korrektheitsbeweise ist die korrekte Ausführung der Pseudocode Befehle
- Keine Schleifen: „Schrittweises Nachvollziehen des Programms“
- Schleifen: Korrektheit mittels Invarianten und Induktion
- Rekursion: Korrektheit mittels Induktion

Teile & Herrsche

Teile & Herrsche (Divide & Conquer)

- Teile Eingabe in mehrere Teile auf
- Löse das Problem rekursiv auf den Teilen
- Füge die Teillösungen zu einer Gesamtlösung zusammen

Teile & Herrsche

Teile & Herrsche (Divide & Conquer)

- Teile Eingabe in mehrere Teile auf
- Löse das Problem rekursiv auf den Teilen
- Füge die Teillösungen zu einer Gesamtlösung zusammen

Beispiel(Sortieren)

15	**7**	**6**	**13**	**25**	**4**	**9**	**12**

Teile & Herrsche

Teile & Herrsche (Divide & Conquer)

- Teile Eingabe in mehrere Teile auf
- Löse das Problem rekursiv auf den Teilen
- Füge die Teillösungen zu einer Gesamtlösung zusammen

Beispiel(Sortieren)

15	**7**	**6**	**13**	**25**	**4**	**9**	**12**

15	**7**	**6**	**13**

25	**4**	**9**	**12**

Schritt 1:
Aufteilen der
Eingabe

Teile & Herrsche

Teile & Herrsche (Divide & Conquer)

- Teile Eingabe in mehrere Teile auf
- Löse das Problem rekursiv auf den Teilen
- Füge die Teillösungen zu einer Gesamtlösung zusammen

Beispiel(Sortieren)

15	**7**	**6**	**13**	**25**	**4**	**9**	**12**

Schritt 2:
Rekursiv Sortieren

6	**7**	**13**	**15**

4	**9**	**12**	**25**

Teile & Herrsche

Teile & Herrsche (Divide & Conquer)

- Teile Eingabe in mehrere Teile auf
- Löse das Problem rekursiv auf den Teilen
- Füge die Teillösungen zu einer Gesamtlösung zusammen

Beispiel(Sortieren)

Schritt 3:
Zusammenfügen

Teile & Herrsche

Teile & Herrsche (Divide & Conquer)

- Teile Eingabe in mehrere Teile auf
- Löse das Problem rekursiv auf den Teilen
- Füge die Teillösungen zu einer Gesamtlösung zusammen

Beispiel(Sortieren)

Schritt 3:
Zusammenfügen

Teile & Herrsche

Teile & Herrsche (Divide & Conquer)

- Teile Eingabe in mehrere Teile auf
- Löse das Problem rekursiv auf den Teilen
- Füge die Teillösungen zu einer Gesamtlösung zusammen

Beispiel(Sortieren)

Schritt 3: Zusammenfügen

Teile & Herrsche

Teile & Herrsche (Divide & Conquer)

- Teile Eingabe in mehrere Teile auf
- Löse das Problem rekursiv auf den Teilen
- Füge die Teillösungen zu einer Gesamtlösung zusammen

Beispiel(Sortieren)

Schritt 3:
Zusammenfügen

4	**6**	**7**					

6	**7**	**13**	**15**

4	**9**	**12**	**25**

Teile & Herrsche

Teile & Herrsche (Divide & Conquer)

- Teile Eingabe in mehrere Teile auf
- Löse das Problem rekursiv auf den Teilen
- Füge die Teillösungen zu einer Gesamtlösung zusammen

Beispiel(Sortieren)

Schritt 3:
Zusammenfügen

4	**6**	**7**	**9**				

6	**7**	**13**	**15**

4	**9**	**12**	**25**

Teile & Herrsche

Teile & Herrsche (Divide & Conquer)

- Teile Eingabe in mehrere Teile auf
- Löse das Problem rekursiv auf den Teilen
- Füge die Teillösungen zu einer Gesamtlösung zusammen

Beispiel(Sortieren)

Schritt 3: Zusammenfügen

4	**6**	**7**	**9**	**12**			

6	**7**	**13**	**15**

4	**9**	**12**	**25**

Teile & Herrsche

Teile & Herrsche (Divide & Conquer)

- Teile Eingabe in mehrere Teile auf
- Löse das Problem rekursiv auf den Teilen
- Füge die Teillösungen zu einer Gesamtlösung zusammen

Beispiel(Sortieren)

Schritt 3:
Zusammenfügen

4	**6**	**7**	**9**	**12**	**13**		

6	**7**	**13**	**15**

4	**9**	**12**	**25**

Teile & Herrsche

Teile & Herrsche (Divide & Conquer)

- Teile Eingabe in mehrere Teile auf
- Löse das Problem rekursiv auf den Teilen
- Füge die Teillösungen zu einer Gesamtlösung zusammen

Beispiel(Sortieren)

Schritt 3: Zusammenfügen

4	**6**	**7**	**9**	**12**	**13**	**15**	

6	**7**	**13**	**15**

4	**9**	**12**	**25**

Teile & Herrsche

Teile & Herrsche (Divide & Conquer)

- Teile Eingabe in mehrere Teile auf
- Löse das Problem rekursiv auf den Teilen
- Füge die Teillösungen zu einer Gesamtlösung zusammen

Beispiel(Sortieren)

Schritt 3:
Zusammenfügen

4	**6**	**7**	**9**	**12**	**13**	**15**	**25**

6	**7**	**13**	**15**

4	**9**	**12**	**25**

Teile & Herrsche

Teile & Herrsche (Divide & Conquer)

- Teile Eingabe in mehrere Teile auf
- Löse das Problem rekursiv auf den Teilen
- Füge die Teillösungen zu einer Gesamtlösung zusammen

- *Wichtig*
- Wir benötigen Rekursionabbruch
- Sortieren: Folgen der Länge 1 sind sortiert

