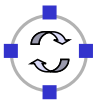
Programmierung und Algorithmen

Kapitel 7
Ausgewählte Algorithmen



Überblick

Suchen in sortierten Folgen

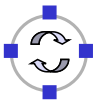
Sortieren

Anhang: Entwurf von Algorithmen per Induktion



Suchen in sortierten Folgen

- Suchen als eine der häufigsten Aufgaben in der Informatik
- Vielzahl von Lösungen für unterschiedlichste Aufgaben
- Hier: Suchen in sortierten Folgen
- Annahmen
  - Folge *F* als Feld von numerischen Werten
  - Zugriff auf *i* -tes Element über *F* [*i*]
  - nur Berücksichtigung des *Suchschlüssels*
- Beispiel: Telefonbuch, Suche nach Namen



Sequenzielle Suche: Algorithmus

```
algorithm SeqSearch (F, k) → p
Eingabe: Folge F der Länge n, Schlüssel k

    for i := 1 to n do
            if F [i] = k then
                    return i
            fi
    od;
    return NO_KEY
```

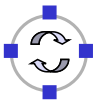
- Der Algorithmus entspricht dem einfachen Induktionsbeweis:
  - Annahme: Es kann bestimmt werden, ob und an welcher Stelle eines Feldes F mit *n-1* Elementen ein gesuchter Wert *k* steht
  - Schritt: Dann kann dieses Problem auch für Felder mit *n* Elementen gelöst werden: Entweder steht *k* im *n*-ten Element oder in den restlichen *n-1*.



Sequenzielle Suche: Aufwand

	Anzahl der Vergleiche
bester Fall	1
schlechtester Fall	*n*
Durchschnitt (erfolgreiche Suche)	(*n* + 1) / 2
Durchschnitt (erfolglose Suche)	*n*

- Der Aufwand im schlechtesten Fall kann auch mit der folgenden Rekurrenz abgeschätzt werden: $T(n) = T(n-1) + 1$
  - Die Lösung dieser Rekurrenz ist einfach und es ist: $T(n) = n$



Binäre Suche

- Voraussetzung: Das zu durchsuchende Feld F ist sortiert.
- Prinzip:
  1. Teste Eintrag an mittlerer Position *m* und prüfe ob gesuchter Wert in der ersten Hälfte (inkl. *m*) oder in der zweiten Hälfte (exkl. *m*) der Liste ist.
  2. Fortsetzung rekursiv in der Hälfte, in der sich der Eintrag befindet.

Induktionsannahme:

- In sortierten (Teil-)Folgen *F* mit maximal *n* Elementen kann festgestellt werden, ob und an welcher Stelle ein zu suchender Wert *k* enthalten ist.

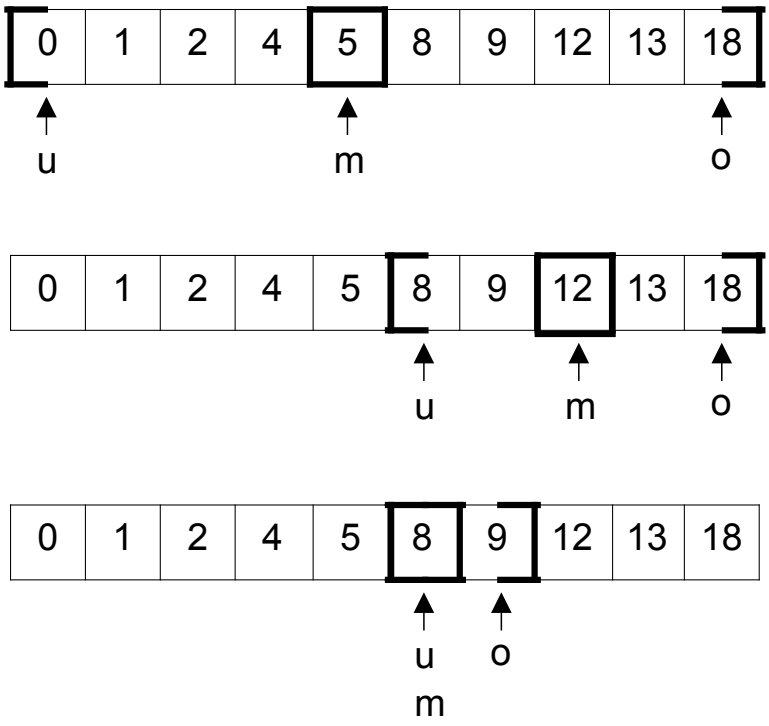
Induktionsanfang: $n = 1$: trivial durch Zugriff und Test an erster Position

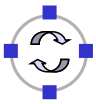
Induktionsschritt: Behauptung gilt für alle $n' < n$:

- Prüfe Element an Stelle m = $\lfloor n/2 \rfloor$ der (Teil-)Folge *F*:
  - Wenn $F(m) = k$ dann ist das Problem gelöst
  - Wenn $F(m) > k$ dann löse das Problem mit der Induktionsbehauptung in der Teilfolge $G = F(1 \ldots m-1)$ // enthält die ersten $m-1 < n$ Elemente
  - Wenn $F(m) < k$ dann löse Problem für $G = F(m+1 \ldots n)$. // $<n$ Elemente



Binäre Suche: Beispiel





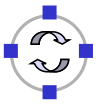
Binäre Suche: Algorithmus (iterativ)

```
algorithm BinarySearch (F, k) → p
Eingabe: Folge F der Länge n, Schlüssel k
    u := 1, o := n;
    while u <= o do
          m := (u + o)/2;
          if F[m] == k then
                return m
          else if k < F[m] then o := m - 1
                else u := m + 1
          fi; fi;
    od;
    return NO_KEY
```



Binäre Suche: Algorithmus (rekursiv)

```
algorithm BinarySearch (F, k) → p
Eingabe: Folge F der Länge n, Schlüssel k

return BinarySearchRec (F, k, 1, n);

algorithm BinarySearchRec (F, k, u, o) → p
Eingabe: Folge F , Schlüssel k,
         untere und obere Schranken u und o

??? /* your task :o) */
```



