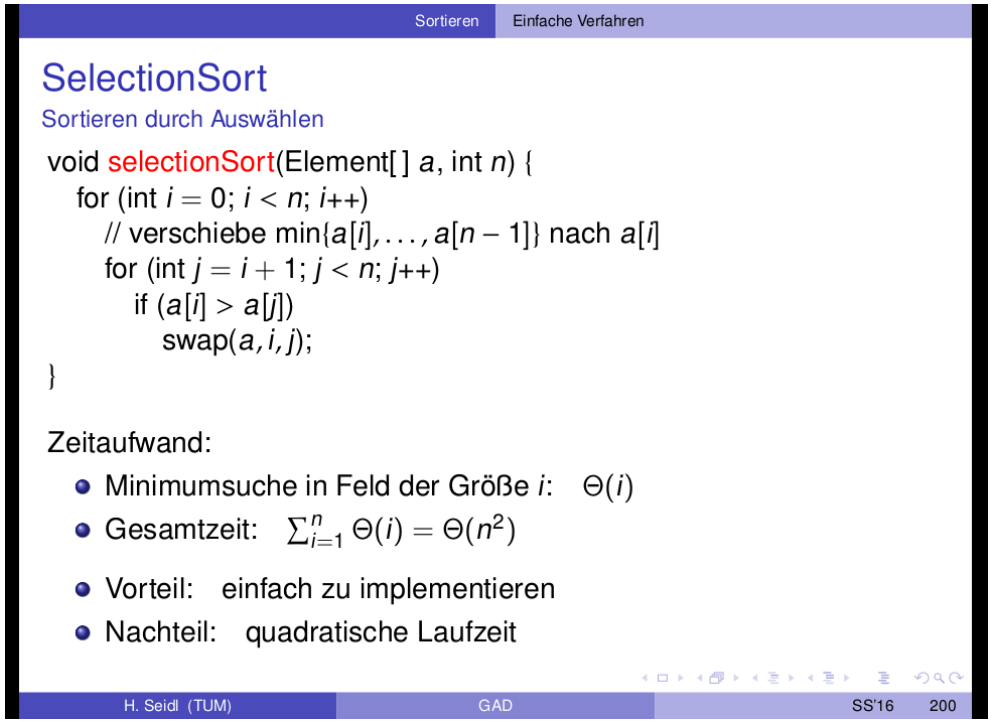
Script **generated by TTT**

Title: Seidl: GAD (11.05.2016)

Date: Wed May 11 13:24:02 CEST 2016

Duration: 46:40 min

Pages: 29



SelectionSort

Sortieren durch Auswählen

```
void selectionSort(Element[ ] a, int n) {
  for (int i = 0; i < n; i++)
    // verschiebe min{a[i], ..., a[n − 1]} nach a[i]
    for (int j = i + 1; j < n; j++)
      if (a[i] > a[j])
        swap(a, i, j);
}
```

Zeitaufwand:

- Minimumsuche in Feld der Größe i: $\Theta(i)$
- Gesamtzeit: $\sum_{i=1}^{n} \Theta(i) = \Theta(n^2)$
- Vorteil: einfach zu implementieren
- Nachteil: quadratische Laufzeit

InsertionSort

Sortieren durch Einfügen

Nimm ein Element aus der Eingabesequenz und füge es an der richtigen Stelle in die Ausgabesequenz ein

Beispiel

5	10	19	1	14	3
5	10	19	1	14	3
5	**10**	19	1	14	3
5	10	**19**	1	14	3
5	10	1	19	14	3

5	1	10	19	14	3
1	5	10	19	14	3
1	5	10	**14**	19	3
1	...	←	...	3	19
1	**3**	5	10	14	19

InsertionSort

Sortieren durch Einfügen

```
void insertionSort(Element[ ] a, int n) {
  for (int i = 1; i < n; i++)
    // verschiebe a_i an die richtige Stelle
    for (int j = i − 1; j ≥ 0; j−−)
      if (a[j] > a[j + 1])
        swap(a, j, j + 1);
}
```

Zeitaufwand:

- Einfügung des i-ten Elements an richtiger Stelle: $O(i)$
- Gesamtzeit: $\sum_{i=1}^{n} O(i) = O(n^2)$
- Vorteil: einfach zu implementieren
- Nachteil: quadratische Laufzeit

Einfache Verfahren

SelectionSort

- mit besserer Minimumstrategie worst case Laufzeit $O(n \log n)$ erreichbar
  (mehr dazu in einer späteren Vorlesung)