Teile & Herrsche

MergeSort(Array A, p, r)

1. **if** $p< r$ **then**
2. $q \leftarrow \lfloor (p+r)/2 \rfloor$
3. MergeSort(A,p,q)
4. MergeSort(A,q+1,r)
5. Merge(A,p,q,r)

Teile & Herrsche

MergeSort(Array A, p, r)

1. **if** p< r **then**
2. q ← ⌊(p+r)/2⌋
3. MergeSort(A,p,q)
4. MergeSort(A,q+1,r)
5. Merge(A,p,q,r)

- Sortiere A[p,…,r]

Teile & Herrsche

MergeSort(Array A, p, r)

1. **if** p< r **then**
2. q ← $\lfloor (p+r)/2 \rfloor$
3. MergeSort(A,p,q)
4. MergeSort(A,q+1,r)
5. Merge(A,p,q,r)

- Sortiere A[p,…,r]

Teile & Herrsche

MergeSort(Array A, p, r)

1. $\boxed{\textbf{if } p< r \textbf{ then}}$
2. q ← ⌊(p+r)/2⌋
3. MergeSort(A,p,q)
4. MergeSort(A,q+1,r)
5. Merge(A,p,q,r)

- Sortiere A[p,…,r]
- p≥r, dann nichts zu tun

Teile & Herrsche

MergeSort(Array A, p, r)

1. **if** p< r **then**
2. $\boxed{q \leftarrow \lfloor (p+r)/2 \rfloor}$
3. MergeSort(A,p,q)
4. MergeSort(A,q+1,r)
5. Merge(A,p,q,r)

- Sortiere A[p,…,r]
- p≥r, dann nichts zu tun

Teile & Herrsche

MergeSort(Array A, p, r)
1. **if** p< r **then**
2. $\boxed{q \leftarrow \lfloor (p+r)/2 \rfloor}$
3. MergeSort(A,p,q)
4. MergeSort(A,q+1,r)
5. Merge(A,p,q,r)

- Sortiere A[p,…,r]
- p≥r, dann nichts zu tun
- Berechne Mitte

Teile & Herrsche

MergeSort(Array A, p, r)

1. **if** p< r **then**
2. q ← ⌊(p+r)/2⌋
3. MergeSort(A,p,q)
4. MergeSort(A,q+1,r)
5. Merge(A,p,q,r)

- Sortiere A[p,…,r]
- p≥r, dann nichts zu tun
- Berechne Mitte

Teile & Herrsche

MergeSort(Array A, p, r)
1. **if** p< r **then**
2. q ← ⌊(p+r)/2⌋
3. MergeSort(A,p,q)
4. MergeSort(A,q+1,r)
5. Merge(A,p,q,r)

- Sortiere A[p,…,r]
- p≥r, dann nichts zu tun
- Berechne Mitte
- Sortiere linke Hälfte

Teile & Herrsche

MergeSort(Array A, p, r)
1. **if** p< r **then**
2. q ← ⌊(p+r)/2⌋
3. MergeSort(A,p,q)
4. $\boxed{\text{MergeSort(A,q+1,r)}}$
5. Merge(A,p,q,r)

- Sortiere A[p,…,r]
- p≥r, dann nichts zu tun
- Berechne Mitte
- Sortiere linke Hälfte

Teile & Herrsche

MergeSort(Array A, p, r)
1. **if** p< r **then**
2. q ← ⌊(p+r)/2⌋
3. MergeSort(A,p,q)
4. MergeSort(A,q+1,r)
5. Merge(A,p,q,r)

- Sortiere A[p,…,r]
- p≥r, dann nichts zu tun
- Berechne Mitte
- Sortiere linke Hälfte
- Sortiere rechte Hälfte

Teile & Herrsche

MergeSort(Array A, p, r)

1. **if** p< r **then**
2. q ← ⌊(p+r)/2⌋
3. MergeSort(A,p,q)
4. MergeSort(A,q+1,r)
5. Merge(A,p,q,r)

- Sortiere A[p,…,r]
- p≥r, dann nichts zu tun
- Berechne Mitte
- Sortiere linke Hälfte
- Sortiere rechte Hälfte

Teile & Herrsche

MergeSort(Array A, p, r)
1. **if** p< r **then**
2. q ← ⌊(p+r)/2⌋
3. MergeSort(A,p,q)
4. MergeSort(A,q+1,r)
5. Merge(A,p,q,r)

- Sortiere A[p,…,r]
- p≥r, dann nichts zu tun
- Berechne Mitte
- Sortiere linke Hälfte
- Sortiere rechte Hälfte
- Zusammenfügen

Teile & Herrsche

MergeSort(Array A, p, r)

1. **if** p< r **then**
2. q ← ⌊(p+r)/2⌋
3. MergeSort(A,p,q)
4. MergeSort(A,q+1,r)
5. Merge(A,p,q,r)

- ➢ Sortiere A[p,…,r]
- ➢ p≥r, dann nichts zu tun
- ➢ Berechne Mitte
- ➢ Sortiere linke Hälfte
- ➢ Sortiere rechte Hälfte
- ➢ Zusammenfügen

Teile & Herrsche

MergeSort(Array A, p, r) ➢ Sortiere A[p,…,r]

1. **if** p< r **then** ➢ p≥r, dann nichts zu tun
2. q ← ⌊(p+r)/2⌋ ➢ Berechne Mitte
3. MergeSort(A,p,q) ➢ Sortiere linke Hälfte
4. MergeSort(A,q+1,r) ➢ Sortiere rechte Hälfte
5. Merge(A,p,q,r) ➢ Zusammenfügen

Aufruf des Algorithmus

- MergeSort(A,1,r) für r=length[A]

Teile & Herrsche

Erweiterte Induktion

- (I.A.) Aussage A(1) ist richtig
- (I.V.) Aussage A(m) gilt für alle $1 \leq m \leq n$
- (I.S.) Aus (I.V.) folgt Aussage A(n+1)

- Bisher hatten wir nur A(n) benutzt, um A(n+1) zu folgern. Nun nutzen wir alle A(m) mit $1 \leq m \leq n$ (oder eine Teilmenge)

Teile & Herrsche

Satz 4

- Algorithmus MergeSort(A,p,r) sortiert das Feld A[p..r] korrekt.