Binäre Suche: Aufwandsanalyse

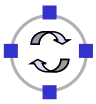
- Mit scharfem Nachdenken:
  - nach dem ersten Teilen der Folge: noch *n*/2 Elemente
  - nach dem zweiten Schritt: *n*/4 Elemente
  - nach dem dritten Schritt: *n*/8 Elemente
  - . . .
  - allgemein: im *i* -ten Schritt max. $n/2^i$ Elemente ⇝ $\log_2 n$

- Mit Hilfe einer Rekurrenz und Master-Theorem:
  - $T(n) = T(n/2) + 1$
  - $a = 1; b = 2; f(n) = 1$
  - Da $n^{\log_b a} = n^{\log_2 1} = n^0 = 1$, gilt $f(n) = \Theta(n^{\log_b a}) = \Theta(1)$
    und es kommt Fall 2 des Master-Theorems zur Anwendung
  - Somit gilt: $T(n) = \Theta(n^{\log_2 1} \log n) = \Theta(\log n)$



Binäre Suche: Aufwand

	Anzahl der Schritte
bester Fall	1
schlechtester Fall	$\approx \log_2 n$
Durchschnitt (erfolgreiche Suche)	$\approx \log_2 n$
Durchschnitt (erfolglose Suche)	$\approx \log_2 n$



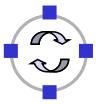
Vergleich der Suchverfahren

Verfahren	10	10^2	10^3	10^4
sequenziell ($n/2$) binär ($\log_2$ n)	≈ 5 ≈ 3.3	≈ 50 ≈ 6.6	≈ 500 ≈ 9.9	≈ 5000 ≈ 13.3



Sortieren

- Grundlegendes Problem in der Informatik
- Aufgabe:
  - Ordnen von Dateien mit *Datensätzen*, die *Schlüssel* enthalten
  - Umordnen der Datensätze, so dass klar definierte Ordnung der Schlüssel (numerisch/alphabetisch) besteht
- Vereinfachung:
  - Nur Betrachtung der Schlüssel, z.B. Feld von `int`-Werten



Sortieren: Grundbegriffe

- **Verfahren**
  - Intern: in Hauptspeicherstrukturen (Felder, Listen)
  - Extern: Datensätze auf externen Medien (Festplatte, Magnetband)



Eine zentrale Eigenschaft von Sortierverfahren: Stabilität

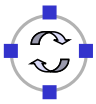
Ein Sortierverfahren heißt *stabil*, wenn es die relative Reihenfolge gleicher Schlüssel in der Datei beibehält.

- Beispiel: alphabetisch geordnete Liste von Personen soll nach Alter sortiert werden → Personen mit gleichem Alter weiterhin alphabetisch geordnet

Name	Alter
Endig, Martin	30
Geist, Ingolf	28
Höpfner, Hagen	24
Schallehn, Eike	28

Sortieren

Name	Alter
Höpfner, Hagen	24
Geist, Ingolf	28
Schallehn, Eike	28
Endig, Martin	30



Sortieren durch Einfügen: Prinzip

- **Idee** (und korrespondierender Induktionsbeweis)
  - Umsetzung der typischen (?) menschlichen Vorgehensweise, etwa beim Sortieren eines Stapels von Karten:
    1. Starte mit der ersten Karte einen neuen Stapel
    2. Nimm jeweils nächste Karte des Originalstapels: füge diese an der richtigen Stelle in den neuen Stapel ein

Induktionsannahme: Felder F der Länge *n-1* können sortiert werden.

Induktionsanfang: *n* = *1*: trivial

Induktionsschritt: *n-1* ⇝ *n* :

- Entnehme dem (Teil-)Feld *F* das letzte Element
- Sortiere das verbleibende Teilfeld mit n-1 Elementen nach Induktionsannahme
- Füge das im ersten Schritt entnommene Element in das sortierte Teilfeld an der richtigen Stelle ein

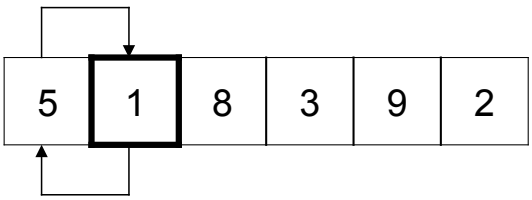


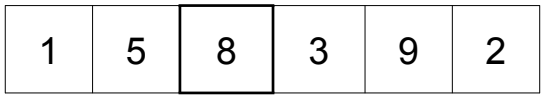
Sortieren durch Einfügen: Algorithmus

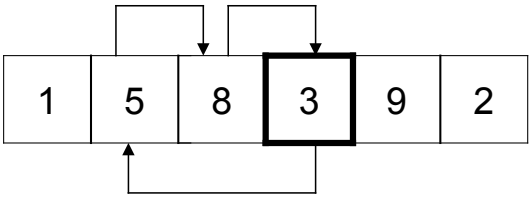
```
algorithm InsertionSort (F)
Eingabe: zu sortierende Folge F der Länge n
   for i := 2 to n do
                m := F[i ];   /* zu merkendes Element */
                j := i ;
                while j > 1 do
                  if F[j - 1]  >= m then
                                /* Verschiebe F[j - 1] nach rechts */
                                F[j ] := F[j - 1]; j := j - 1
                  else
                                /*  Verlasse innere Schleife  */
                  fi;
                od
        F[j ] := m;   /* Einfügeposition */
   od
```

