Script **generated by TTT**

Title: TÃ¤ubig: GAD (28.05.2013)

Date: Tue May 28 14:17:08 CEST 2013

Duration: 91:41 min

Pages: 49

Hashfunktionen mit wenig Kollisionen

Lemma

Für mindestens die Hälfte der Funktionen $h \in H_m$ gilt:

$$C(h) \leq 2cn(n-1)/m$$

Beweis.

- Aus Lemma $\mathbb{E}[C(h)] \leq cn(n-1)/m$ und Markov-Ungleichung $\Pr[X \geq k \cdot \mathbb{E}[X]] \leq \frac{1}{k}$ folgt:

$$\Pr[C(h) \geq 2cn(n-1)/m] \leq \Pr\left[C(h) \geq 2\mathbb{E}[C(h)]\right] \leq \frac{1}{2}$$

$\Rightarrow$ Für höchstens die Hälfte der Funktionen ist $C(h) \geq \frac{2cn(n-1)}{m}$

$\Rightarrow$ Für mindestens die Hälfte der Funktionen ist $C(h) \leq \frac{2cn(n-1)}{m}$

□

Hashing | Perfektes Hashing

Hashfunktionen mit wenig Kollisionen

Lemma

Für mindestens die Hälfte der Funktionen $h \in H_m$ gilt:

$$C(h) \leq 2cn(n-1)/m$$

H. Täubig (TUM) GAD SS'13 214 / 647

Hashing | Perfektes Hashing

Hashfunktionen mit wenig Kollisionen

Lemma

Für mindestens die Hälfte der Funktionen $h \in H_m$ gilt:

$$C(h) \leq 2cn(n-1)/m$$

H. Täubig (TUM) GAD SS'13 214 / 647

Statisches Wörterbuch

Ziel: lineare Tabellengröße

Idee: zweistufige Abbildung der Schlüssel

- Wähle Hashfunktion h mit wenig Kollisionen ($\approx$ 2 Versuche)

$\Rightarrow$ 1. Stufe bildet Schlüssel auf Buckets von konstanter durchschnittlicher Größe ab

- Wähle Hashfunktionen h_ℓ ohne Kollisionen ($\approx$ 2 Versuche pro h_ℓ)

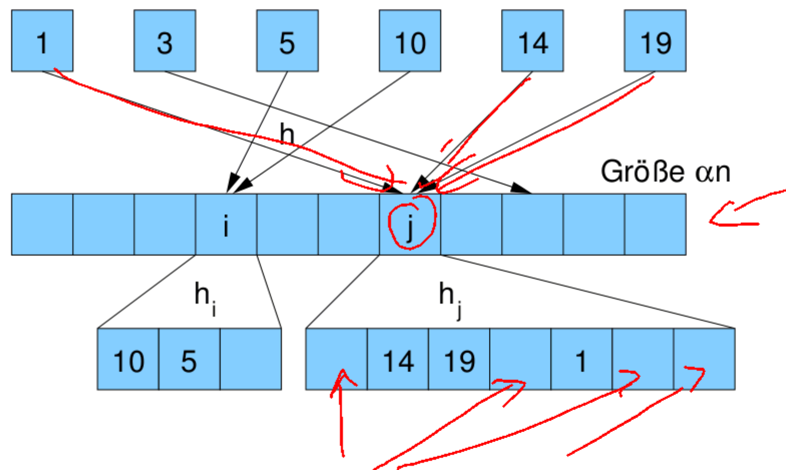
$\Rightarrow$ 2. Stufe benutzt quadratisch viel Platz für jedes Bucket, um alle Kollisionen aus der 1. Stufe aufzulösen