Teile & Herrsche

Satz 4

- Algorithmus MergeSort(A,p,r) sortiert das Feld A[p..r] korrekt.

Beweis

- Wir zeigen die Korrektheit per Induktion über n=r-p.

Teile & Herrsche

Satz 4

- Algorithmus MergeSort(A,p,r) sortiert das Feld A[p..r] korrekt.

Beweis

- Wir zeigen die Korrektheit per Induktion über n=r-p.
- (I.A.) Für n=0, d.h. p=r, macht der Algorithmus nichts. Das Feld A[p..r] enthält nur ein Element und ist somit sortiert.

Teile & Herrsche

Satz 4

- Algorithmus MergeSort(A,p,r) sortiert das Feld A[p..r] korrekt.

Beweis

- Wir zeigen die Korrektheit per Induktion über n=r-p.
- (I.A.) Für n=0, d.h. p=r, macht der Algorithmus nichts. Das Feld A[p..r] enthält nur ein Element und ist somit sortiert.
- (I.V.) Für alle r,p mit m=r-p und $0 \leq m \leq n$ sortiert MergeSort(A,p,r) das Feld A[p..r] korrekt.

Teile & Herrsche

Satz 4

- Algorithmus MergeSort(A,p,r) sortiert das Feld A[p..r] korrekt.

Beweis

- (I.V.) Für alle r,p mit $m=r-p$ und $0\leq m\leq n$ sortiert MergeSort(A,p,r) das Feld A[p..r] korrekt.

Teile & Herrsche

Satz 4

- Algorithmus MergeSort(A,p,r) sortiert das Feld A[p..r] korrekt.

Beweis

- (I.V.) Für alle r,p mit $m=r-p$ und $0 \le m \le n$ sortiert MergeSort(A,p,r) das Feld A[p..r] korrekt.
- (I.S.) Wir betrachten den Aufruf von MergeSort für beliebige p, r mit $n+1 = r-p$. Da $n+1>0$ folgt $p<r$ und der Algorithmus führt den **then**-Fall aus. Hier wird q auf $\lfloor (p+r)/2 \rfloor$ gesetzt.

Teile & Herrsche

Satz 4

- Algorithmus MergeSort(A,p,r) sortiert das Feld A[p..r] korrekt.

Beweis

- (I.V.) Für alle r,p mit $m=r-p$ und $0\leq m\leq n$ sortiert MergeSort(A,p,r) das Feld A[p..r] korrekt.
- (I.S.) Wir betrachten den Aufruf von MergeSort für beliebige p, r mit $n+1 = r-p$. Da $n+1>0$ folgt $p<r$ und der Algorithmus führt den **then**-Fall aus. Hier wird q auf $\lfloor (p+r)/2 \rfloor$ gesetzt. Es gilt $q\geq p$ und $q<r$.

Teile & Herrsche

Satz 4

- Algorithmus MergeSort(A,p,r) sortiert das Feld A[p..r] korrekt.

Beweis

- (I.V.) Für alle r,p mit m=r-p und 0≤m≤n sortiert MergeSort(A,p,r) das Feld A[p..r] korrekt.
- (I.S.) Wir betrachten den Aufruf von MergeSort für beliebige p, r mit n+1 = r-p. Da n+1>0 folgt p<r und der Algorithmus führt den **then**-Fall aus. Hier wird q auf $\lfloor(p+r)/2\rfloor$ gesetzt. Es gilt q≥p und q<r. Dann wird MergeSort rekursiv in den Grenzen p,q bzw. q+1,r aufgerufen.

Teile & Herrsche

Satz 4

- Algorithmus MergeSort(A,p,r) sortiert das Feld A[p..r] korrekt.

Beweis

- (I.V.) Für alle r,p mit $m=r-p$ und $0\leq m\leq n$ sortiert MergeSort(A,p,r) das Feld A[p..r] korrekt.
- (I.S.) Wir betrachten den Aufruf von MergeSort für beliebige p, r mit $n+1 = r-p$. Da $n+1>0$ folgt $p<r$ und der Algorithmus führt den **then**-Fall aus. Hier wird q auf $\lfloor (p+r)/2 \rfloor$ gesetzt. Es gilt $q\geq p$ und $q<r$. Dann wird MergeSort rekursiv in den Grenzen p,q bzw. q+1,r aufgerufen. Nach (I.V.) Sortiert MergeSort in diesem Fall korrekt.

Teile & Herrsche

Satz 4

- Algorithmus MergeSort(A,p,r) sortiert das Feld A[p..r] korrekt.

Beweis

- (I.V.) Für alle r,p mit $m=r-p$ und $0\leq m\leq n$ sortiert MergeSort(A,p,r) das Feld A[p..r] korrekt.
- (I.S.) Wir betrachten den Aufruf von MergeSort für beliebige p, r mit $n+1 = r-p$. Da $n+1>0$ folgt $p<r$ und der Algorithmus führt den **then**-Fall aus. Hier wird q auf $\lfloor(p+r)/2\rfloor$ gesetzt. Es gilt $q\geq p$ und $q<r$. Dann wird MergeSort rekursiv in den Grenzen p,q bzw. q+1,r aufgerufen. Nach (I.V.) Sortiert MergeSort in diesem Fall korrekt. Nun folgt die Korrektheit aus der Tatsache, dass Merge die beiden Bereiche korrekt zu einem sortierten Feld zusammenfügt.

