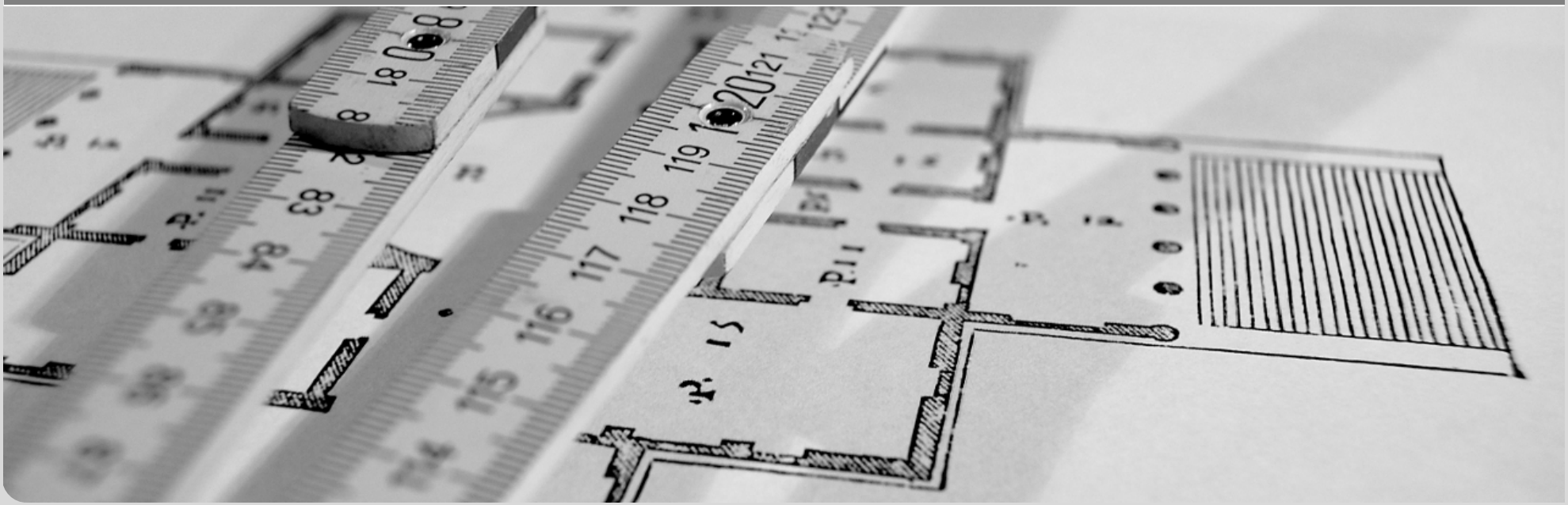
Karlsruher Institut für Technologie

Tutorium 2:
O-Kalkül, Felder und Listen

Holger Ebhart | 29. April 2015

TUTORIUM ZUR VORLESUNG ALGORITHMEN I IM SS15



Gliederung

Organisatorisches

- Mail: **holger.ebhart@ira.uka.de**
- Folien auf github: **github.com/holger-e/Algorithmen-I-SS15**
- Ab sofort Deckblätter verwenden
  - https://webinscribe.ira.uka.de/deckblatt/index.php?course=10516
- Bei Fragen $\rightarrow$ ilias
  - Tutoriumsforum
  - allgemeines Forum

Aufgaben zum O-Kalkül

- $2^n = \mathcal{O}(3^n)$
- $\sqrt{n} = \mathcal{O}(\log n)$
- $n^2(1 + \sqrt{n}) = \mathcal{O}(n^2 \log n)$
- $3n^2 + \sqrt{n} = \mathcal{O}(n^2)$
- $n^2 = \mathcal{O}(2^n)$
- $2^{2n} = \mathcal{O}(2^n)$
-
- $f(n) + g(n) = \mathcal{O}(max(f(n), g(n)))$
- $[f(n) + g(n)]^2 = \mathcal{O}(f(n)^2) + \mathcal{O}(g(n)^2)$

Aufgaben zum O-Kalkül

- $2^n = \mathcal{O}(3^n)$
- $\sqrt{n} = \mathcal{O}(\log n)$
- $n^2(1 + \sqrt{n}) = \mathcal{O}(n^2 \log n)$
- $3n^2 + \sqrt{n} = \mathcal{O}(n^2)$
- $n^2 = \mathcal{O}(2^n)$
- $2^{2n} = \mathcal{O}(2^n)$
-
- $f(n) + g(n) = \mathcal{O}(max(f(n), g(n)))$
- $[f(n) + g(n)]^2 = \mathcal{O}(f(n)^2) + \mathcal{O}(g(n)^2)$

