Script generated by TTT

Title: Seidl: GAD (02.06.2015)

Date: Tue Jun 02 13:56:20 CEST 2015

Duration: 137:44 min

Pages: 72

Übersicht

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

SelectionSort

Sortieren durch Auswählen

Wähle das kleinste Element aus der (verbleibenden) Eingabesequenz und verschiebe es an das Ende der Ausgabesequenz

Beispiel

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

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

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, 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 ≥ 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

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, 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 ≥ 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)

Übersicht

MergeSort

Sortieren durch Verschmelzen

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

Beispiel

10	5	7	19	14	1	3

MergeSort

Sortieren durch Verschmelzen

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

Beispiel

10	5	7	19	14	1	3

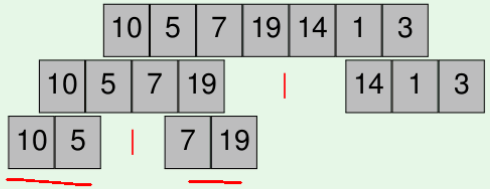
10	5	7	19		14	1	3

MergeSort

Sortieren durch Verschmelzen

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

Beispiel

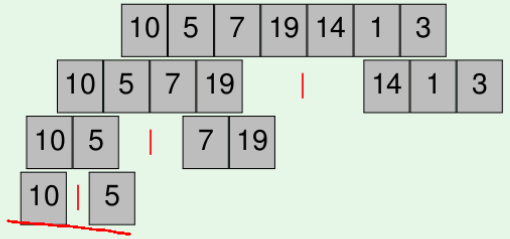


MergeSort

Sortieren durch Verschmelzen

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

Beispiel

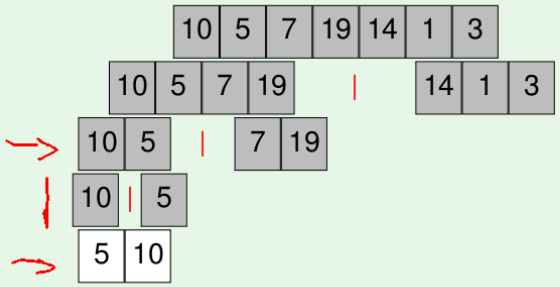


MergeSort

Sortieren durch Verschmelzen

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

Beispiel

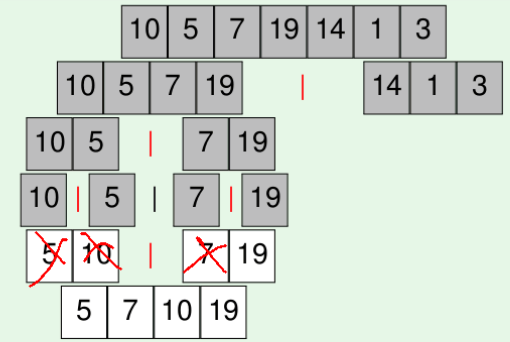


MergeSort

Sortieren durch Verschmelzen

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

Beispiel



MergeSort

Sortieren durch Verschmelzen

```
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

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

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

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

Sortieren MergeSort

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

H. Seidl (TUM) GAD SS'15 235

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.

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

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

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

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
- $d < b$: Kosten $ad^k < ab^k = an \in O(n)$ auf Level k,

Rest: $$cn \sum_{i=0}^{k-1} \left(\frac{d}{b}\right)^i = cn \frac{1 - (d/b)^k}{1 - d/b} < cn \frac{1}{1 - d/b} \in O(n)$$

$$> cn \in \Omega(n) \quad \Rightarrow \quad \Theta(n)$$

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
- $d < b$: Kosten $ad^k < ab^k = an \in O(n)$ auf Level k,

Rest: $$cn \sum_{i=0}^{k-1} \left(\frac{d}{b}\right)^i = cn \frac{1 - (d/b)^k}{1 - d/b} < cn \frac{1}{1 - d/b} \in O(n)$$

$$> \quad cn \in \Omega(n) \quad \Rightarrow \quad \Theta(n)$$

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
- $d < b$: Kosten $ad^k < ab^k = an \in O(n)$ auf Level k,

Rest: $$cn \sum_{i=0}^{k-1} \left(\frac{d}{b}\right)^i = cn \frac{1 - (d/b)^k}{1 - d/b} < cn \frac{1}{1 - d/b} \in O(n)$$

$$> \quad cn \in \Omega(n) \quad \Rightarrow \quad \Theta(n)$$

H. Seidl (TUM) GAD SS'15 238

Sortieren MergeSort

Analyse rekursiver Funktionen

Divide-and-Conquer-Algorithmen