Statisches Wörterbuch

- B_ℓ^h: Menge der Elemente in S, die h auf ℓ abbildet, $\ell \in \{0, \ldots, m-1\}$ und $h \in H_m$
- b_ℓ^h: Kardinalität von B_ℓ^h, also $b_\ell^h := |B_\ell^h|$
- Für jedes ℓ führen die Schlüssel in B_ℓ^h zu $b_\ell^h(b_\ell^h - 1)$ Kollisionen
- Also ist die Gesamtzahl der Kollisionen

$$C(h) = \sum_{\ell=0}^{m-1} b_\ell^h(b_\ell^h - 1)$$

Perfektes statisches Hashing



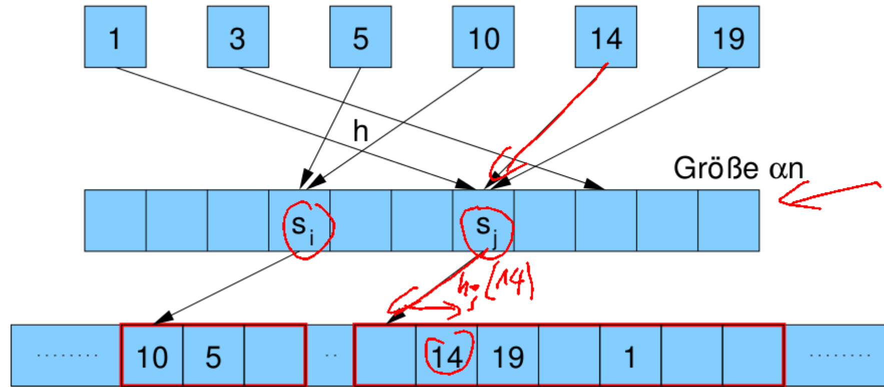
Perfektes statisches Hashing: 2. Stufe

- Für jedes B_ℓ:
  Berechne $m_\ell = cb_\ell(b_\ell - 1) + 1$.
  Wähle zufällig Funktion $h_\ell \in H_{m_\ell}$, bis h_ℓ die Menge B_ℓ injektiv in $\{0, \ldots, m_\ell - 1\}$ abbildet (also ohne Kollisionen).
  Mindestens die Hälfte der Funktionen in H_{m_ℓ} tut das.
- Hintereinanderreihung der einzelnen Tabellen ergibt eine Gesamtgröße der Tabelle von $\sum_\ell m_\ell$
- Teiltabelle für B_ℓ beginnt an Position $s_\ell = m_0 + m_1 + \ldots + m_{\ell-1}$ und endet an Position $s_\ell + m_\ell - 1$
- Für gegebenen Schlüssel x, berechnen die Anweisungen

$$\ell = h(x); \quad \text{return } s_\ell + h_\ell(x);$$

dann eine injektive Funktion auf der Menge S.

Perfektes statisches Hashing



Perfektes statisches Hashing

- Die Funktion ist beschränkt durch:

$$\begin{aligned}
\sum_{\ell=0}^{\lceil \alpha n \rceil - 1} m_\ell &= \sum_{\ell=0}^{\lceil \alpha n \rceil - 1} (c \cdot b_\ell(b_\ell - 1) + 1) \qquad \text{(siehe Def. der } m_\ell\text{'s)} \\
&\leq c \cdot C(h) + \lceil \alpha n \rceil \\
&\leq c \cdot 2cn/\alpha + \alpha n + 1 \\
&\leq (2c^2/\alpha + \alpha)n + 1
\end{aligned}$$

- Zur Minimierung der Schranke betrachte die Ableitung

$$\begin{aligned}
f(\alpha) &= (2c^2/\alpha + \alpha)n + 1 \\
f'(\alpha) &= (-2c^2/\alpha^2 + 1)n
\end{aligned}$$

$\Rightarrow$ $f'(\alpha) = 0$ liefert $\alpha = \sqrt{2}c$

$\Rightarrow$ Adressbereich: $0 \ldots 2\sqrt{2}cn$

Perfektes statisches Hashing

Satz

Für eine beliebige Menge von n Schlüsseln kann eine perfekte Hashfunktion mit Zielmenge $\{0, \ldots, 2\sqrt{2}cn\}$ in linearer erwarteter Laufzeit konstruiert werden.

