Übung Algorithmen und Datenstrukturen

Sommersemester 2017

Patrick Schäfer, Humboldt-Universität zu Berlin

Agenda: Suchen und Amortisierte Analyse

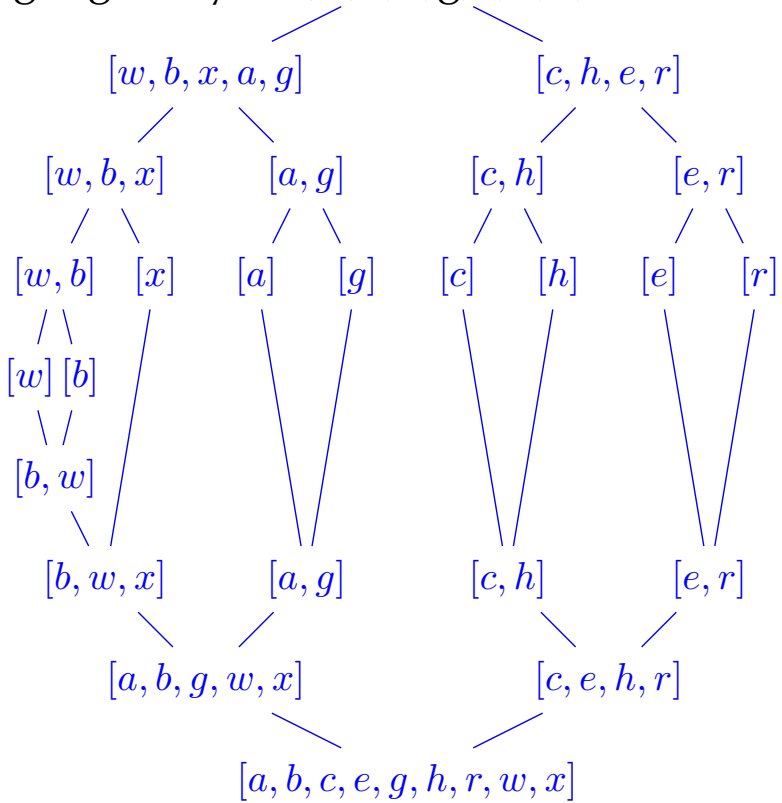
- Vorrechnen (first-come first-served)
  - Gruppe 5 13-15 Uhr https://dudle.inf.tu-dresden.de/AlgoDatGr5U3/
  - Gruppe 6 15-17 Uhr https://dudle.inf.tu-dresden.de/AlgoDatGr6U3/

Übung: https://hu.berlin/algodat17

Vorlesung: https://hu.berlin/vl_algodat17

MergeSort

a) Ausgangsarray: [w,b,x,a,g,c,h,e,r]



Algorithmus MergeSort(A, l, r)

Input: Array A, l linke und r rechte Grenze
Output: Sortiertes Array A

```
1: if l < r then
2:     m := (r − l) div 2;            # sortiere beide Hälften
3:     mergesort(A, l, l + m);
4:     mergesort(A, l + m + 1, r);
5:     merge(A, l, l + m, r);         # Zusammenführen der sortieren Listen
6: end if
7: return A;
```

QuickSort

b) Ausgangsarray: [8, 6, 2, 7, 1, 4, 3, 5]

[3, 6, 2, 7, 1, 4, 8, 5]
[3, 4, 2, 7, 1, 6, 8, 5]
[3, 4, 2, 1, 7, 6, 8, 5]
[3, 4, 2, 1, 5, 6, 8, 7]
[1, 4, 2, 3, 5, 6, 8, 7]
[1, 2, 4, 3, 5, 6, 8, 7]
[1, 2, 3, 4, 5, 6, 7, 8]

Algorithmus divide(A, l, r)

Input: Array A, l linke und r rechte Grenze
Output: Sortiertes Array A

```
1:  val := A[r];
2:  i := l;
3:  j := r − 1;
4:  repeat
5:    while A[i] <= val and i < r do
6:      i := i + 1;                  # Element kleiner als Pivot
7:    end while
8:    while A[j] >= val and j > l do
9:      j := j − 1;                  # Element größer als Pivot
10:   end while
11:   if i < j then
12:     swap(A[i], A[j]);
13:   end if
14: until i >= j
15: swap(A[i], A[r]);
16: return i;
```

BucketSort

c) Eingabe: A = [203, 202, 100, 123, 121, 323, 103, 211, 320]

A_1: [100, 320, | 121, 211, | 202, | 203, 123, 323, 103]

A_2: [100, 202, 203, 103,| 211, | 320, 121, 123, 323|]

A_3: [| 100, 103, 121, 123,| 202, 203, 211,| 320, 323]

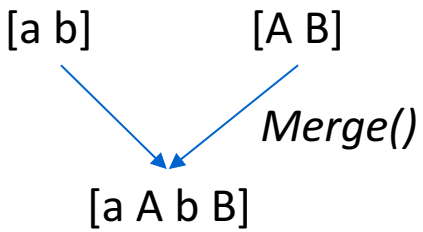
Stabilität (QuickSort)

- **QuickSort** ist nicht stabil.
- Beispiel [c,a,A,b] mit a.key = A.key

[A, a, c, b]
[A, a, b, c]
[A, a, b, c]

Stabilität (MergeSort)

- **MergeSort** ist stabil.
- Beim Aufteilen wird die Reihenfolge nicht verändert.
- Beim *Merge* werden bei gleichen Elementen erst die Elemente des "linken" Teil-Arrays und dann die Elemente des "rechten" Teil-Arrays eingefügt.