- $d > b$: $n = b^k$, also $k = \log_b n = \log_b d \cdot \log_d n$

$$d^k = d^{\log_b n} = d^{\log_d n \cdot \log_b d} = n^{\log_b d}$$

Kosten $an^{\log_b d} \in \Theta\left(n^{\log_b d}\right)$ auf Level k,

Rest: $$cb^k \frac{(d/b)^k - 1}{d/b - 1} = c \frac{d^k - b^k}{d/b - 1}$$

$$= cd^k \frac{1 - (b/d)^k}{d/b - 1} \in \Theta\left(d^k\right) \in \Theta\left(n^{\log_b d}\right)$$

Sortieren MergeSort

Analyse rekursiver Funktionen

Divide-and-Conquer-Algorithmen

- $d > b$: $n = b^k$, also $k = \log_b n = \log_b d \cdot \log_d n$

$$d^k = d^{\log_b n} = d^{\log_d n \cdot \log_b d} = n^{\log_b d}$$

Kosten $an^{\log_b d} \in \Theta\left(n^{\log_b d}\right)$ auf Level k,

Rest: $$cb^k \frac{(d/b)^k - 1}{d/b - 1} = c \frac{d^k - b^k}{d/b - 1}$$

$$= cd^k \frac{1 - (b/d)^k}{d/b - 1} \in \Theta\left(d^k\right) \in \Theta\left(n^{\log_b d}\right)$$

H. Seidl (TUM) GAD SS'15 239

Analyse rekursiver Funktionen

Divide-and-Conquer-Algorithmen

- $d > b$: $n = b^k$, also $k = \log_b n = log_b d \cdot \log_d n$

$$d^k = d^{\log_b n} = d^{log_d n \cdot \log_b d} = n^{\log_b d}$$

Kosten $an^{\log_b d} \in \Theta\left(n^{\log_b d}\right)$ auf Level k,

Rest: $$cb^k \frac{(d/b)^k - 1}{d/b - 1} = c\frac{d^k - b^k}{d/b - 1}$$
$$= cd^k \frac{1 - (b/d)^k}{d/b - 1} \in \Theta\left(d^k\right) \in \Theta\left(n^{\log_b d}\right)$$

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:

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

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

Untere Schranke

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

InsertionSort kann so implementiert werden, dass es im best case lineare Laufzeit hat.

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

$\Rightarrow$ nicht auf der Basis einfacher Schlüsselvergleiche

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

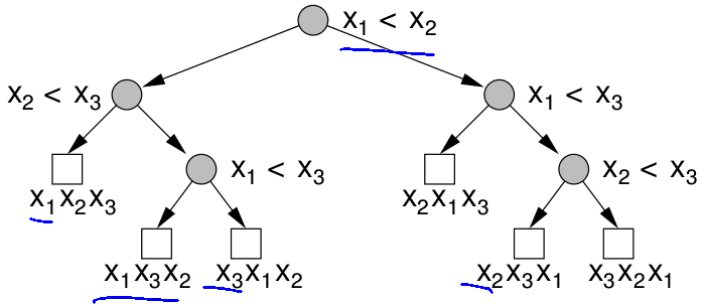
Satz

Jeder vergleichsbasierte Sortieralgorithmus benötigt im worst case mindestens $n \log n - O(n) \in \Theta(n \log n)$ 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

$\Rightarrow$ Annahme: alle verschieden

Wieviele verschiedene Ergebnisse gibt es?
$\Rightarrow$ alle Permutationen:

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

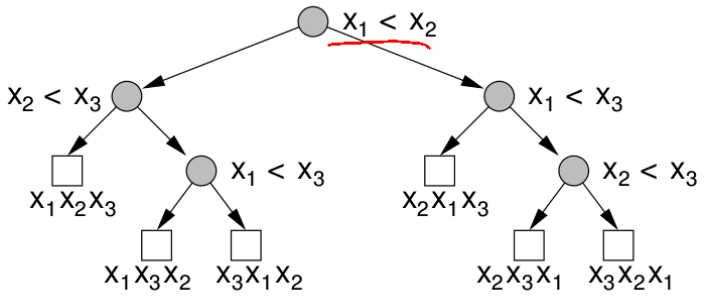
Binärbaum der Höhe h hat höchstens 2^h Blätter bzw.
Binärbaum mit b Blättern hat mindestens Höhe $\log_2 b$

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

Untere Schranke

Vergleichsbasiertes Sortieren

Entscheidungsbaum mit Entscheidungen an den Knoten:



Untere Schranke

Vergleichsbasiertes Sortieren

muss insbesondere auch funktionieren, wenn alle n Schlüssel verschieden sind

$\Rightarrow$ Annahme: alle verschieden

Wieviele verschiedene Ergebnisse gibt es?
$\Rightarrow$ alle Permutationen:

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

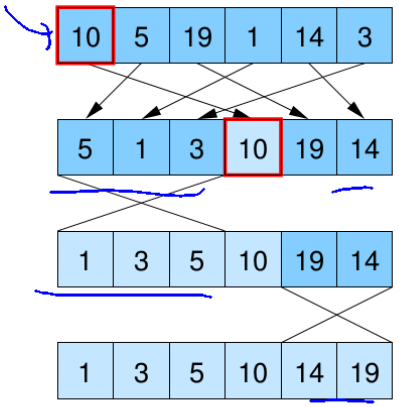
Binärbaum der Höhe h hat höchstens 2^h Blätter bzw.
Binärbaum mit b Blättern hat mindestens Höhe $\log_2 b$

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

Übersicht

QuickSort

Idee:
Aufspaltung in zwei Teilmengen, aber nicht in der Mitte der Sequenz wie bei MergeSort, sondern getrennt durch ein Pivotelement



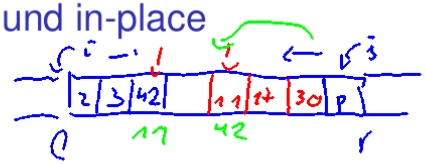
QuickSort abstrakt