- Da wir wissen, dass wir für beliebige m eine 2-universelle Familie von Hashfunktionen finden können, kann man also z.B. eine Hashfunktion mit Adressmenge $\{0, \ldots, 4\sqrt{2}n\}$ in linearer erwarteter Laufzeit konstruieren. (Las Vegas-Algorithmus)
- Unsere Minimierung hat nicht den Platz berücksichtigt, der benötigt wird, um die Werte s_ℓ, sowie die ausgewählten Hashfunktionen h_ℓ zu speichern.

$\Rightarrow$ Berechnung von α sollte eigentlich angepasst werden (entsprechend Speicherplatz pro Element, pro m_ℓ und pro h_ℓ)

Perfektes dynamisches Hashing

Kann man perfekte Hashfunktionen auch dynamisch konstruieren?

ja, z.B. mit Cuckoo Hashing

- 2 Hashfunktionen h_1 und h_2
- 2 Hashtabellen T_1 und T_2
- bei find und remove jeweils in beiden Tabellen nachschauen
- bei insert abwechselnd beide Tabellen betrachten, das zu speichernde Element an die Zielposition der aktuellen Tabelle schreiben und wenn dort schon ein anderes Element stand, dieses genauso in die andere Tabelle verschieben usw.
- evt. Anzahl Verschiebungen durch $2 \log n$ beschränken, um Endlosschleife zu verhindern (ggf. kompletter Rehash mit neuen Funktionen h_1, h_2)

Übersicht

5 Hashing
- Hashtabellen
- Hashing with Chaining
- Universelles Hashing
- Hashing with Linear Probing
- Anpassung der Tabellengröße
- Perfektes Hashing
- Diskussion / Alternativen

Probleme beim linearen Sondieren

- Offene Hashverfahren allgemein:

  Erweiterte Hashfunktion $h(k, i)$ gibt an, auf welche Adresse ein Schlüssel k abgebildet werden soll, wenn bereits i Versuche zu einer Kollision geführt haben

- Lineares Sondieren (Linear Probing):

$$h(k, i) = (h(k) + i) \bmod m$$

- Primäre Häufung (primary clustering): tritt auf, wenn für Schlüssel k_1, k_2 mit unterschiedlichen Hashwerten $h(k_1) \neq h(k_2)$ ab einem bestimmten Punkt i_1 bzw. i_2 die gleiche Sondierfolge auftritt:

$$\exists i_1, i_2 \quad \forall j : \quad h(k_1, i_1 + j) = h(k_2, i_2 + j)$$

Probleme beim quadratischen Sondieren

- Quadratisches Sondieren (Quadratic Probing):

$$h(k, i) = \left(h(k) + c_1 i + c_2 i^2\right) \bmod m \qquad (c_2 \neq 0)$$

oder:

$$h(k, i) = \left(h(k) - (-1)^i \lceil i/2 \rceil^2\right) \bmod m$$

Probleme beim quadratischen Sondieren

- Quadratisches Sondieren (Quadratic Probing):

$$h(k, i) = \left(h(k) + c_1 i + c_2 i^2\right) \bmod m \qquad (c_2 \neq 0)$$

oder:

$$h(k, i) = \left(h(k) - (-1)^i \lceil i/2 \rceil^2\right) \bmod m$$

- $h(k, i)$ soll möglichst surjektiv auf die Adressmenge $\{0, \ldots, m-1\}$ abbilden, um freie Positionen auch immer zu finden. Bei

$$h(k, i) = \left(h(k) - (-1)^i \lceil i/2 \rceil^2\right) \bmod m$$

  z.B. durch Wahl von $\quad m$ prim $\quad \wedge \quad m \equiv 3 \bmod 4$

Probleme beim quadratischen Sondieren

- Quadratisches Sondieren (Quadratic Probing):