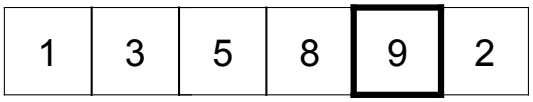


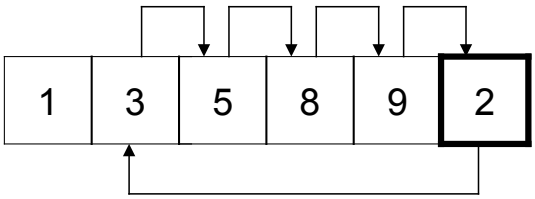
Sortieren durch Einfügen: Beispiel

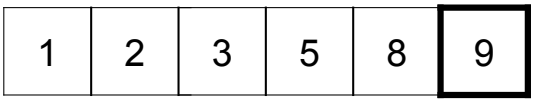


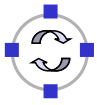






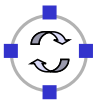






Analyse: InsertionSort

- Aufwand:
  - Anzahl der Vertauschungen
  - Anzahl der Vergleiche
  - Vergleiche **dominieren** Vertauschungen, d. h. es werden (wesentlich) mehr Vergleiche als Vertauschungen benötigt
- Außerdem Unterscheidung:
  - Bester Fall: Liste ist schon sortiert
  - Mittlerer (zu erwartender) Fall: Liste ist unsortiert
  - Schlechtester Fall: z. B. Liste ist absteigend sortiert



Analyse: InsertionSort /2

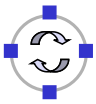
- Wir müssen in jedem Fall alle Elemente $i = 2$ bis n durchgehen, d. h. immer Faktor $n - 1$
- Dann müssen wir zur korrekten Einfügeposition zurückgehen
- Bester Fall: Liste sortiert
  → Einfügeposition ist gleich nach einem Schritt an Position $i - 1$
  - bei jedem Rückweg Faktor 1
  - Gesamtanzahl der Vergleiche:
    $(n - 1) * 1 = n - 1$
  - für Große Listen abgeschätzt als $n - 1 \approx n$
  - ”linearer Aufwand”



Analyse: InsertionSort /3

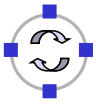
- Mittlerer (zu erwartender) Fall: Liste unsortiert
  → Einfügeposition wahrscheinlich auf der Hälfte des Rückwegs
  - bei jedem der *n* - 1 Rückwege Faktor *(i* - 1)/2
  - Gesamtanzahl der Vergleiche:

$$
\begin{aligned}
& (n-1)/2 + (n-2)/2 + (n-3)/2 + \ldots + 2/2 + 1/2 \\
= \; & \frac{(n-1) + (n-2) + (n-3) + \ldots + 2 + 1}{2} \\
= \; & \frac{1}{2} \cdot \frac{n \cdot (n-1)}{2} \\
= \; & \frac{n \cdot (n-1)}{4} \\
\approx \; & \frac{n^2}{4}
\end{aligned}
$$



Analyse: InsertionSort /4

- Schlechtester Fall: z. B. Liste umgekehrt sortiert
  → Einfügeposition am Ende des Rückwegs bei Position 1
  - bei jedem der *n* - 1 Rückwege Faktor *i* - 1
  - analog zu vorhergehenden Überlegungen, bloß doppelte Rückweglänge
  - Gesamtanzahl der Vergleiche:
    $\frac{n \cdot (n-1)}{2} \approx \frac{n^2}{2}$
- Letzte beide Fälle: ”quadratischer Aufwand”, wenn konstante Faktoren nicht berücksichtigt werden
- Rekurrenz für schlechtesten Fall:
  - T(n) = T(n-1) + n - 1; T(1) = 1
  - Die Lösung führt auf die bekannte Summenformel $\sum_{i=1}^{n-1} i = \frac{n \cdot (n-1)}{2} \approx \frac{n^2}{2}$



Sortieren durch Selektion

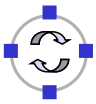
- In manchen Situationen ist das Einfügen an der richtigen Stelle eine „teure Operation“, da das Verschieben der anderen Elemente ggf. aufwändig ist

- **Idee:** Suche jeweils größten Wert, und tausche diesen an die letzte Stelle; fahre dann mit der um 1 kleineren Liste fort.

  Induktionsannahme: Felder F der Länge *n-1* können sortiert werden.

  Induktionsanfang: *n* = *1*: trivial

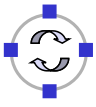
  Induktionsschritt: *n-1* ⇝ *n* :
  - Suche und entnehme das größte Element im (Teil-)Feld *F*
  - Sortiere das verbleibende Teilfeld mit n-1 Elementen nach Induktionsannahme
  - Füge das im ersten Schritt entnommene Element in das sortierte Teilfeld hinter der letzten Stelle ein



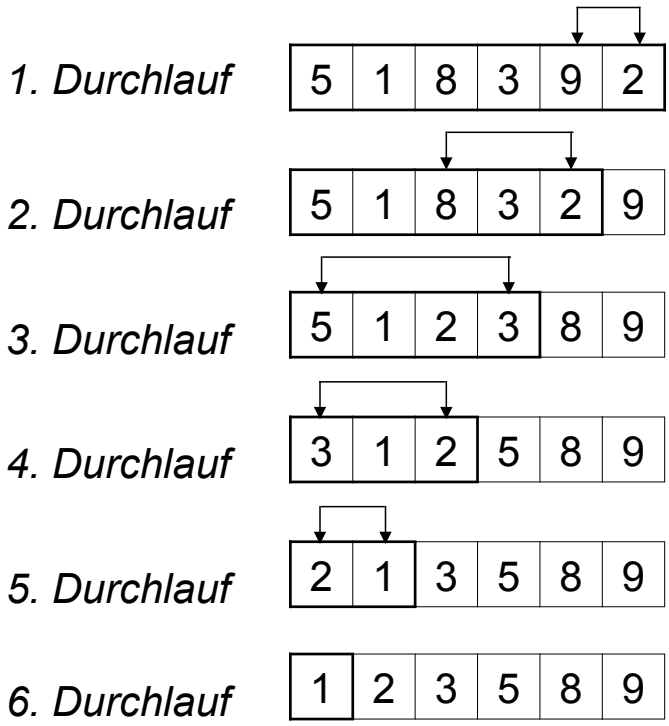
Sortieren durch Selektion (2)