Teile & Herrsche

MergeSort(Array A, p, r) ➢ Sortiere A[p,…,r]

1. **if** p< r **then** ➢ p≥r, dann nichts zu tun
2. q ← ⌊(p+r)/2⌋ ➢ Berechne Mitte
3. MergeSort(A,p,q) ➢ Sortiere linke Hälfte
4. MergeSort(A,q+1,r) ➢ Sortiere rechte Hälfte
5. Merge(A,p,q,r) ➢ Zusammenfügen

Aufruf des Algorithmus

- MergeSort(A,1,n) für Feld A[1...n]
- Laufzeit?

Teile & Herrsche

MergeSort(Array A, p, r) ➢ Sortiere A[p,…,r]

1. **if** p< r **then** ➢ p≥r, dann nichts zu tun
2. $q \leftarrow \lfloor (p+r)/2 \rfloor$ ➢ Berechne Mitte
3. MergeSort(A,p,q) ➢ Sortiere linke Hälfte
4. MergeSort(A,q+1,r) ➢ Sortiere rechte Hälfte
5. Merge(A,p,q,r) ➢ Zusammenfügen

Aufruf des Algorithmus

- MergeSort(A,1,n) für Feld A[1...n]
- T(m) = maximale Laufzeit bei Eingabe A, p, r mit r-p+1=m

Teile & Herrsche

MergeSort(Array A, p, r) — Laufzeit:

1. $\boxed{\textbf{if } p< r \textbf{ then}}$ — 1
2. q ← ⌊(p+r)/2⌋
3. MergeSort(A,p,q)
4. MergeSort(A,q+1,r)
5. Merge(A,p,q,r)

Aufruf des Algorithmus

- MergeSort(A,1,n) für Feld A[1...n]
- T(m) = maximale Laufzeit bei Eingabe A, p, r mit r-p+1=m

Teile & Herrsche

MergeSort(Array A, p, r)

1. **if** p< r **then**
2. $\boxed{q \leftarrow \lfloor (p+r)/2 \rfloor}$
3. MergeSort(A,p,q)
4. MergeSort(A,q+1,r)
5. Merge(A,p,q,r)

Laufzeit:

1

1

Aufruf des Algorithmus

- MergeSort(A,1,n) für Feld A[1...n]
- T(m) = maximale Laufzeit bei Eingabe A, p, r mit r-p+1=m

Teile & Herrsche

MergeSort(Array A, p, r)

1. **if** p< r **then**
2. q ← ⌊(p+r)/2⌋
3. MergeSort(A,p,q)
4. MergeSort(A,q+1,r)
5. Merge(A,p,q,r)

Laufzeit:

1

1

1+T(n/2)

Wir nehmen an, dass n eine Zweierpotenz ist, d.h. wir müssen uns nicht um das Runden kümmern.

Aufruf des Algorithmus

- MergeSort(A,1,n) für Feld A[1...n]
- T(m) = maximale Laufzeit bei Eingabe A, p, r mit r-p+1=m

Teile & Herrsche

MergeSort(Array A, p, r)	Laufzeit:
1. **if** p< r **then**	1
2. $q \leftarrow \lfloor (p+r)/2 \rfloor$	1
3. MergeSort(A,p,q)	1+T(n/2)
4. $\boxed{\text{MergeSort(A,q+1,r)}}$	1+T(n/2)
5. Merge(A,p,q,r)	

Wir nehmen an, dass n eine Zweierpotenz ist, d.h. wir müssen uns nicht um das Runden kümmern.

Aufruf des Algorithmus

- MergeSort(A,1,n) für Feld A[1...n]
- T(m) = maximale Laufzeit bei Eingabe A, p, r mit r-p+1=m

Teile & Herrsche

MergeSort(Array A, p, r)	Laufzeit:
1. **if** p< r **then**	1
2. q ← $\lfloor (p+r)/2 \rfloor$	1
3. MergeSort(A,p,q)	1+T(n/2)
4. MergeSort(A,q+1,r)	1+T(n/2)
5. Merge(A,p,q,r)	≤ c‘n

c‘ ist genügend große Konstante

Aufruf des Algorithmus

- MergeSort(A,1,n) für Feld A[1...n]
- T(m) = maximale Laufzeit bei Eingabe A, p, r mit r-p+1=m

Teile & Herrsche

MergeSort(Array A, p, r)	Laufzeit:
1. **if** p< r **then**	1
2. $q \leftarrow \lfloor (p+r)/2 \rfloor$	1
3. MergeSort(A,p,q)	1+T(n/2)
4. MergeSort(A,q+1,r)	1+T(n/2)
5. Merge(A,p,q,r)	$\le c'n$
	$\le 2T(n/2) + cn$

$c \ge c'+4$

Aufruf des Algorithmus

- MergeSort(A,1,n) für Feld A[1...n]
- T(m) = maximale Laufzeit bei Eingabe A, p, r mit r-p+1=m

Teile & Herrsche

Laufzeit als Rekursion

- $$T(n) \leq \begin{cases} C & \text{, falls } n=1 \\ 2\,T(n/2) + cn & \text{, falls } n>1 \end{cases}$$
- Wobei c,C geeignete Konstanten sind.