InsertionSort

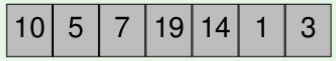
- mit besserer Einfügestrategie worst case Laufzeit $O(n \log^2 n)$ erreichbar
  ($\rightarrow$ ShellSort)

MergeSort

Sortieren durch Verschmelzen

Zerlege die Eingabesequenz rekursiv in Teile, die separat sortiert und dann zur Ausgabesequenz verschmolzen werden

Beispiel



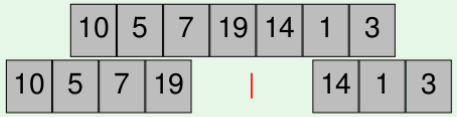
Sortieren MergeSort

MergeSort

Sortieren durch Verschmelzen

Zerlege die Eingabesequenz rekursiv in Teile, die separat sortiert und dann zur Ausgabesequenz verschmolzen werden

Beispiel



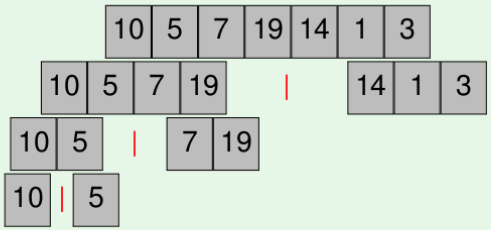
Sortieren MergeSort

MergeSort

Sortieren durch Verschmelzen

Zerlege die Eingabesequenz rekursiv in Teile, die separat sortiert und dann zur Ausgabesequenz verschmolzen werden

Beispiel



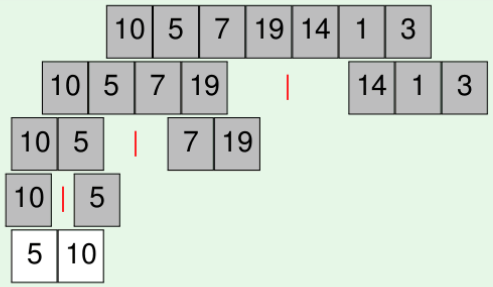
Sortieren MergeSort

MergeSort

Sortieren durch Verschmelzen

Zerlege die Eingabesequenz rekursiv in Teile, die separat sortiert und dann zur Ausgabesequenz verschmolzen werden

Beispiel



MergeSort

Sortieren durch Verschmelzen

```
void mergeSort(Element[] a, int l, int r) {
  if (l == r) return;            // nur ein Element
  m = ⌊(r + l)/2⌋;               // Mitte
  mergeSort(a, l, m);            // linken Teil sortieren
  mergeSort(a, m + 1, r);        // rechten Teil sortieren
  j = l;   k = m + 1;            // verschmelzen
  for (i = 0, i ≤ r − l, i++)
    if (j > m) { b[i] = a[k];   k++; }          // linker Teil leer
    else
      if (k > r) { b[i] = a[j];   j++; }        // rechter Teil leer
      else
        if (a[j] ≤ a[k]) { b[i] = a[j];   j++; }
        else { b[i] = a[k];   k++; }
  for (i = 0, i ≤ r − l, i++)    a[l + i] = b[i];          // zurückkopieren
}
```


MergeSort

Sortieren durch Verschmelzen

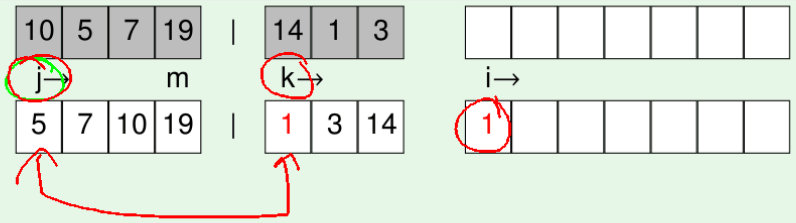
Beispiel (Verschmelzen)

10	5	7	19		14	1	3							

MergeSort

Sortieren durch Verschmelzen

Beispiel (Verschmelzen)



MergeSort

Sortieren durch Verschmelzen

Beispiel (Verschmelzen)

10	5	7	19		14	1	3							
j→			m		k→			i→						
5	7	10	19		1	3	14	1						
5	7	10	19			3	14	1	3					

MergeSort

Sortieren durch Verschmelzen

Beispiel (Verschmelzen)

10	5	7	19	|	14	1	3							
j→			m		k→			i→						
5	7	10	19	|	1	3	14	1						
5	7	10	19	|		3	14	1	3					
5	7	10	19	|			14	1	3	5				