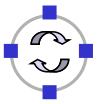
```
algorithm SelectionSort (F)
 Eingabe: zu sortierende Folge F der Länge n

  p := n;
  while p > 0 do
        g := Index des größten Elementes aus F
             im Bereich 1...p;
        Vertausche Werte von F[p] und F[g];
        p := p - 1
  od
```

- Hinweis: Kommt das aktuell „größte“ Element mehrere Male in der Folge vor, dann wähle das jeweils erste Auftreten, um ein „stabiles Sortierverfahren“ zu erhalten



Sortieren durch Selektion: Beispiel





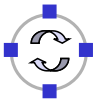
Analyse: SelectionSort

- In jedem Durchlauf das Element von *F*[*p*] mit dem größten Element tauschen
- Variable *p* läuft von *n* . . . 1
  ⇝ daher *n* Vertauschungen
- In jedem Durchlauf das größte Element aus 1 . . . *p* ermitteln
  ⇝ *p* - 1 Vergleiche

$$(n-1)+(n-2)+(n-3)+\ldots+2+1=\frac{n(n-1)}{2}\approx\frac{n^2}{2}$$

- **Anzahl Vergleiche identisch für besten, mittleren und schlechtesten Fall!**

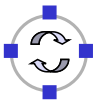
- Rekurrenz: T(n) = T(n-1) + n - 1



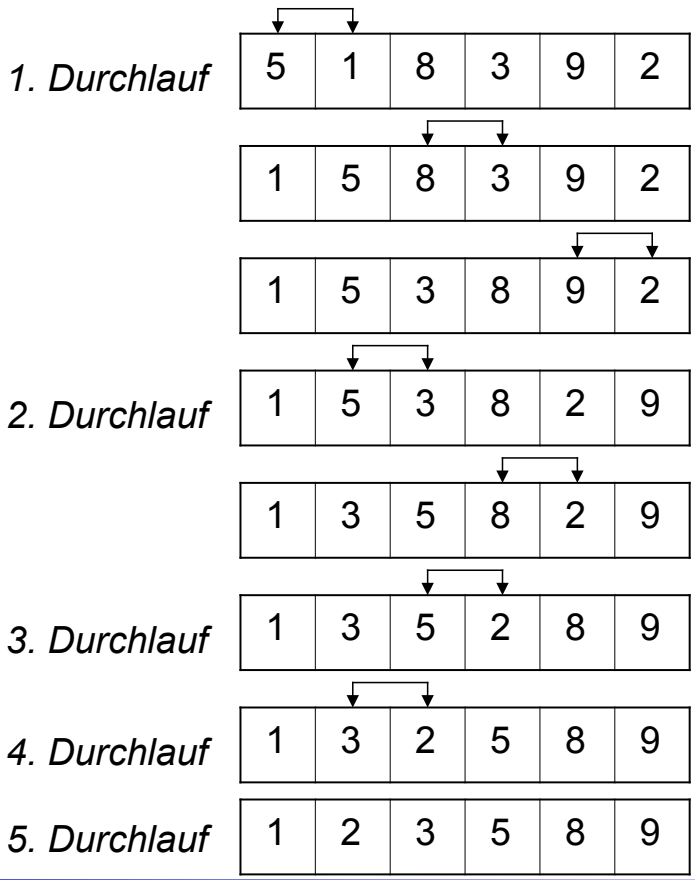
BubbleSort

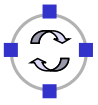
- **Idee:** Verschieden große aufsteigende Blasen („Bubbles“) in einer Flüssigkeit sortieren sich quasi von allein, da größere Blasen die kleineren „überholen“.

```
algorithm BubbleSort (F)
  Eingabe: zu sortierende Folge F der Länge n
  do
      for i := 1 to n - 1 do
          if F[i] > F[i + 1] then
            Vertausche Werte von F[i] und F[i + 1]
          fi
      od
  until keine Vertauschung mehr aufgetreten
```



BubbleSort: Beispiel

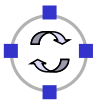




BubbleSort: Optimierung

- Größte Zahl rutscht in jedem Durchlauf automatisch an das Ende der Liste
- Im Durchlauf *j* reicht die Untersuchung bis Position *n* - *j*

- Für BubbleSort ist die Angabe eines entsprechenden Induktionsbeweises übrigens nicht so einfach, da die Induktionsannahme hierfür nicht sehr intuitiv ist.
  - Es wird auch gar nicht behauptet, dass mit vollständiger Induktion alle Algorithmen einfach entwickelt werden können.



Analyse: BubbleSort

- Bester Fall: *n*
- Durchschnittlicher Fall
  normal: n^2
  optimiert: $\frac{n^2}{2}$

- Schlechtester Fall
  normal: n^2
  optimiert: $\frac{n^2}{2}$



MergeSort: Prinzip

- **Idee:**
  1. Teile die zu sortierende Liste in zwei Teillisten
  2. Sortiere diese (rekursives Verfahren!)
  3. Mische die Ergebnisse

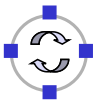
  Induktionsannahme: Felder F der Länge *n/2* können sortiert werden.