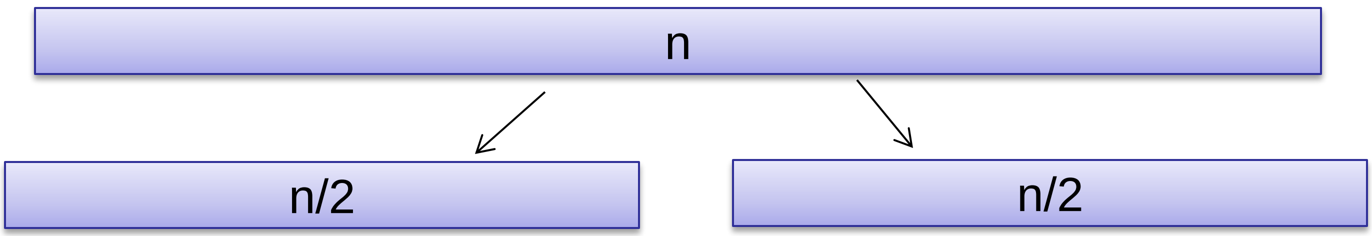
Teile & Herrsche

Auflösen von $T(n) \leq 2\, T(n/2) + cn$ (Intuition)

n	cn

Teile & Herrsche

Auflösen von $T(n) \leq 2\ T(n/2) + cn$ (Intuition)

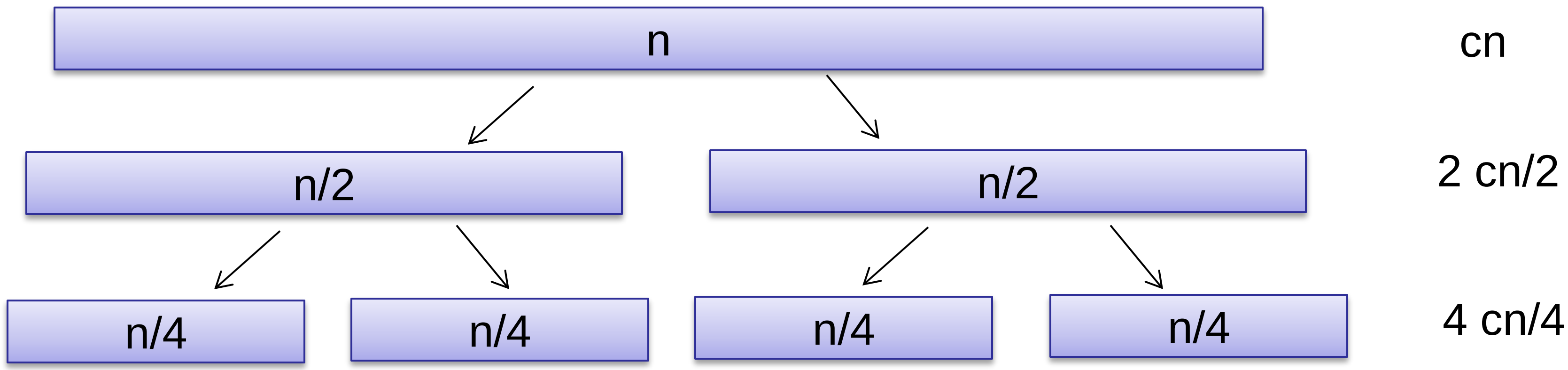


cn

2 cn/2

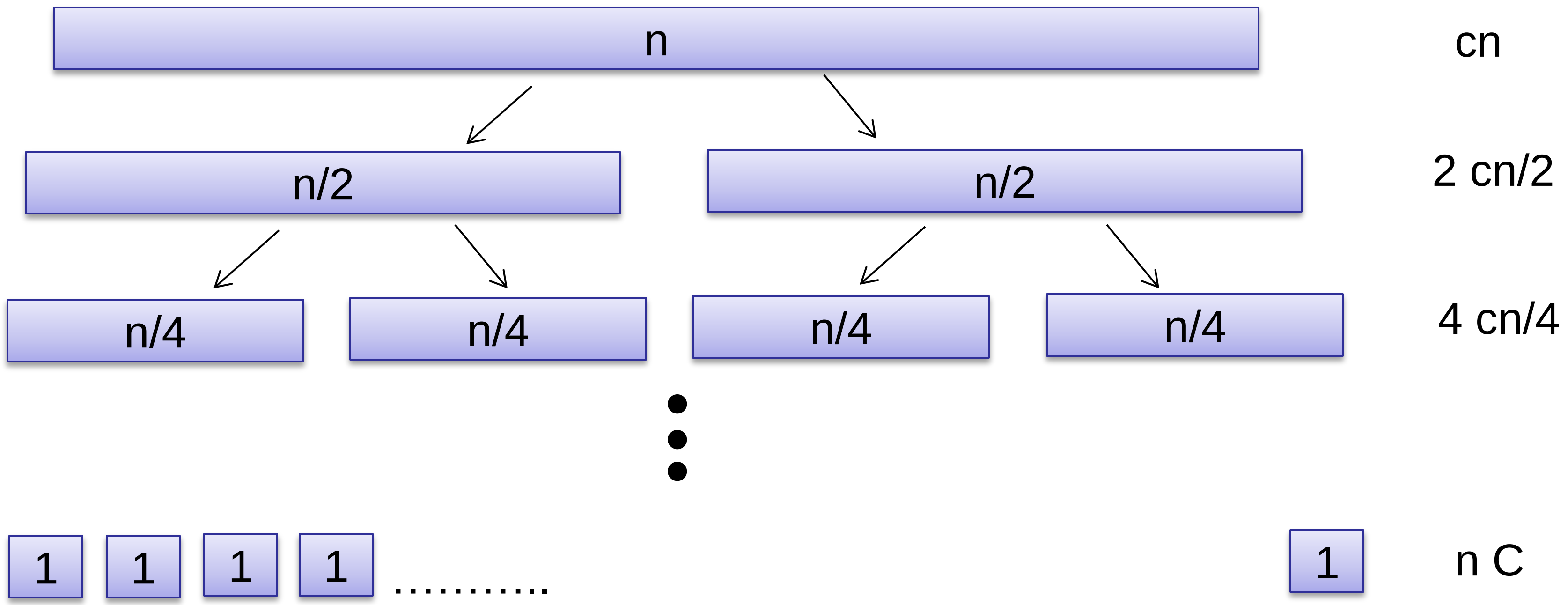
Teile & Herrsche

Auflösen von $T(n) \leq 2\ T(n/2) + cn$ (Intuition)