$$h(k,i) = \left(h(k) + c_1 i + c_2 i^2\right) \bmod m \qquad (c_2 \neq 0)$$

oder: $$h(k,i) = \left(h(k) - (-1)^i \lceil i/2 \rceil^2\right) \bmod m$$

- $h(k,i)$ soll möglichst surjektiv auf die Adressmenge $\{0,\ldots,m-1\}$ abbilden, um freie Positionen auch immer zu finden. Bei

$$h(k,i) = \left(h(k) - (-1)^i \lceil i/2 \rceil^2\right) \bmod m$$

z.B. durch Wahl von m prim $\wedge$ $m \equiv 3 \bmod 4$

- Sekundäre Häufung (secondary clustering): tritt auf, wenn für Schlüssel k_1, k_2 mit gleichem Hashwert $h(k_1) = h(k_2)$ auch die nachfolgende Sondierfolge gleich ist:

$$\forall i: \quad h(k_1,i) = h(k_2,i)$$

Double Hashing

- Auflösung der Kollisionen der Hashfunktion h durch eine zweite Hashfunktion h':

$$h(k,i) = [h(k) + i \cdot h'(k)] \bmod m$$

wobei für alle k gelten soll, dass $h'(k)$ teilerfremd zu m ist,

z.B. $$h'(k) = 1 + k \bmod m - 1$$
oder $$h'(k) = 1 + k \bmod m - 2$$

für Primzahl m

- primäre und sekundäre Häufung werden weitgehend vermieden, aber nicht komplett ausgeschlossen

Übersicht

6 Sortieren
- Einfache Verfahren
- MergeSort
- Untere Schranke
- QuickSort
- Selektieren
- Schnelleres Sortieren
- Externes Sortieren

Statisches Wörterbuch

Lösungsmöglichkeiten:

- Perfektes Hashing
  - Vorteil: Suche in konstanter Zeit
  - Nachteil: keine Ordnung auf Elementen, d.h. Bereichsanfragen (z.B. alle Namen, die mit 'A' anfangen) teuer
- Speicherung der Daten in sortiertem Feld
  - Vorteil: Bereichsanfragen möglich
  - Nachteil: Suche teurer (logarithmische Zeit)

Sortierproblem

- gegeben: Ordnung $\leq$ auf der Menge möglicher Schlüssel
- Eingabe: Sequenz $s = \langle e_1, \ldots, e_n \rangle$

  Beispiel:

5	10	19	1	14	3

- Ausgabe: Permutation $s' = \langle e'_1, \ldots, e'_n \rangle$ von s, so dass $\text{key}(e'_i) \leq \text{key}(e'_{i+1})$ für alle $i \in \{1, \ldots, n\}$

  Beispiel:

1	3	5	10	14	19

Übersicht

6 Sortieren
- Einfache Verfahren
- MergeSort
- Untere Schranke
- QuickSort
- Selektieren
- Schnelleres Sortieren
- Externes Sortieren