  Induktionsanfang: *n = 1*: trivial

  Induktionsschritt: *n/2 ⇝ n* :
  - Teile das (Teil-)Feld *F* der Länge *n* in zwei gleich große Hälften
  - Sortiere beide Teilfelder mit *n/2* Elementen nach Induktionsannahme
  - Füge die beiden sortierten Teilfelder zu einem Feld zusammen, in dem jeweils das kleinste Element der beiden Teilfelder entfernt und in das zusammengefügte Feld aufgenommen wird (erfordert *n* Schritte)
- Rekurrenz: $T(n) = 2\,T(n/2) + n;$ also $a = 2;\ b = 2;\ f(n) = n;$

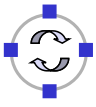
  $\Rightarrow\ n^{\log_b a} = n^{\log_2 2} = n^1 = n\ \Rightarrow\ f(n) = \Theta(n^{\log_b a}) = \Theta(n)$

  $\Rightarrow\ T(n) = \Theta(n^{\log_2 2}\log n) = \Theta(n \log n)$



Mischen von zwei Folgen: Algorithmus

```
procedure Merge (F1, F2) → F
Eingabe: zwei zu sortierende Folgen F1, F2
Ausgabe: eine sortierte Folge F

  F := leere Folge;
  while F1 und F2 nicht leer do
      Entferne das kleinere der Anfangselemente
      aus F1 bzw. F2;
      Füge dieses Element an F an;
  od;
  Füge die verbliebene nichtleere Folge F1
  oder F2 an F an;
  return F;
```

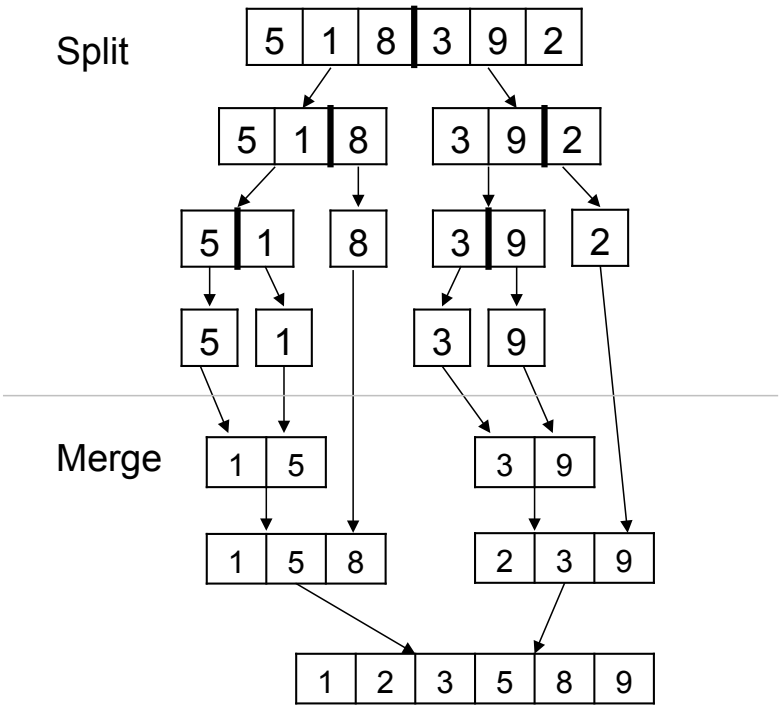


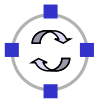
Sortieren durch Mischen: Algorithmus

```
algorithm MergeSort (F) → F_S
Eingabe: eine zu sortierende Folge F
Ausgabe: eine sortierte Folge F_S

  if F einelementig then
            return F;
  else
            Teile F in F_1 und F_2;
            F_1 := MergeSort (F_1);
            F_2 := MergeSort (F_2);
            return Merge (F_1, F_2);
  fi
```



MergeSort: Beispiel





Analyse (ausführlich): MergeSort

- Folge der Länge *n*, Anzahl der Vergleiche = V_n

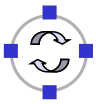
$$V_n = 2V_{n/2} + n \qquad \text{für } n \geq 2 \text{ mit } V_1 = 0$$

Jede Teilfolge sortieren (Größe jeweils n/2), Mischen (*n* Vergleiche)

- Annahme: $n = 2^N$ (*n* ist Zweierpotenz)

$$V_{2^N} = 2 \cdot V_{2^{N-1}} + 2^N$$

$$\Leftrightarrow \quad \frac{V_{2^N}}{2^N} = \frac{V_{2^{N-1}}}{2^{N-1}} + 1$$



Analyse (ausführlich): MergeSort /2

- Sukzessive $\frac{V_{2^{N-i}}}{2^{N-i}}$ ersetzen bis $i = N$

$$\frac{V_{2^N}}{2^N} = N$$

- *Aus* $2^N = n$ *folgt* $N = \log n$:

$$V_n = n \log n$$



QuickSort: Prinzip

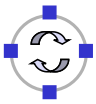
- **Idee:**
  - Ähnlich wie MergeSort durch rekursive Aufteilung
  - Vermeidung des Mischvorgangs durch Aufteilung der Teillisten in zwei Hälften bezüglich eines Pivot-Elementes, wobei
    - In einer Liste alle Elemente größer als das Pivot-Element sind
    - In der anderen Liste alle Elemente kleiner sind

Induktionsannahme: Felder F der Länge < *n* können sortiert werden.

Induktionsanfang: *n* = *1*: trivial

Induktionsschritt: < *n* ⇝ *n* :

- Wähle ein beliebiges Pivot-Element
- Verschiebe nun alle Elemente, die kleiner als das Pivot-Element sind, in ein erstes Teilfeld *L* und alle Elemente, die größer als das Pivot-Element sind, in ein zweites Teilfeld *R*
- Sortiere beide Teilfelder mit < n Elementen nach Induktionsannahme
- Füge die sortierten Teilfelder zusammen: *L'* + *Pivot-Element* + *R'*



QuickSort: Algorithmus

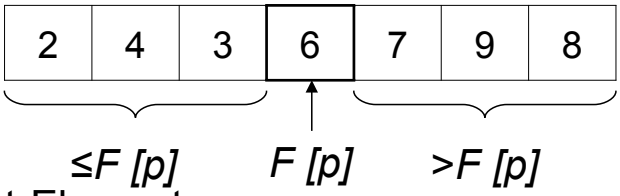
```
algorithm QuickSort (F, left, right)
  Eingabe: eine zu sortierende Folge F,
           die linke und rechte Grenze left,right
  if left < right then
     middle = Partition (F, left, right);
     QuickSort (F, left, middle - 1);
     QuickSort (F, middle + 1, right);
  fi
```

(Quelle: Udi Manber. Introduction to Algorithms. pp. 134-136, Addison-Wesley, 1989.)