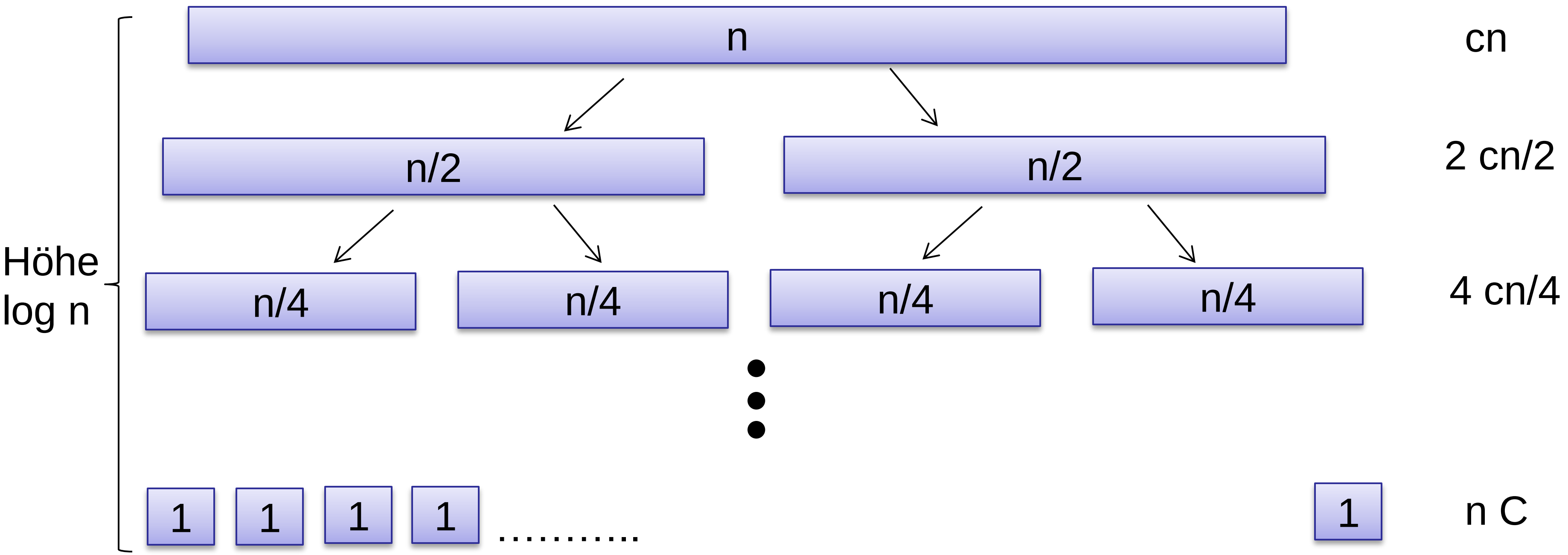
Teile & Herrsche

Auflösen von $T(n) \leq 2\ T(n/2) + cn$ (Intuition)



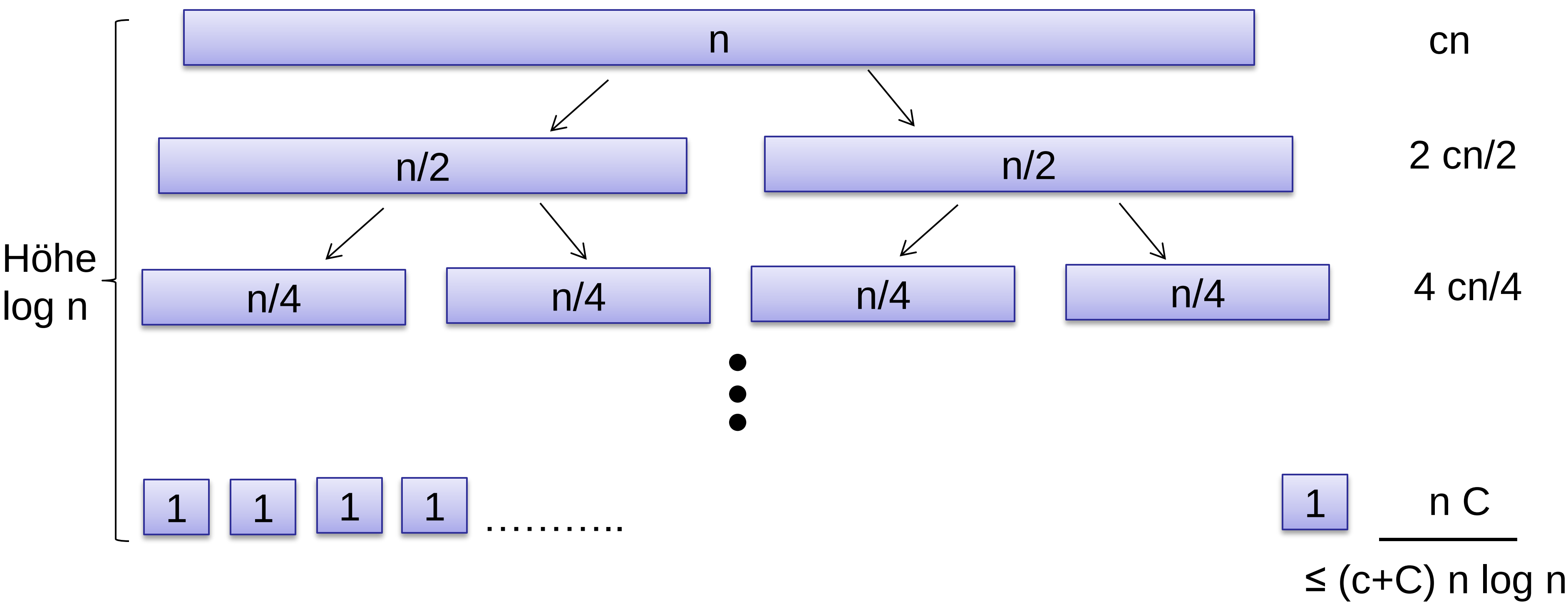
Teile & Herrsche

Auflösen von $T(n) \leq 2\ T(n/2) + cn$ (Intuition)