```
quickSort {
    Wähle Pivotelement;
    // z.B. erstes, mittleres, letztes oder zufälliges Element

    Splitte in kleinere und größere Schlüssel bzgl. Pivotelement;
    // entweder in temporäre Arrays oder in-place
    // ein Scan des Felds    ⇒   O(n) Zeit

    Sortiere Teilfeld mit kleineren Schlüsseln (rekursiv);
    Sortiere Teilfeld mit größeren Schlüsseln (rekursiv);
}
```

Implementierung: effizient und in-place



```
quickSort(Element[] a, int ℓ, int r) {
    // a[ℓ ... r]: zu sortierendes Feld
    if (ℓ < r) {
        p = a[r];   // Pivot
        int i = ℓ − 1;   int j = r;
        do {   // spalte Elemente in a[ℓ, ..., r − 1] nach Pivot p
            do { i++ } while (a[i] < p);
            do { j−− } while (j ≥ ℓ ∧ a[j] > p);
            if (i < j) swap(a, i, j);
        } while (i < j);
        swap (a, i, r);   // Pivot an richtige Stelle
        quickSort(a, ℓ, i − 1);
        quickSort(a, i + 1, r);
    }
}
```


QuickSort: Laufzeit

Problem:

- im Gegensatz zu MergeSort kann die Aufteilung in Teilprobleme unbalanciert sein (also nur im Optimalfall eine Halbierung)
- im worst case quadratische Laufzeit
  (z.B. wenn Pivotelement immer kleinstes oder größtes aller Elemente ist)

QuickSort: Laufzeit

Problem:

- im Gegensatz zu MergeSort kann die Aufteilung in Teilprobleme unbalanciert sein (also nur im Optimalfall eine Halbierung)
- im worst case quadratische Laufzeit
  (z.B. wenn Pivotelement immer kleinstes oder größtes aller Elemente ist)

Lösungen:

- wähle zufälliges Pivotelement:
  Laufzeit $O(n \log n)$ mit hoher Wahrscheinlichkeit
- berechne Median (mittleres Element):
  mit Selektionsalgorithmus, später in der Vorlesung

QuickSort

Laufzeit bei zufälligem Pivot-Element

- Zähle Anzahl Vergleiche (Rest macht nur konstanten Faktor aus)
- $\bar{C}(n)$: erwartete Anzahl Vergleiche bei n Elementen

Satz

Die erwartete Anzahl von Vergleichen für QuickSort mit zufällig ausgewähltem Pivotelement ist

$$\bar{C}(n) \leq 2n \ln n \leq 1.39 n \log_2 n$$

QuickSort

Beweis.

- Betrachte sortierte Sequenz $\langle e'_1, \ldots, e'_n \rangle$
- nur Vergleiche mit Pivotelement
- Pivotelement ist nicht in den rekursiven Aufrufen enthalten

$\Rightarrow$ e'_i und e'_j werden höchstens einmal verglichen und zwar dann, wenn e'_i oder e'_j Pivotelement ist