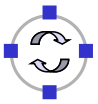


QuickSort: Vertauschen von Elementen

- **Pivot-Element** *p*
  - Folge von links durchsuchen, bis Element gefunden, das größer *p* ist
  - Folge von rechts durchsuchen, bis Element gefunden, das kleiner oder gleich *p* ist
- Elemente ggf. tauschen

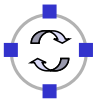


- Auswahl des Pivot-Elements *p*:
  - Kann beliebig gewählt werden, z.B. das erste Element der zu sortierenden Teilfolge
  - Aber: “ungünstige” Wahl des Pivotelements führt zu längerer Laufzeit



QuickSort: Zerlegen beim QuickSort

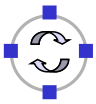
```
algorithm Partition (F, left, right)
  Eingabe: Folge F, linke/rechte Grenze left, right
  Ausgabe: Position middle der Mitte der Zerlegung
           F modifiziert, so dass:
             ∀ i ≤ middle: F[i] ≤ F[middle]
             ∀ j > middle: F[j] > F[middle]
  p := F[left]; l := left; r := right;
  while l < r do
     while l ≤ right and F[l] ≤ p do l := l + 1; od
     while r ≥ left  and F[r] > p do r := r - 1; od
     if l < r then Exchange F[l] with F[r]; fi
  od
  middle := r;
  Exchange F[left] with F[middle];
```



Beispielablauf für Partition

- Im folgenden Beispiel wird das erste Element (6) als Pivot gewählt
- Die umkreisten Elemente sind nach einem Schleifendurchlauf vertauscht worden

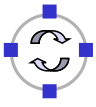
6	2	8	5	10	9	12	1	15	7	3	13	4	11	16	14
6	2	(4)	5	10	9	12	1	15	7	3	13	(8)	11	16	14
6	2	4	5	(3)	9	12	1	15	7	(10)	13	8	11	16	14
6	2	4	5	3	(1)	12	(9)	15	7	10	13	8	11	16	14
(1)	2	4	5	3	(6)	12	9	15	7	10	13	8	11	16	14



Beispielablauf für Quicksort

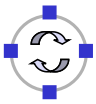
- Pivot-Elemente werden umkreist; steht ein einzelnes Element zwischen zwei Pivot-Elementen, muss es sich an der „richtigen“ Position der Folge befinden

6	2	8	5	10	9	12	1	15	7	3	13	4	11	16	14
1	2	4	5	3	(6)	12	9	15	7	10	13	8	11	16	14
(1)	2	4	5	3	(6)	12	9	15	7	10	13	8	11	16	14
(1)	(2)	4	5	3	(6)	12	9	15	7	10	13	8	11	16	14
(1)	(2)	3	(4)	5	(6)	12	9	15	7	10	13	8	11	16	14
(1)	(2)	3	(4)	5	(6)	8	9	11	7	10	(12)	13	15	16	14
(1)	(2)	3	(4)	5	(6)	7	(8)	11	9	10	(12)	13	15	16	14
(1)	(2)	3	(4)	5	(6)	7	(8)	10	9	(11)	(12)	13	15	16	14
(1)	(2)	3	(4)	5	(6)	7	(8)	9	(10)	(11)	(12)	13	15	16	14
(1)	(2)	3	(4)	5	(6)	7	(8)	9	(10)	(11)	(12)	(13)	15	16	14
(1)	(2)	3	(4)	5	(6)	7	(8)	9	(10)	(11)	(12)	(13)	14	(15)	16



Analyse: QuickSort

- Aufwandsabschätzung komplizierter (daher nicht hier)
  - Bester Fall: $n \log n$
    - Relativ einfach, da Rekurrenz: $T(n) = 2\,T(n/2) + n$ entsteht
  - Durchschnittlicher Fall: $n \log n$
    - Das ist der schwierige Teil der Analyse
  - Schlechtester Fall: n^2
    - Rekurrenz ist in diesem Fall: $T(n) = T(n-1) + n$
    - Tritt bei Wahl des Pivot-Elements ganz links dann auf, wenn die Folge schon sortiert ist (ungünstig!)
    - Die Wahrscheinlichkeit, dass dieser Fall auftritt, kann durch zufällige Wahl eines Elements der Teilfolge und Vertauschen mit dem ganz linken Element vor Partition gesenkt werden
- Außerdem: Im Gegensatz zur MergeSort ist QuickSort durch Vorgehensweise bei Vertauschungen **instabil**



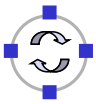
Sortierverfahren im Vergleich

Verfahren	Stabilität	Vergleiche im Mittel
SelectionSort	instabil	$\approx n^2/2$
InsertionSort	stabil	$\approx n^2/4$
BubbleSort	stabil	$\approx n^2/2$
MergeSort	stabil	$\approx n \log n$
QuickSort	instabil	$\approx n \log n$



Zusammenfassung zu Suchen und Sortieren

- Suchen und Sortieren sind Grundaufgaben in der Informatik
- Basisalgorithmen:
  - Suchen: Sequentielle Suche, Binärsuche
  - Sortieren: Sortieren durch Einfügen, Suchen durch Selektion, BubbleSort, Suchen durch Mischen (MergeSort), QuickSort
- Algorithmenmuster „Teile und Herrsche“
- Relation zwischen Algorithmen und Induktionsbeweisen
- Abschätzung des Aufwands (mit/ohne Rekurrenzen)
- Literatur: Saake/Sattler: *Algorithmen und Datenstrukture*n, Kapitel 5



Anhang: Entwurf von Algorithmen per Induktion

„Nichts ist wichtiger als die Quellen von Erfindungen zu betrachten, die meiner Meinung nach interessanter sind als die Erfindungen selbst.“

(G. W. Leibniz, 1646-1716)

- Mit den folgenden Beispielen soll aufgezeigt werden, wie vollständige Induktion beim Entwurf von Algorithmen eingesetzt werden kann
- Die Grundidee hierbei ist, die Rechenschritte, die zur Lösung eines Problems erforderlich sind, nicht „alle auf einmal” anzugeben, sondern zu zeigen, dass:
  - ein „kleines Problem“ gelöst werden kann (Basisfall) und
  - jedes Problem durch Lösen und Kombinieren „kleinerer Probleme“ gelöst werden kann (Induktionsschritt).