MergeSort

Sortieren durch Verschmelzen

Beispiel (Verschmelzen)

10	5	7	19	|	14	1	3							
j→			m		k→			i→						
5	7	10	19	|	1	3	14	1						
5	7	10	19	|		3	14	1	3					
5	7	10	19	|			14	1	3	5				
	7	10	19	|			14	1	3	5	7			
		10	19	|			14	1	3	5	7	10		
			19	|			14	1	3	5	7	10	14	

MergeSort

Sortieren durch Verschmelzen

Beispiel (Verschmelzen)

10	5	7	19	|	14	1	3							
j→			m		k→			i→						
5	7	10	19	|	1	3	14	1						
5	7	10	19	|		3	14	1	3					
5	7	10	19	|			14	1	3	5				
	7	10	19	|			14	1	3	5	7			
		10	19	|			14	1	3	5	7	10		
			19	|			14	1	3	5	7	10	14	
			19	|				1	3	5	7	10	14	19

MergeSort

Sortieren durch Verschmelzen

Zeitaufwand:

- $T(n)$: Laufzeit bei Feldgröße n
- $T(1) = \Theta(1)$
  $T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n)$

$\Rightarrow$ $T(n) \in O(n \log n)$
(folgt aus dem sogenannten Master-Theorem)

Analyse rekursiver Funktionen

Divide-and-Conquer-Algorithmen

- nichtrekursive Unterprogrammaufrufe sind einfach zu analysieren (separate Analyse des Funktionsaufrufs und Einsetzen)

- rekursive Aufrufstrukturen liefern Rekursionsgleichungen

$\Rightarrow$ Funktionswert wird in Abhängigkeit von Funktionswerten für kleinere Argumente bestimmt
gesucht: nichtrekursive / geschlossene Form

Analyse rekursiver Funktionen

Divide-and-Conquer-Algorithmen

- nichtrekursive Unterprogrammaufrufe sind einfach zu analysieren (separate Analyse des Funktionsaufrufs und Einsetzen)

- rekursive Aufrufstrukturen liefern Rekursionsgleichungen

$\Rightarrow$ Funktionswert wird in Abhängigkeit von Funktionswerten für kleinere Argumente bestimmt
gesucht: nichtrekursive / geschlossene Form

Anwendung: Divide-and-Conquer-Algorithmen

- gegeben: Problem der Größe $n = b^k$ ($k \in \mathbb{N}_0$)
- falls $k \geq 1$:
  - zerlege das Problem in d Teilprobleme der Größe n/b
  - löse die Teilprobleme (d rekursive Aufrufe)
  - setze aus den Teillösungen die Lösung zusammen
- falls $k = 0$ bzw. $n = 1$: investiere Aufwand a zur Lösung

Analyse rekursiver Funktionen

Divide-and-Conquer-Algorithmen

- nichtrekursive Unterprogrammaufrufe sind einfach zu analysieren (separate Analyse des Funktionsaufrufs und Einsetzen)

- rekursive Aufrufstrukturen liefern Rekursionsgleichungen

$\Rightarrow$ Funktionswert wird in Abhängigkeit von Funktionswerten für kleinere Argumente bestimmt
gesucht: nichtrekursive / geschlossene Form

Anwendung: Divide-and-Conquer-Algorithmen

- gegeben: Problem der Größe $n = b^k$ ($k \in \mathbb{N}_0$)
- falls $k \geq 1$:
  - zerlege das Problem in d Teilprobleme der Größe n/b
  - löse die Teilprobleme (d rekursive Aufrufe)
  - setze aus den Teillösungen die Lösung zusammen
- falls $k = 0$ bzw. $n = 1$: investiere Aufwand a zur Lösung

a

Analyse rekursiver Funktionen

Divide-and-Conquer-Algorithmen

- nichtrekursive Unterprogrammaufrufe sind einfach zu analysieren (separate Analyse des Funktionsaufrufs und Einsetzen)

- rekursive Aufrufstrukturen liefern Rekursionsgleichungen

$\Rightarrow$ Funktionswert wird in Abhängigkeit von Funktionswerten für kleinere Argumente bestimmt
gesucht: nichtrekursive / geschlossene Form

$c \cdot n$

Anwendung: Divide-and-Conquer-Algorithmen

- gegeben: Problem der Größe $n = b^k$ ($k \in \mathbb{N}_0$)
- falls $k \geq 1$:
  - zerlege das Problem in d Teilprobleme der Größe n/b
  - löse die Teilprobleme (d rekursive Aufrufe)
  - setze aus den Teillösungen die Lösung zusammen
- falls $k = 0$ bzw. $n = 1$: investiere Aufwand a zur Lösung