QuickSort

Beweis.

- Zufallsvariable $X_{ij} \in \{0, 1\}$
- $X_{ij} = 1 \quad \Leftrightarrow \quad e'_i$ und e'_j werden verglichen

$$\begin{aligned} \bar{C}(n) &= \mathbb{E}\left[\sum_{i<j} X_{ij}\right] = \sum_{i<j} \mathbb{E}\left[X_{ij}\right] \\ &= \sum_{i<j} \left(0 \cdot \Pr\left[X_{ij} = 0\right] + 1 \cdot \Pr\left[X_{ij} = 1\right]\right) \\ &= \sum_{i<j} \Pr\left[X_{ij} = 1\right] \end{aligned}$$

QuickSort

Lemma

$$\Pr[X_{ij} = 1] = 2/(j - i + 1)$$

Beweis.

- Sei $M = \{e'_i, \ldots, e'_j\}$
- Irgendwann wird ein Element aus M als Pivot ausgewählt.
- Bis dahin bleibt M immer zusammen.
- e'_i und e'_j werden genau dann *direkt* verglichen, wenn eines der beiden als Pivot ausgewählt wird
- Wahrscheinlichkeit:

$$\Pr[e'_i \text{ oder } e'_j \text{ aus } M \text{ ausgewählt}] = \frac{2}{|M|} = \frac{2}{j - i + 1}$$

□

QuickSort

Beweis.

$$\begin{aligned} \bar{C} &= \sum_{i<j} \Pr\left[X_{ij} = 1\right] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j - i + 1} \\ &= \sum_{i=1}^{n-1} \sum_{k=2}^{n-i+1} \frac{2}{k} = 2 \sum_{i=1}^{n-1} \sum_{k=2}^{n-i+1} \frac{1}{k} \\ &\leq 2 \sum_{i=1}^{n-1} \sum_{k=2}^{n} \frac{1}{k} = 2(n-1) \sum_{k=2}^{n} \frac{1}{k} = 2(n-1)(H_n - 1) \\ &\leq 2(n-1)(1 + \ln n - 1) \leq 2n \ln n = 2n \ln(2) \log_2(n) \end{aligned}$$

□

QuickSort

Beweis.

$$\begin{aligned}
\bar{C} &= \sum_{i<j} \Pr\left[X_{ij}=1\right] = \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} \frac{2}{j-i+1} \\
&= \sum_{i=1}^{n-1}\sum_{k=2}^{n-i+1} \frac{2}{k} = 2\sum_{i=1}^{n-1}\sum_{k=2}^{n-i+1} \frac{1}{k} \\
&\le 2\sum_{i=1}^{n-1}\sum_{k=2}^{n} \frac{1}{k} = 2(n-1)\sum_{k=2}^{n} \frac{1}{k} = 2(n-1)(H_n-1) \\
&\le 2(n-1)(1+\ln n-1) \le 2n\ln n = 2n\ln(2)\log_2(n)
\end{aligned}$$

□

QuickSort

Beweis.

$$\begin{aligned}
\bar{C} &= \sum_{i<j} \Pr\left[X_{ij}=1\right] = \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} \frac{2}{j-i+1} \\
&= \sum_{i=1}^{n-1}\sum_{k=2}^{n-i+1} \frac{2}{k} = 2\sum_{i=1}^{n-1}\sum_{k=2}^{n-i+1} \frac{1}{k} \\
&\le 2\sum_{i=1}^{n-1}\sum_{k=2}^{n} \frac{1}{k} = 2(n-1)\sum_{k=2}^{n} \frac{1}{k} = 2(n-1)(H_n-1) \\
&\le 2(n-1)(1+\ln n-1) \le 2n\ln n = 2n\ln(2)\log_2(n)
\end{aligned}$$

□

QuickSort

Verbesserte Version ohne Check für Array-Grenzen