SelectionSort

Sortieren durch Auswählen

```
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], a[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

insertionSort(Element[] a, int n) {
  for (int $i = 1$; $i < n$; i++)
    // verschiebe a_i an die richtige Stelle
    for (int $j = i - 1$; $j \geq 0$; $j - -$)
      if ($a[j] > a[j+1]$)
        swap($a[j]$, $a[j+1]$);
}

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
  (→ ShellSort)

Übersicht

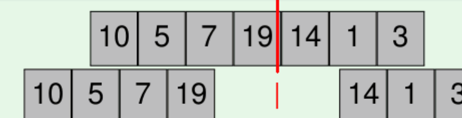
6 Sortieren
- Einfache Verfahren
- MergeSort
- Untere Schranke
- QuickSort
- Selektieren
- Schnelleres Sortieren
- Externes Sortieren

MergeSort

Sortieren durch Verschmelzen

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

Beispiel

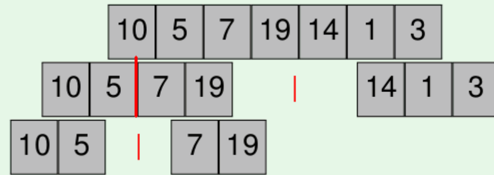


MergeSort

Sortieren durch Verschmelzen

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

Beispiel



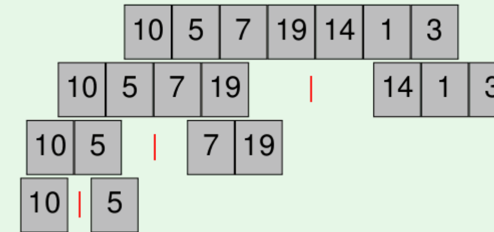
Sortieren MergeSort

MergeSort

Sortieren durch Verschmelzen

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

Beispiel



H. Täubig (TUM) GAD SS'13 239 / 647

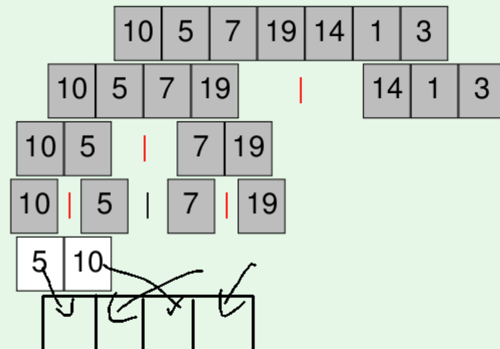
Sortieren MergeSort

MergeSort

Sortieren durch Verschmelzen

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

Beispiel



H. Täubig (TUM) GAD SS'13 239 / 647

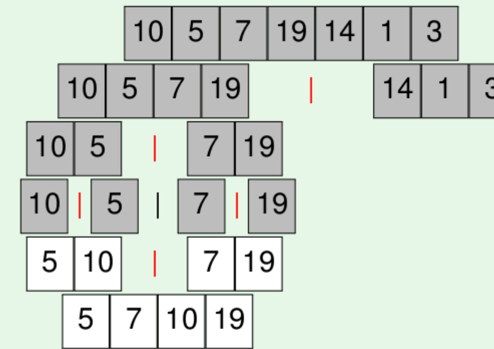
Sortieren MergeSort

MergeSort

Sortieren durch Verschmelzen

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

Beispiel



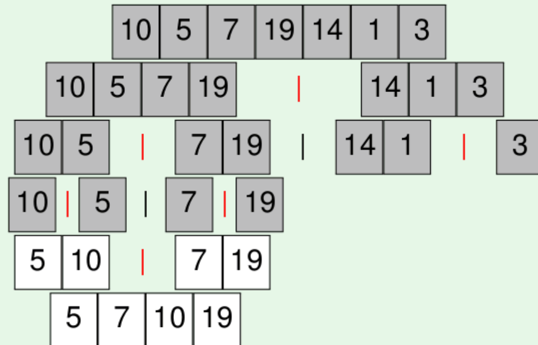
H. Täubig (TUM) GAD SS'13 239 / 647

MergeSort