(Quelle: Udi Manber. Introduction to Algorithms. Kapitel 5 & 6, Addison-Wesley, 1989.)



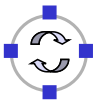
Auswerten eines Polynoms an einem Punkt (1)

- Gegeben: Koeffizienten $a_n, a_{n-1}, ..., a_1, a_0$ und eine Zahl x aus |R
- Aufgabe: Berechne den Wert des Polynoms $P_n(x) = \sum_{i=0}^{n} a_i \cdot x^i$
- Erster Ansatz:
  - Das Problem umfasst ein Polynom mit $n+1$ Koeffizienten
  - Als erste Idee probieren wir, den Koeffizienten a_n wegzulassen

  Induktionshypothese 1: Ein Polynom mit Koeffizienten a_0 bis a_{n-1} kann an jeder beliebigen Stelle x ausgewertet werden, d.h. wir wissen wie $P_{n-1}(x)$ berechnet wird

  Induktionsanfang: n = 1: $P_0(x) = a_0$

  Induktionsschritt: $n-1 \rightsquigarrow n$:
  - Da $P_{n-1}(x) + a_n \cdot x^n = P_n(x)$ müssen wir lediglich x^n berechnen, mit a_n multiplizieren und zu $P_{n-1}(x)$ addieren
  - Da dieses Vorgehen offensichtlich ist, scheint uns der Ansatz mit vollständiger Induktion hier keinen Nutzen zu bringen... :o(



Auswerten eines Polynoms an einem Punkt (2)

- Analyse:
  - Die Anzahl der Multiplikationen des Algorithmus ist: $T(n) = T(n-1) + n$
  - Somit fallen $\frac{n \cdot (n+1)}{2} \approx \frac{n^2}{2}$ Multiplikationen an.
  - Zusätzlich werden n Additionen benötigt.
  - Viele der Multiplikationen sind redundant: Bevor x^n berechnet wird, wurde schon x^{n-1} berechnet.
- Zweiter Ansatz:

  Induktionshypothese 2: Wir wissen wie $P_{n-1}(x)$ berechnet wird und wie x^{n-1} berechnet wird

  Induktionsanfang: n = 1: $P_0(x) = a_0$; $x^0 = 1$

  Induktionsschritt: $n-1 \rightsquigarrow n$:
  - Da $P_{n-1}(x) + a_n \cdot x^n = P_n(x)$ und $x^n = x^{n-1} \cdot x$ berechnen wir x^n mit lediglich einer zusätzlichen Multiplikation, multiplizieren mit a_n und addieren das Ergebnis zu $P_{n-1}(x)$. Weiterhin merken wir uns x^n.



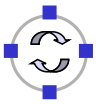
Auswerten eines Polynoms an einem Punkt (3)

- Analyse:
  - Die Anzahl der Multiplikationen des Algorithmus ist: *T(n) = T(n-1) + 2*
  - Es fallen somit *2·n* Multiplikationen und *n* Additionen an.
  - Obwohl unsere Induktionshypothese stärker ist (wir berechnen nun zwei Zwischenergebnisse), führt sie zu einem effizienteren Algorithmus.
- Dritter Ansatz:
  - Wir haben bisher a_n entfernt; das ist jedoch nicht die einzige Möglichkeit, da wir ebenso a_0 entfernen können.
  - Wir berechnen also das Polynom: $P'_{k,n}(x) = \sum_{i=k}^{n} a_i \cdot x^{i-k}$, wobei die niedrigsten k Koeffizienten entfernt werden
  - Induktion verläuft nun rückwärts über k (n ist fest)
  - Induktionshypothese 3: Wir wissen wie $P'_{k,n}(x)$ berechnet wird

    Induktionsanfang: $\mathrm{k} = n: P'_{n,n}(x) = \sum_{i=n}^{n} a_i \cdot x^{i-n} = a_n \cdot x^0 = a_n$

    Induktionsschritt: $\mathrm{k} \to k-1$:

$$P'_{k-1,n}(x) = \sum_{i=k-1}^{n} a_i \cdot x^{i-(k-1)} = a_{k-1} + x \cdot \sum_{i=k}^{n} a_i \cdot x^{i-k} = a_{k-1} + x \cdot P'_{k,n}(x)$$

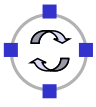


Auswerten eines Polynoms an einem Punkt (4)

- Analyse:
  - Es gilt: $P'_{0,n}(x) = \sum_{i=0}^{n} a_i \cdot x^{i-0} = P_n(x)$
  - Die Anzahl der Multiplikationen ist in diesem Fall: T(n) = T(n-1) + 1
  - Es fallen somit *n* Multiplikationen und *n* Additionen an.

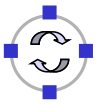
```
algorithm Polynomial (double a[], x; int n)
  Eingabe: ein Feld a[0, n] mit n+1 Koeffizienten,
           der Punkt x
  Ausgabe: p, der Wert des Polynoms P_n am Punkt x

  p = a[n];
  for i := 1 to n do p := x · p + a[n-i]; od
```

- Dieser Algorithmus ist unter dem Namen *Horner-Schema* bekannt (benannt nach dem englischen Mathematiker G.W. Horner)
- Er ist nicht nur effizienter, sondern sein Programmtext ist auch einfacher!