```
qSort(Element[] a, int ℓ, int r) {
    while (r − ℓ ≥ n0) {
        j = pickPivotPos(a, ℓ, r);
        swap(a, ℓ, j);      p = a[ℓ];
        int i = ℓ;      int j = r;
        repeat {
            while (a[i] < p)    i++;
            while (a[j] > p)    j−−;
            if (i ≤ j) { swap(a, i, j);   i++;   j−−; }
        } until (i > j);
        if (i < (ℓ + r)/2) { qSort(a, ℓ, j);   ℓ = i; }
        else               { qSort(a, i, r);   r = j; }
    }
    insertionSort(a, ℓ, r);
}
```


Übersicht

Rang-Selektion

- Bestimmung des kleinsten und größten Elements ist mit einem einzigen Scan über das Array in Linearzeit möglich
- Aber wie ist das beim k-kleinsten Element, z.B. beim $\lfloor n/2 \rfloor$-kleinsten Element (Median)?

Problem:

Finde k-kleinstes Element in einer Menge von n Elementen

Rang-Selektion

- Bestimmung des kleinsten und größten Elements ist mit einem einzigen Scan über das Array in Linearzeit möglich
- Aber wie ist das beim k-kleinsten Element, z.B. beim $\lfloor n/2 \rfloor$-kleinsten Element (Median)?

Problem:

Finde k-kleinstes Element in einer Menge von n Elementen

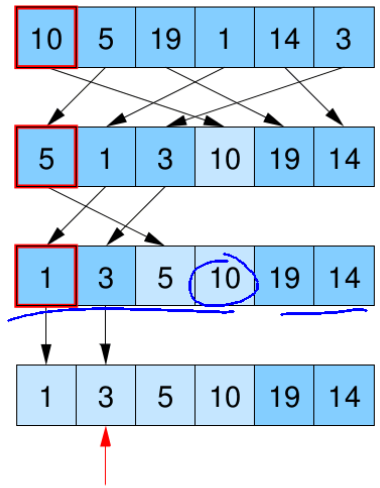
Naive Lösung: Sortieren und k-tes Element ausgeben

$\Rightarrow$ Zeit $O(n \log n)$

Geht das auch schneller?

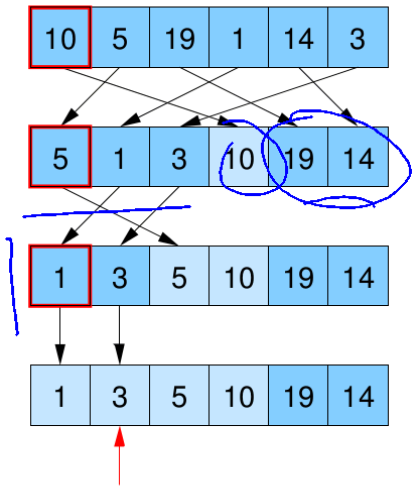
QuickSelect

Ansatz: ähnlich zu QuickSort, aber nur eine Seite betrachten



QuickSelect

Ansatz: ähnlich zu QuickSort, aber nur eine Seite betrachten



QuickSelect

Methode analog zu QuickSort

```
Element quickSelect(Element[] a, int l, int r, int k) {
  // a[l...r]: Restfeld,   k: Rang des gesuchten Elements
  if (r == l) return a[l];
  int z = zufällige Position in {l,...,r};  swap(a,z,r);
  Element p = a[r];  int i = l-1;  int j = r;
  do {  // spalte Elemente in a[l,...,r-1] nach Pivot p
    do i++ while (a[i] < p);
    do j-- while (a[j] > p && j ≠ l);
    if (i < j) swap(a,i,j);
  } while (i < j);
  swap (a,i,r);  // Pivot an richtige Stelle
  if (k < i) return quickSelect(a, l, i-1, k);
  if (k > i) return quickSelect(a, i+1, r, k);
  else return a[k];  // k == i
}
```


QuickSelect

Alternative Methode

```
Element select(Element[] s, int k) {
  assert(|s| ≥ k);
  Wähle p ∈ s zufällig (gleichverteilt);

  Element[] a := {e ∈ s : e < p};
  if (|a| ≥ k)
    return select(a,k);