Aufgaben zum O-Kalkül

1. $f(n) = \log n^2; g(n) = \log n + 5$
2. $f(n) = \sqrt{n}; g(n) = \log n^2$
3. $f(n) = n \log n + n; g(n) = \log n$
4. $f(n) = n; g(n) = \log^2 n$

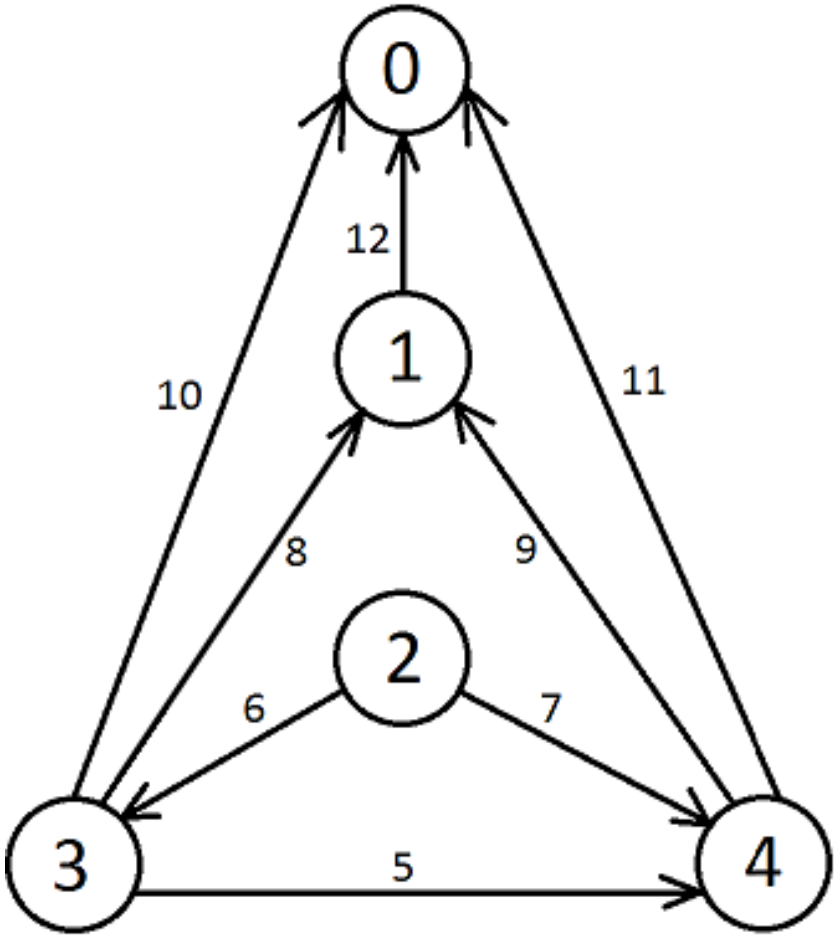
Aufgaben zum O-Kalkül

1. $f(n) = 4n \log n + n; g(n) = \frac{n^2 - n}{2}$
2. $f(n) = n + \log n; g(n) = \sqrt{n})$
3. $f(n) = \frac{n^2 - n}{2}; g(n) = 6n$

Aufgaben zum Mastertheorem

- $T(n) = 4 \cdot T(\frac{n}{2}) + n$
- $T(n) = 4 \cdot T(\frac{n}{2}) + n^2$
- $T(n) = 4 \cdot T(\frac{n}{2}) + n^3$
- $T(n) = 8 \cdot T(\frac{n}{2}) + n^3$
- $T(n) = 8 \cdot T(\frac{n}{2}) + n^4$

DAG-Erkennung



DAG-Erkennung

```
1 function isDAG(G=(V,E))
2         while node v with outdegree 0 exists do
3                 remove v from V;
4                 remove all edges containing v from E;
5         end
6         return |V| = 0;
```

Laufzeit: $\mathcal{O}(|V| + |E|)$

DAG-Erkennung

```
1 function isDAG(G=(V,E))
2         while node v with outdegree 0 exists do
3                 remove v from V;
4                 remove all edges containing v from E;
5         end
6         return |V| = 0;
```

Laufzeit: $\mathcal{O}(|V| + |E|)$

Linked-List

```
1 class Item of Element
2         e:Element;
3         next:Handle;
4         prev:Handle;
```

Linked-List

- SList: Zeiger auf *front* und *end*
- isEmpty()
- first():Handle
- last():Handle
- dummy header
- findNext(e:Element, b:Handle):Handle
- Sentinel
- !!!Sonderfälle!!!