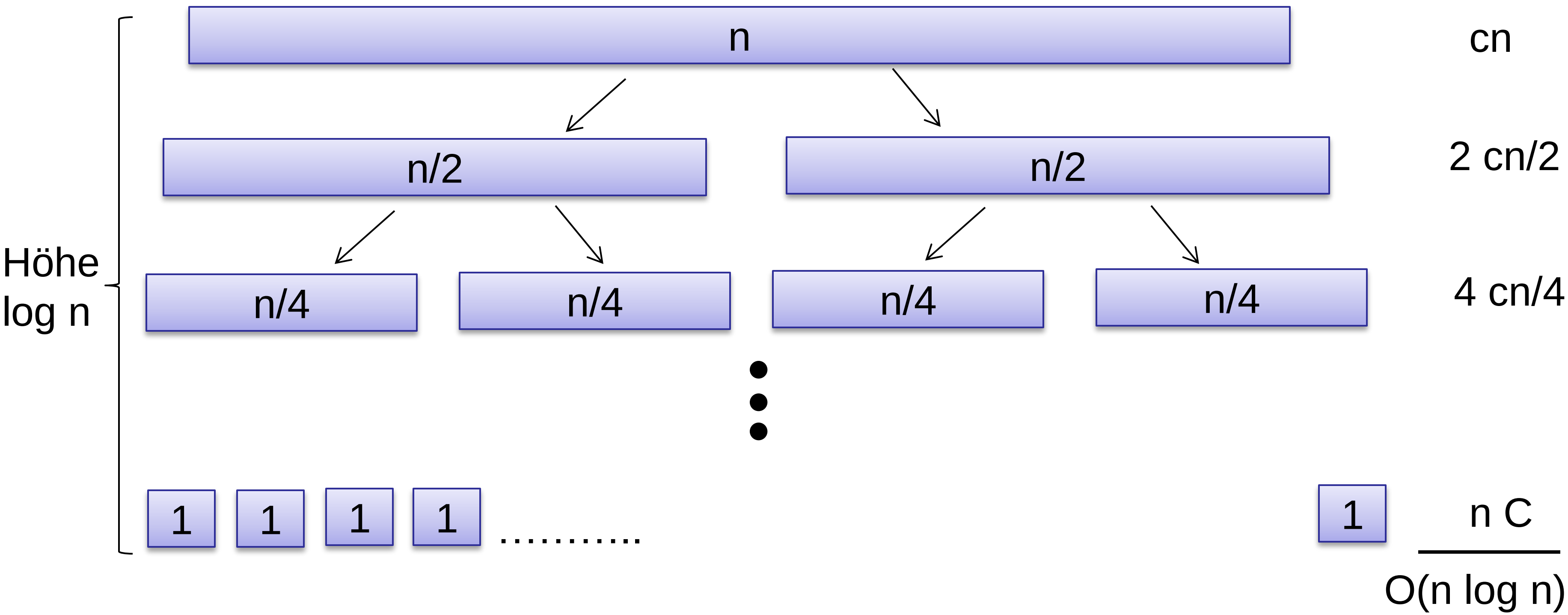
Teile & Herrsche

Auflösen von $T(n) \leq 2\, T(n/2) + cn$ (Intuition)



Teile & Herrsche

Auflösen von $T(n) \le 2\ T(n/2) + cn$ (Intuition)



Teile & Herrsche

Satz 5

- Algorithmus MergeSort hat eine Laufzeit von O(n log n).

Beweis

Teile & Herrsche

Satz 5

- Algorithmus MergeSort hat eine Laufzeit von O(n log n).

Beweis

- Die Laufzeit für T(1) und T(2) ist konstant.

Teile & Herrsche

Satz 5

- Algorithmus MergeSort hat eine Laufzeit von O(n log n).

Beweis

- Die Laufzeit für T(1) und T(2) ist konstant.
- Sei also T(2) ≤ C‘ und C* ≥ max{c,C‘}. Wir zeigen per Induktion, T(n) ≤ C*n log n für alle n≥2

Teile & Herrsche

Satz 5

- Algorithmus MergeSort hat eine Laufzeit von $O(n \log n)$.

Beweis

- Die Laufzeit für $T(1)$ und $T(2)$ ist konstant.
- Sei also $T(2) \leq C'$ und $C^* \geq \max\{c, C'\}$. Wir zeigen per Induktion, $T(n) \leq C^* n \log n$ für alle $n \geq 2$
- (I.A.) für $n=2$ gilt $T(2) \leq C' \leq C^*\, 2 \log 2$.

Teile & Herrsche

Satz 5

- Algorithmus MergeSort hat eine Laufzeit von $O(n \log n)$.

Beweis

- Die Laufzeit für $T(1)$ und $T(2)$ ist konstant.
- Sei also $T(2) \le C'$ und $C^* \ge \max\{c, C'\}$. Wir zeigen per Induktion, $T(n) \le C^* n \log n$ für alle $n \ge 2$
- (I.A.) für $n=2$ gilt $T(2) \le C' \le C^*\, 2 \log 2$.
- (I.V.) Für Eingabelänge $m<n$ ist die Laufzeit $T(m) \le C^*\, m \log m$.

Teile & Herrsche

Satz 5

- Algorithmus MergeSort hat eine Laufzeit von $O(n \log n)$.