  Element[] b := {e ∈ s : e = p};
  if (|a| + |b| ≥ k)
    return p;

  Element[] c := {e ∈ s : e > p};
  return select(c,k - |a| - |b|);
}
```


QuickSelect

Alternative Methode

```
Element select(Element[] s, int k) {
  assert(|s| ≥ k);
  Wähle p ∈ s zufällig (gleichverteilt);

  Element[] a := {e ∈ s : e < p};
  if (|a| ≥ k)
    return select(a,k);

  Element[] b := {e ∈ s : e = p};
  if (|a| + |b| ≥ k)
    return p;

  Element[] c := {e ∈ s : e > p};
  return select(c,k - |a| - |b|);
}
```


QuickSelect

Alternative Methode

Beispiel

s	k	a b c
⟨3,1,4,1,5,9,2,6,5,3,5,8,9⟩	7	⟨1,1⟩ ⟨2⟩ ⟨3,4,5,9,6,5,3,5,8,9⟩
⟨3,4,5,9,6,5,3,5,8,9⟩	4	⟨3,4,5,5,3,5⟩ ⟨6⟩ ⟨9,8,9⟩
⟨3,4,5,5,3,5⟩	4	⟨3,4,3⟩ ⟨5,5,5⟩ ⟨⟩

In der sortierten Sequenz würde also an 7. Stelle das Element 5 stehen.

Hier wurde das mittlere Element als Pivot verwendet.

QuickSelect

Alternative Methode

Beispiel

s	k	a b c
$\langle 3,1,4,1,5,9,2,6,5,3,5,8,9\rangle$	7	$\langle 1,1\rangle\ \langle 2\rangle\ \langle 3,4,5,9,6,5,3,5,8,9\rangle$
$\langle 3,4,5,9,6,5,3,5,8,9\rangle$	4	$\langle 3,4,5,5,3,5\rangle\ \langle 6\rangle\ \langle 9,8,9\rangle$
$\langle 3,4,5,5,3,5\rangle$	4	$\langle 3,4,3\rangle\ \langle 5,5,5\rangle\ \langle\rangle$

In der sortierten Sequenz würde also an 7. Stelle das Element 5 stehen.

Hier wurde das mittlere Element als Pivot verwendet.

QuickSelect

teilt das Feld jeweils in 3 Teile:

- a Elemente kleiner als das Pivot
- b Elemente gleich dem Pivot
- c Elemente größer als das Pivot

$T(n)$: erwartete Laufzeit bei n Elementen

Satz

Die erwartete Laufzeit von QuickSelect ist linear: $T(n) \in O(n)$.

QuickSelect

Beweis.

- Pivot ist gut, wenn weder a noch c länger als 2/3 der aktuellen Feldgröße sind:

$\Rightarrow$ Pivot ist gut, falls es im mittleren Drittel liegt

$$p = \Pr[\text{Pivot ist gut}] = 1/3$$

QuickSelect

Beweis.

- Pivot ist gut, wenn weder a noch c länger als 2/3 der aktuellen Feldgröße sind:

$\Rightarrow$ Pivot ist gut, falls es im mittleren Drittel liegt

$$p = \Pr[\text{Pivot ist gut}] = 1/3$$

Erwartete Zeit bei n Elementen

- linearer Aufwand außerhalb der rekursiven Aufrufe: cn
- Pivot gut (Wsk. 1/3): Restaufwand $\leq T(2n/3)$
- Pivot schlecht (Wsk. 2/3): Restaufwand $\leq T(n-1) < T(n)$

QuickSelect

Beweis.

$$
\begin{aligned}
T(n) &\leq cn + p \cdot T(n \cdot 2/3) + (1-p) \cdot T(n) \\
p \cdot T(n) &\leq cn + p \cdot T(n \cdot 2/3) \\
T(n) &\leq cn/p + T(n \cdot 2/3) \\
&\leq cn/p + c \cdot (n \cdot 2/3)/p + T(n \cdot (2/3)^2) \\
&\ldots \text{ wiederholtes Einsetzen} \\
&\leq (cn/p)(1 + 2/3 + 4/9 + 8/27 + \ldots) \\
&\leq \frac{cn}{p} \cdot \sum_{i \geq 0} (2/3)^i \\
&\leq \frac{cn}{1/3} \cdot \frac{1}{1-2/3} = 9cn \in O(n)
\end{aligned}
$$

□