Linked-List

- SList: Zeiger auf *front* und *end*
- isEmpty()
- first():Handle
- last():Handle
- dummy header
- findNext(e:Element, b:Handle):Handle
- Sentinel
- !!!Sonderfälle!!!

Linked-List

- SList: Zeiger auf *front* und *end*
- isEmpty()
- first():Handle
- last():Handle
- dummy header
- findNext(e:Element, b:Handle):Handle
- Sentinel
- !!!Sonderfälle!!!

Linked-List

- SList: Zeiger auf *front* und *end*
- isEmpty()
- first():Handle
- last():Handle
- dummy header
- findNext(e:Element, b:Handle):Handle
- Sentinel
- !!!Sonderfälle!!!

Linked-List

- SList: Zeiger auf *front* und *end*
- isEmpty()
- first():Handle
- last():Handle
- dummy header
- findNext(e:Element, b:Handle):Handle
- Sentinel
- !!!Sonderfälle!!!

Linked-List

- SList: Zeiger auf *front* und *end*
- isEmpty()
- first():Handle
- last():Handle
- dummy header
- findNext(e:Element, b:Handle):Handle
- Sentinel
- !!!Sonderfälle!!!

Linked-List

- SList: Zeiger auf *front* und *end*
- isEmpty()
- first():Handle
- last():Handle
- dummy header
- findNext(e:Element, b:Handle):Handle
- Sentinel
- !!!Sonderfälle!!!

Linked-List

- SList: Zeiger auf *front* und *end*
- isEmpty()
- first():Handle
- last():Handle
- dummy header
- findNext(e:Element, b:Handle):Handle
- Sentinel
- !!!Sonderfälle!!!

Linked-List

- splice
- moveAfter(a,b:Handle)
- moveToFront(b:Handle)
- remove(b:Handle)
- popBack():Handle
- insertBefore(e:Element, b:Handle)
- pushFront(e:Element)
- concat(l:List)

Linked-List

- splice
- moveAfter(a,b:Handle)
- moveToFront(b:Handle)
- remove(b:Handle)
- popBack():Handle
- insertBefore(e:Element, b:Handle)
- pushFront(e:Element)
- concat(l:List)

Linked-List

- splice
- moveAfter(a,b:Handle)
- moveToFront(b:Handle)
- remove(b:Handle)
- popBack():Handle
- insertBefore(e:Element, b:Handle)
- pushFront(e:Element)
- concat(l:List)

Linked-List

- splice
- moveAfter(a,b:Handle)
- moveToFront(b:Handle)
- remove(b:Handle)
- popBack():Handle
- insertBefore(e:Element, b:Handle)
- pushFront(e:Element)
- concat(l:List)

Linked-List

- splice
- moveAfter(a,b:Handle)
- moveToFront(b:Handle)
- remove(b:Handle)
- popBack():Handle
- insertBefore(e:Element, b:Handle)
- pushFront(e:Element)
- concat(l:List)

Linked-List

- splice
- moveAfter(a,b:Handle)
- moveToFront(b:Handle)
- remove(b:Handle)
- popBack():Handle
- insertBefore(e:Element, b:Handle)
- pushFront(e:Element)
- concat(l:List)

Linked-List

- splice
- moveAfter(a,b:Handle)
- moveToFront(b:Handle)
- remove(b:Handle)
- popBack():Handle
- insertBefore(e:Element, b:Handle)
- pushFront(e:Element)
- concat(l:List)

Linked-List

- splice
- moveAfter(a,b:Handle)
- moveToFront(b:Handle)
- remove(b:Handle)
- popBack():Handle
- insertBefore(e:Element, b:Handle)
- pushFront(e:Element)
- concat(l:List)

Arrays (bounded)

- isEmpty
- first / last
- moveAfter / moveBefore
- moveToFront / moveToBack
- remove
- popBack / popFront
- insertBefore / inserAfter
- pushFront / pushBack
- findNext
- concat(a:Array)

Arrays (bounded)

- isEmpty
- first / last
- moveAfter / moveBefore
- moveToFront / moveToBack
- remove
- popBack / popFront
- insertBefore / inserAfter
- pushFront / pushBack
- findNext
- concat(a:Array)

Arrays (bounded)