Beweis

- Die Laufzeit für $T(1)$ und $T(2)$ ist konstant.
- Sei also $T(2) \le C'$ und $C^* \ge \max\{c, C'\}$. Wir zeigen per Induktion, $T(n) \le C^* n \log n$ für alle $n \ge 2$
- (I.A.) für $n=2$ gilt $T(2) \le C' \le C^* \, 2 \log 2$.
- (I.V.) Für Eingabelänge $m<n$ ist die Laufzeit $T(m) \le C^* \, m \log m$.
- (I.S.) Es gilt $T(n) \le 2\, T(n/2) + cn$.

Teile & Herrsche

Satz 5

- Algorithmus MergeSort hat eine Laufzeit von $O(n \log n)$.

Beweis

- Die Laufzeit für $T(1)$ und $T(2)$ ist konstant.
- Sei also $T(2) \le C'$ und $C^* \ge \max\{c,C'\}$. Wir zeigen per Induktion, $T(n) \le C^* n \log n$ für alle $n \ge 2$
- (I.A.) für $n=2$ gilt $T(2) \le C' \le C^*\, 2 \log 2$.
- (I.V.) Für Eingabelänge $m<n$ ist die Laufzeit $T(m) \le C^*\, m \log m$.
- (I.S.) Es gilt $T(n) \le 2\, T(n/2) + cn$. Nach (I.V.) gilt
  $T(n) \le 2\, C^*\, n/2 \log(n/2) + cn$

Teile & Herrsche

Satz 5

- Algorithmus MergeSort hat eine Laufzeit von $O(n \log n)$.

Beweis

- Die Laufzeit für $T(1)$ und $T(2)$ ist konstant.
- Sei also $T(2) \leq C'$ und $C^* \geq \max\{c, C'\}$. Wir zeigen per Induktion, $T(n) \leq C^* n \log n$ für alle $n \geq 2$
- (I.A.) für $n=2$ gilt $T(2) \leq C' \leq C^*\, 2 \log 2$.
- (I.V.) Für Eingabelänge $m<n$ ist die Laufzeit $T(m) \leq C^*\, m \log m$.
- (I.S.) Es gilt $T(n) \leq 2\, T(n/2) + cn$. Nach (I.V.) gilt

$$T(n) \leq 2\, C^*\, n/2 \log(n/2) + cn$$

$$\leq C^*\, n\, (\log(n)-1) + cn$$

Teile & Herrsche

Satz 5

- Algorithmus MergeSort hat eine Laufzeit von O(n log n).

Beweis

- Die Laufzeit für T(1) und T(2) ist konstant.
- Sei also T(2) ≤ C‘ und C* ≥ max{c,C‘}. Wir zeigen per Induktion, T(n) ≤ C*n log n für alle n≥2
- (I.A.) für n=2 gilt T(2) ≤ C‘ ≤ C* 2 log 2.
- (I.V.) Für Eingabelänge m<n ist die Laufzeit T(m) ≤ C* m log m.
- (I.S.) Es gilt T(n) ≤ 2 T(n/2) + cn. Nach (I.V.) gilt

$$\begin{aligned} T(n) &\le 2\, C^*\, n/2 \log(n/2) + cn \\ &\le C^*\, n\, (\log(n)-1) + cn \\ &\le C^*\, n\, (\log(n)-1) + C^*n = C^*\, n \log(n) \end{aligned}$$

Teile & Herrsche

Satz 5

- Algorithmus MergeSort hat eine Laufzeit von $O(n \log n)$.

Beweis

- Die Laufzeit für $T(1)$ und $T(2)$ ist konstant.
- Sei also $T(2) \leq C'$ und $C^* \geq \max\{c, C'\}$. Wir zeigen per Induktion, $T(n) \leq C^* n \log n$ für alle $n \geq 2$
- (I.A.) für $n=2$ gilt $T(2) \leq C' \leq C^*\, 2 \log 2$.
- (I.V.) Für Eingabelänge $m<n$ ist die Laufzeit $T(m) \leq C^*\, m \log m$.
- (I.S.) Es gilt $T(n) \leq 2\, T(n/2) + cn$. Nach (I.V.) gilt

$$\begin{aligned} T(n) &\leq 2\, C^*\, n/2 \log(n/2) + cn \\ &\leq C^*\, n\, (\log(n)-1) + cn \\ &\leq C^*\, n\, (\log(n)-1) + C^* n = C^*\, n \log(n) \end{aligned}$$

- Also gilt $T(n) = O(n \log n)$, [da für $n \geq n_0 = 2$, $T(n) \leq C^* n \log n$ ist]

Teile & Herrsche

Satz 5

- Algorithmus MergeSort hat eine Laufzeit von O(n log n).

Beweis

- Die Laufzeit für T(1) und T(2) ist konstant.
- Sei also T(2) ≤ C‘ und C* ≥ max{c,C‘}. Wir zeigen per Induktion, T(n) ≤ C*n log n für alle n≥2
- (I.A.) für n=2 gilt T(2) ≤ C‘ ≤ C* 2 log 2.
- (I.V.) Für Eingabelänge m<n ist die Laufzeit T(m) ≤ C* m log m.
- (I.S.) Es gilt T(n) ≤ 2 T(n/2) + cn. Nach (I.V.) gilt

  T(n) ≤ 2 C* n/2 log(n/2) + cn
  ≤ C* n (log(n)-1) + cn
  ≤ C* n (log(n)-1) + C*n = C* n log(n)
- Also gilt T(n) = O(n log n), [da für n≥n_0=2, T(n) ≤ C*n log n ist]

Teile & Herrsche

Wodurch unterscheiden sich Teile & Herrsche Algorithmen?

- Die Anzahl der Teilprobleme
- Die Größe der Teilprobleme
- Den Algorithmus für das Zusammensetzen der Teilprobleme
- Den Rekursionsabbruch