Sortieren durch Verschmelzen

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

Beispiel

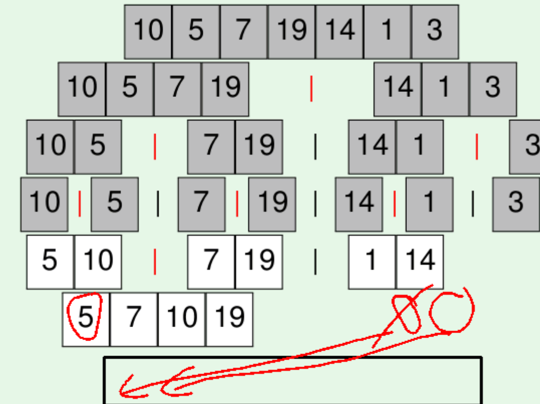


MergeSort

Sortieren durch Verschmelzen

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

Beispiel

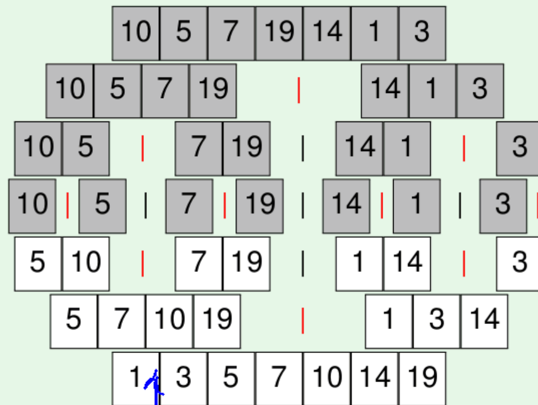


MergeSort

Sortieren durch Verschmelzen

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

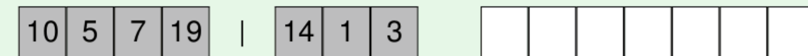
Beispiel



MergeSort

Sortieren durch Verschmelzen

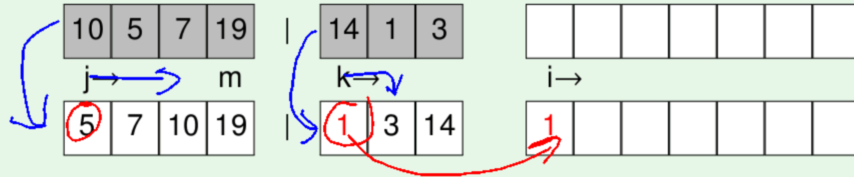
Beispiel (Verschmelzen)



MergeSort

Sortieren durch Verschmelzen

Beispiel (Verschmelzen)

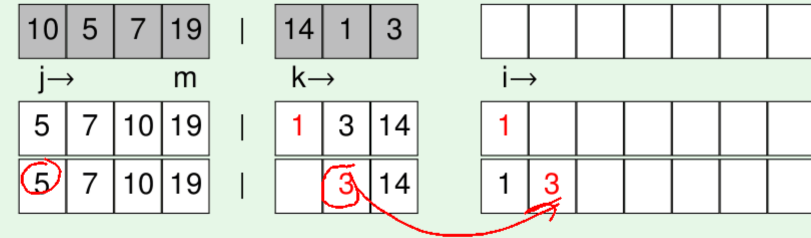


Sortieren MergeSort

MergeSort

Sortieren durch Verschmelzen

Beispiel (Verschmelzen)

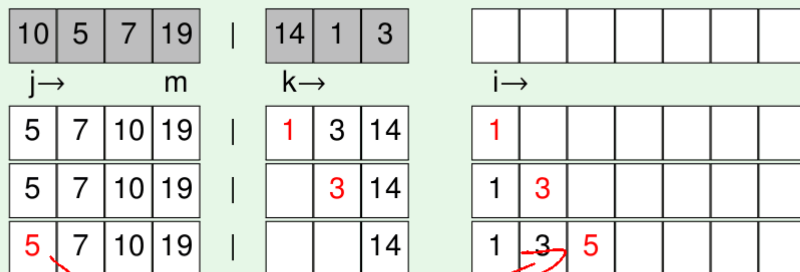


Sortieren MergeSort

MergeSort

Sortieren durch Verschmelzen

Beispiel (Verschmelzen)

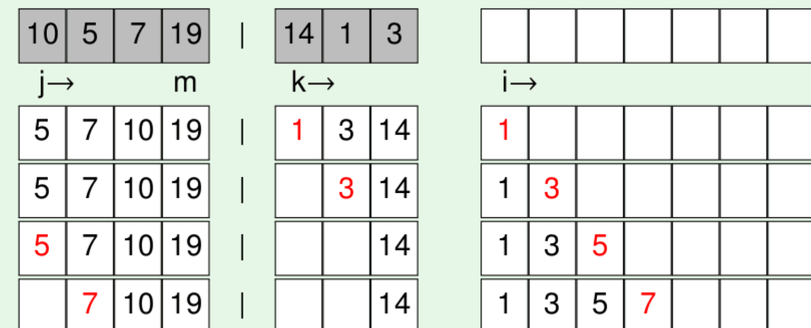


Sortieren MergeSort

MergeSort

Sortieren durch Verschmelzen

Beispiel (Verschmelzen)



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

- 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:

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

bzw.

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

Master-Theorem

Lösung von Rekursionsgleichungen

Satz (vereinfachtes Master-Theorem)

Seien a, b, c, d positive Konstanten und $n = b^k$ mit $k \in \mathbb{N}$.

Betrachte folgende Rekursionsgleichung:

$$r(n) = \begin{cases} a & \text{falls } n = 1, \\ cn + d \cdot r(n/b) & \text{falls } n > 1. \end{cases}$$

Dann gilt:

$$r(n) = \begin{cases} \Theta(n) & \text{falls } d < b, \\ \Theta(n \log n) & \text{falls } d = b, \\ \Theta(n^{\log_b d}) & \text{falls } d > b. \end{cases}$$

Übersicht

6 Sortieren
- Einfache Verfahren
- MergeSort
- Untere Schranke
- QuickSort
- Selektieren
- Schnelleres Sortieren
- Externes Sortieren

Untere Schranke

MergeSort hat Laufzeit $\mathcal{O}(n \log n)$ im worst case.

insertionSort kann so implementiert werden, dass es im best case nur lineare Laufzeit hat

Gibt es Sortierverfahren mit Laufzeit besser als $\mathcal{O}(n \log n)$ im worst case, z.B. $\mathcal{O}(n)$ oder $\mathcal{O}(n \log \log n)$?

$\Rightarrow$ nicht auf der Basis einfacher Schlüsselvergleiche

Entscheidungen: $x_i < x_j \quad \rightarrow \quad$ ja / nein

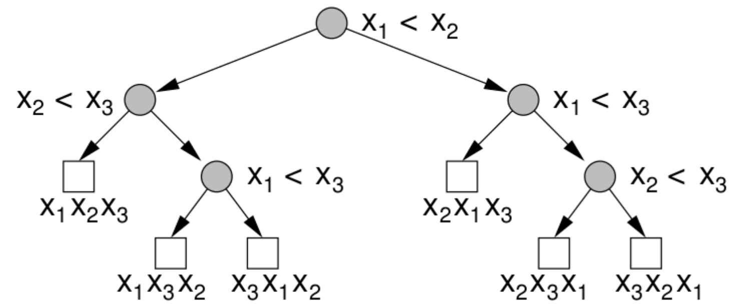
Satz

Jeder vergleichsbasierte Sortieralgorithmus benötigt im worst case mindestens $n \log n - \mathcal{O}(n) \in \Theta(n \log n)$ Vergleiche.

Untere Schranke

Vergleichsbasiertes Sortieren

Entscheidungsbaum mit Entscheidungen an den Knoten:



Untere Schranke

Vergleichsbasiertes Sortieren

muss insbesondere auch funktionieren, wenn alle n Schlüssel verschieden sind

⇒ Annahme: alle verschieden

Wieviele verschiedene Ergebnisse gibt es?
⇒ alle Permutationen:

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n\left(1+\mathcal{O}\left(\frac{1}{n}\right)\right) \geq \frac{n^n}{e^n}\sqrt{2\pi n}$$

Binärbaum der Höhe h hat höchstens 2^h Blätter bzw.
Binärbaum mit b Blättern hat mindestens Höhe $\log_2 b$

$\Rightarrow h \geq \log_2(n!) \geq n\log n - n\log e + \frac{1}{2}\log(2\pi n)$