- isEmpty
- first / last
- moveAfter / moveBefore
- moveToFront / moveToBack
- remove
- popBack / popFront
- insertBefore / inserAfter
- pushFront / pushBack
- findNext
- concat(a:Array)

Arrays (bounded)

- isEmpty
- first / last
- moveAfter / moveBefore
- moveToFront / moveToBack
- remove
- popBack / popFront
- insertBefore / inserAfter
- pushFront / pushBack
- findNext
- concat(a:Array)

Arrays (bounded)

- isEmpty
- first / last
- moveAfter / moveBefore
- moveToFront / moveToBack
- remove
- popBack / popFront
- insertBefore / inserAfter
- pushFront / pushBack
- findNext
- concat(a:Array)

Arrays (bounded)

- isEmpty
- first / last
- moveAfter / moveBefore
- moveToFront / moveToBack
- remove
- popBack / popFront
- insertBefore / inserAfter
- pushFront / pushBack
- findNext
- concat(a:Array)

Arrays (bounded)

- isEmpty
- first / last
- moveAfter / moveBefore
- moveToFront / moveToBack
- remove
- popBack / popFront
- insertBefore / inserAfter
- pushFront / pushBack
- findNext
- concat(a:Array)

Arrays (bounded)

- isEmpty
- first / last
- moveAfter / moveBefore
- moveToFront / moveToBack
- remove
- popBack / popFront
- insertBefore / inserAfter
- pushFront / pushBack
- findNext
- concat(a:Array)

Arrays (bounded)

- isEmpty
- first / last
- moveAfter / moveBefore
- moveToFront / moveToBack
- remove
- popBack / popFront
- insertBefore / inserAfter
- pushFront / pushBack
- findNext
- concat(a:Array)

Arrays (bounded)

- isEmpty
- first / last
- moveAfter / moveBefore
- moveToFront / moveToBack
- remove
- popBack / popFront
- insertBefore / inserAfter
- pushFront / pushBack
- findNext
- concat(a:Array)

Arrays (unbounded)

- alloziere Array A[1...n]
- pushBack:
  - A voll $\rightarrow$ alloziere $A'[1...2|A|]$ und kopiere A nach A'
  - hänge Element am Ende ein
- popBack:
  - entnehme letztes Element
  - $\frac{1}{4}|A| >= used(A) \rightarrow$ alloziere $A'[1...\frac{1}{2}|A|]$ und kopiere A nach A'
- dynamisch wachsendes Array (verhält sich wie Liste)

Arrays (unbounded)

- alloziere Array A[1...n]
- pushBack:
  - A voll $\rightarrow$ alloziere $A'[1...2|A|]$ und kopiere A nach A'
  - hänge Element am Ende ein
- popBack:
  - entnehme letztes Element
  - $\frac{1}{4}|A| >= used(A) \rightarrow$ alloziere $A'[1...\frac{1}{2}|A|]$ und kopiere A nach A'
- dynamisch wachsendes Array (verhält sich wie Liste)

Arrays (unbounded)

- alloziere Array A[1...n]
- pushBack:
  - A voll $\rightarrow$ alloziere $A'[1...2|A|]$ und kopiere A nach A'
  - hänge Element am Ende ein
- popBack:
  - entnehme letztes Element
  - $\frac{1}{4}|A| >= used(A) \rightarrow$ alloziere $A'[1...\frac{1}{2}|A|]$ und kopiere A nach A'
- dynamisch wachsendes Array (verhält sich wie Liste)

Arrays (unbounded)

- alloziere Array A[1...n]
- pushBack:
  - A voll $\rightarrow$ alloziere $A'[1...2|A|]$ und kopiere A nach A'
  - hänge Element am Ende ein
- popBack:
  - entnehme letztes Element
  - $\frac{1}{4}|A| >= used(A) \rightarrow$ alloziere $A'[1...\frac{1}{2}|A|]$ und kopiere A nach A'
- dynamisch wachsendes Array (verhält sich wie Liste)