Analyse rekursiver Funktionen

Divide-and-Conquer-Algorithmen

- Betrachte den Aufwand für jede Rekursionstiefe
- Anfang: Problemgröße n
- Level für Rekursionstiefe i: d^i Teilprobleme der Größe n/b^i

$\Rightarrow$ Gesamtaufwand auf Rekursionslevel i:

$$d^i c\frac{n}{b^i} = cn\left(\frac{d}{b}\right)^i \qquad \text{(geometrische Reihe)}$$

- $d < b$ Aufwand sinkt mit wachsender Rekursionstiefe; *erstes* Level entspricht konstantem Anteil des Gesamtaufwands
- $d = b$ Gesamtaufwand für jedes Level gleich groß; maximale Rekursionstiefe logarithmisch, Gesamtaufwand $\Theta(n \log n)$
- $d > b$ Aufwand steigt mit wachsender Rekursionstiefe; *letztes* Level entspricht konstantem Anteil des Gesamtaufwands

Analyse rekursiver Funktionen

Divide-and-Conquer-Algorithmen

Geometrische Folge: $(a_i)_{i\in\mathbb{N}}$

Verhältnis benachbarter Folgenglieder konstant: $q = a_{i+1}/a_i$

n-te Partialsumme der geometrischen Reihe:

$$s_n = \sum_{i=0}^{n} a_i = a_0 + \ldots + a_n = a_0 + a_0 q + a_0 q^2 + \ldots a_0 q^n$$

Wert:

$$s_n = a_0 \frac{q^{n+1} - 1}{q - 1} \quad \text{für } q \neq 1$$

bzw.

$$s_n = a_0(n+1) \quad \text{für } q = 1$$

Sortieren MergeSort

Analyse rekursiver Funktionen

Divide-and-Conquer-Algorithmen

- Betrachte den Aufwand für jede Rekursionstiefe
- Anfang: Problemgröße n
- Level für Rekursionstiefe i: d^i Teilprobleme der Größe n/b^i

$\Rightarrow$ Gesamtaufwand auf Rekursionslevel i:

$$d^i c\frac{n}{b^i} = cn\left(\frac{d}{b}\right)^i \qquad \text{(geometrische Reihe)}$$

- $d < b$ Aufwand sinkt mit wachsender Rekursionstiefe; *erstes* Level entspricht konstantem Anteil des Gesamtaufwands
- $d = b$ Gesamtaufwand für jedes Level gleich groß; maximale Rekursionstiefe logarithmisch, Gesamtaufwand $\Theta(n \log n)$
- $d > b$ Aufwand steigt mit wachsender Rekursionstiefe; *letztes* Level entspricht konstantem Anteil des Gesamtaufwands

Sortieren MergeSort

Analyse rekursiver Funktionen

Divide-and-Conquer-Algorithmen

Geometrische Folge: $(a_i)_{i\in\mathbb{N}}$

Verhältnis benachbarter Folgenglieder konstant: $q = a_{i+1}/a_i$

n-te Partialsumme der geometrischen Reihe:

$$s_n = \sum_{i=0}^{n} a_i = a_0 + \ldots + a_n = a_0 + a_0 q + a_0 q^2 + \ldots a_0 q^n$$

Wert:

$$s_n = a_0 \frac{q^{n+1} - 1}{q - 1} \quad \text{für } q \neq 1$$

bzw.

$$s_n = a_0(n+1) \quad \text{für } q = 1$$

Analyse rekursiver Funktionen

Divide-and-Conquer-Algorithmen

- Level 0: 1 Problem der Größe $n = b^k$
- Level i: d^i Probleme der Größe $n/b^i = b^{k-i}$
- Level k: d^k Probleme der Größe $n/b^k = b^{k-k} = 1$, hier jedes mit Kosten a, also Kosten ad^k

Sortieren MergeSort

Analyse rekursiver Funktionen

Divide-and-Conquer-Algorithmen

- Level 0: 1 Problem der Größe $n = b^k$
- Level i: d^i Probleme der Größe $n/b^i = b^{k-i}$
- Level k: d^k Probleme der Größe $n/b^k = b^{k-k} = 1$, hier jedes mit Kosten a, also Kosten ad^k

- $d = b$: Kosten $ad^k = ab^k = an \in \Theta(n)$ auf Level k, $cnk = cn \log_b n \in \Theta(n \log n)$ für den Rest

H. Seidl (TUM) GAD SS'16 211