Algorithmus Merge(A, l, m, r)

Input: Array A, l, m und r Intervall-Grenzen
Output: Sortiertes Array A

```
1:  B := array[1..r − l + 1];
2:  i := l;                          # Anfang der 1. Liste
3:  j := m + 1;                      # Anfang der 2. Liste
4:  k := 1;                          # Ergebnisliste
5:  while (i <= m) and (j <= r) do
6:    if A[i] <= A[j] then
7:      B[k] := A[i + +];            # Aus der 1. Liste
8:    else
9:      B[k] := A[j + +];            # Aus der 2. Liste
10:   end if
11:   k := k + 1;
12: end while
13: if i > m then
14:   copy A[j..r] to B[k..k + r − j];
15: else
16:   copy A[i..m] to B[k..k + m − i];
17: end if
18: copy B[1..r − l + 1] to A[l..r]; # Zurück ins Array A schreiben
19: return A;
```

Sortierung spezieller Arrays

d) . . . das Array so vorsortiert ist, dass alle Elemente der ersten Hälfte des Arrays kleiner sind als alle Elemente der zweiten Hälfte.

Falsch: Jede Hälfte des Arrays muss noch sortiert werden und dafür sind, laut unterer Schranke für allgemeine Sortierverfahren, $\Omega\left(\frac{n}{2}\log\frac{n}{2}\right)$ Vergleiche nötig.

e) . . . im Array nur höchstens $l \in \mathbb{N}_{>0}$ viele unterschiedliche Elemente vorkommen. Hierbei sei l eine Konstante.

Wahr: Folgendes Verfahren sortiert das Array in $O(n)$:

1. Lege ein Array X der Länge l an, wobei in jedem Eintrag ein Tupel (Key, Zähler) steht : $O(l)$
2. Für jedes Element der Eingabe vergleicht man der Reihe nach die Tupel in X und zählt beim passenden Zähler um 1 hoch oder legt ein neues Tupel an : $O(nl)$
3. Anschließend sortiert man das Array X nach den Tupel-Keys (mit einem beliebigen allgemeinen Sortierverfahren) und baut mit Hilfe des sortierten Arrays X und den Zählerständen die korrekte Ausgabefolge zusammen: $O(l \log l) + O(n)$

Untere Schranke für allgemeine Sortierverfahren

Ein allgemeines Sortierverfahren kann im Worst Case nicht schneller sein als:

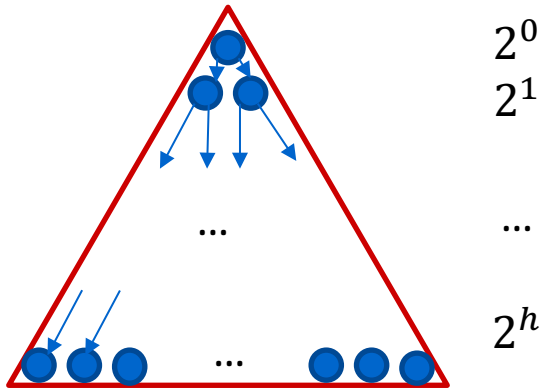
$$\Omega(n \log n)$$

Für ein Array mit n Elementen gibt es $n!$ mögliche Permutationen. In einem Entscheidungsbaum mit $n!$ Blättern, der vollständig balanciert ist, hat ein Blatt eine minimale Tiefe von $h \geq \log\ (n!)$:

$$h \geq \log(n!) > \frac{n}{2} \log \frac{n}{2} \in\ \Omega(n \log \mathrm{n})$$

Hinweis:

$$\begin{aligned} log\, n! &= log(1)+log(2)+\ldots+log(n-1)+log(n) \\ &> log\left(\frac{n}{2}\right)+log\left(\frac{n}{2}+1\right)+\ldots+log(n) \\ &> log\left(\frac{n}{2}\right)+log\left(\frac{n}{2}\right)+\ldots+log(\frac{n}{2}) \\ &= \frac{n}{2}\ log\frac{n}{2} \end{aligned}$$



Allgemeine Sortierverfahren

b) Beweisen oder widerlegen Sie, dass es ein allgemeines Sortierverfahren gibt, welches immerhin für einen Anteil von $\frac{1}{2^n}$ aller $n!$ vielen Eingaben in Zeit $O(n)$ sortiert.

- Das Verfahren muss alle der $\frac{n!}{2^n}$ Eingaben unterscheiden können. Somit muss der Entscheidungsbaum $\frac{n!}{2^n}$ Blätter haben. Ein vollständig balancierter Baum hat somit eine minimale Tiefe von $h \geq \log\left(\frac{n!}{2^n}\right)$:

$$h \geq \log\left(\frac{n!}{2^n}\right) = \log(n!) - n > \frac{n}{2}\log\frac{n}{2} - n \in \Omega(n \log \mathrm{n})$$