Aufgabe

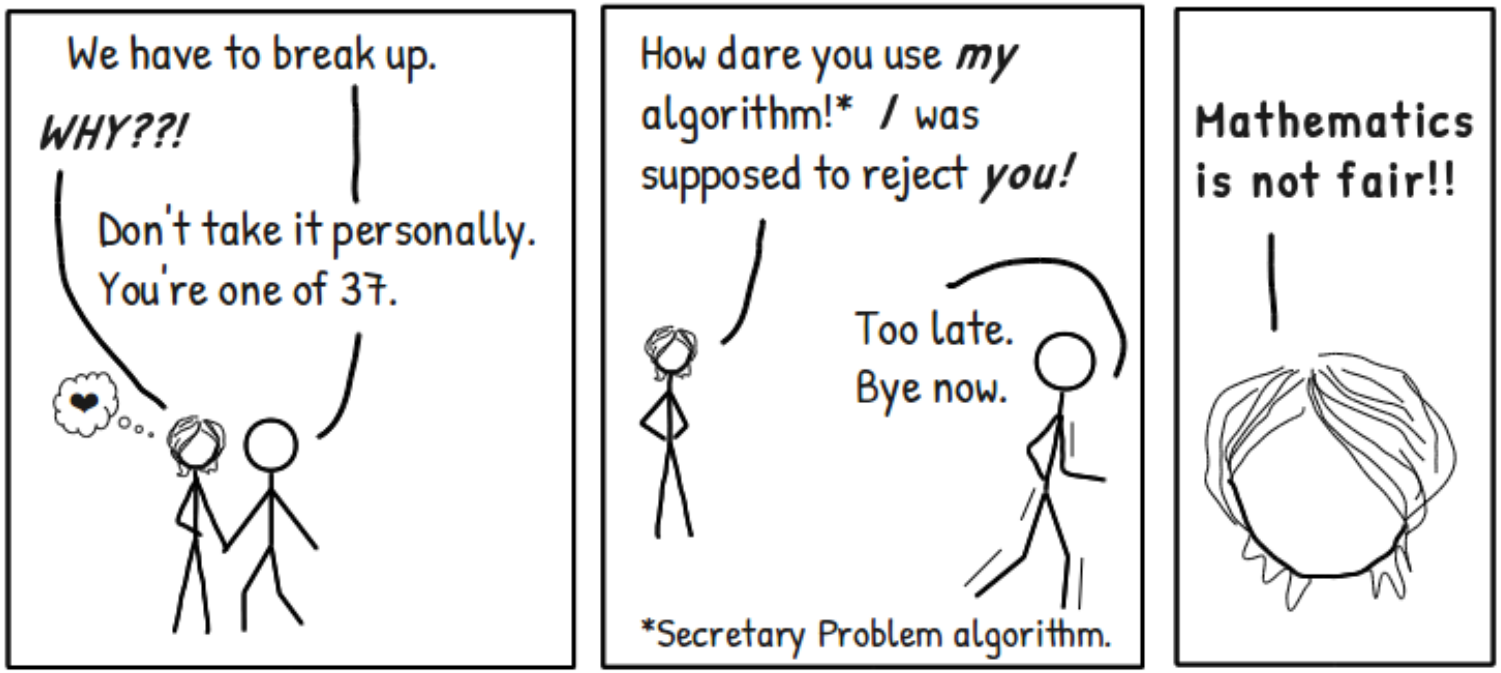
Gegeben: Arrays $A[1...n_1], B[1...n_2] \subseteq N^*$ aufsteigend sortiert
Gesucht: Array $C[1...n](n := n_1 + n_2)$ aufsteigend sortiert

Wie sollte ein Algorithmus aussehen der das Problem löst?

concat sorted arrays

1: **procedure** *merge*(A : Array $[1..n_1]$ of $\mathbb{N}_{\geq 0}$, B : Array $[1..n_2]$ of $\mathbb{N}_{\geq 0}$)
2: $A[n_1 + 1] := \infty$, $B[n_2 + 1] := \infty$
3: $n := n_1 + n_2$
4: $j_A := 1$, $j_B := 1$;
5: **for** $i := 1$ **to** n **do**
6: $C[i] = \min(A[j_A], B[j_B])$
7: **if** $A[j_A] < B[j_B]$ **then**
8: $j_A = j_A + 1$
9: **else**
10: $j_B = j_B + 1$
11: **invariant** $C[1..i]$ enthält genau $A[1..j_A - 1], B[1..j_B - 1]$
12: **invariant** $B[k] \leq A[j_A] \quad \forall k \in \{1..j_B - 1\}$, $A[k] \leq B[j_B] \quad \forall k \in \{1..j_A - 1\}$
13: **invariant** $C[1..i]$ ist sortiert
14: **postcondition** $C[i] \leq C[j] \quad \forall i \leq j, i, j \in \{1, ..., n\}$
15: **return** C

Vielen Dank für eure Aufmerksamkeit!
Bis zum nächsten Mal.



Soulmate Algorithm -- III.