Reflexion

- Induktion erlaubt es, uns darauf zu konzentrieren, Lösungen für kleinere Probleme zu Lösungen für größere Probleme zu erweitern
- Sei also *P(n)* ein Problem das vom Parameter *n* abhängt (oft ist *n* die Größe des Problems, also z.B. Anzahl Elemente etc.)
  - Wir starten mit einer beliebigen Instanz P(n) und versuchen es dann unter der Annahme zu lösen, dass wir eine Lösung von P(n-1) kennen.
  - Hierbei gibt es viele Möglichkeiten, die Induktionshypothese zu wählen, und auch viele Möglichkeiten, diese Hypothese einzusetzen.
    - Bei der Polynomauswertung konnte man beispielsweise *von rechts nach links* oder *von links nach rechts* rechnen.
    - Manchmal hat man die Wahl zwischen *Top-Down* und *Bottom-Up.*
    - Manchmal kann man in jedem Schritt um 2 (oder mehr) inkrementieren.
    - Es kann auch sein, dass die beste Reihenfolge für unterschiedliche Eingaben unterschiedlich ist, so dass es sich ggf. lohnt, einen Algorithmus zu entwerfen, der die beste Reduktion berechnet.



Maximal Induzierter Subgraph (1)

- Angenommen, Sie möchten eine Party organisieren:
  - Sie haben eine Liste von Freunden, die Sie gerne einladen würden.
  - Sie wollen aber, dass jeder von Ihnen eingeladene Freund mindestens *k* weitere Bekannte (außer Ihnen) trifft, damit er sich nicht langweilen muss.
  - Hierfür wissen Sie, welche Ihrer Freunde sich untereinander kennen.
  - Ihre Party soll so groß wie möglich werden! :o)
- Diese Aufgabe entspricht dem folgenden graphentheoretischen Problem:

Problem *Maximal Induzierter Subgraph*

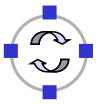
Gegeben: ein ungerichteter Graph $G = (V, E)$ und eine ganze Zahl k

Gesucht: ein induzierter Subgraph $H = (U, F)$ von G maximaler Größe, so dass alle Knoten in H einen Grad $\geq k$ haben, oder es soll ermittelt werden, dass kein solcher Graph existiert.



Maximal Induzierter Subgraph (2)

- Eine erste Idee ist es, alle Knoten aus G zu entfernen, die einen geringeren Grad als *k* haben:
  - Hierdurch entsteht ein reduzierter Graph, in dem unter Umständen wiederum Knoten mit einem Grad < k enthalten sind.
  - Es ist allerdings noch nicht klar, in welcher Reihenfolge die Knoten entfernt werden sollten:
    - Sollten wir erst alle Knoten mit Grad < k entfernen und dann erneut die Knoten betrachten, deren Grad reduziert wurde?
    - Oder besser erst einen Knoten mit Grad < k entfernen und dann die mit diesem Knoten verbundenen Knoten betrachten?
    - Führen beide Ansätze zum gleichen Ergebnis?
    - Ist der resultierende Subgraph von maximaler Größe?
  - Diese Fragen können einfach beantwortet werden. Die im Folgenden beschriebene Argumentation macht die Beantwortung noch leichter.



Maximal Induzierter Subgraph (3)

- Wir haben (mindestens) die beiden folgenden Möglichkeiten über das Problem nachzudenken:
  - *Welche Anweisungsfolge muss ein Computer abarbeiten, um das Problem zu lösen?*
  - *Wie kann bewiesen werden, dass ein Algorithmus für das Problem überhaupt existiert?*
- Obwohl die zweite Frage auf den ersten Blick komplizierter aussieht, ist Ihre Betrachtung oft einfacher.
  - Natürlich werden wir als Beweistechnik vollständige Induktion einsetzen. :o)

Induktionshypothese: Wir wissen wie der maximale induzierte Subgraph H in Graphen G mit < n Knoten gefunden werden kann (so dass alle Knoten einen Grad ≥ *k* haben)

Induktionsanfang: $n \leq k$: $H := (\emptyset, \emptyset)$, da alle Knoten einen Grad < k haben
$n > k$: Haben alle Knoten in G Grad ≥ $k \Rightarrow H := G$;

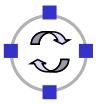


Maximal Induzierter Subgraph (4)

Induktionsschritt: $n\text{-}1 \rightsquigarrow n$:

- Angenommen, G ist ein Graph mit $n > k$ Knoten
- Wenn alle Knoten in G einen Grad ≥ k haben (das ist einfach zu prüfen), sind wir fertig.
- Wenn nicht, dann gibt es mindestens einen Knoten mit Grad < k
- Dieser kann nicht zu dem gesuchten Subgraph H gehören und muss daher aus unserem zu untersuchenden Graph entfernt werden (einschließlich aller seiner Kanten).
- Nachdem dieser Knoten entfernt wurde, hat der zu untersuchende Graph n-1 Knoten und wir wissen, wie dieses Problem gelöst werden kann.

- Der Beweis hat die zuvor gestellten Fragen auf einfache Weise beantwortet:
  - Jeder beliebige Knoten mit Grad < k kann (und muss) entfernt werden.
  - Die Reihenfolge ist dabei unerheblich, da seine Entfernung zwingend erforderlich ist.
  - Der hieraus ablesbare Algorithmus ist korrekt per Konstruktion. :o)



Suchen Maximaler Injektiver Abbildungen (1)

- Sei f eine Funktion, die eine endliche Menge A in sich selbst abbildet
  - Der Einfachheit halber, benennen wir die Elemente von A mit den ganzen Zahlen 1 bis n.
  - Die Funktion f sein durch ein Feld $f[1...n]$ repräsentiert.
  - Die Funktion f heißt injektiv, wenn gilt:

$$\forall\ a, b \in A: a \neq b \Rightarrow f(a) \neq f(b)$$

Problem *Finde Maximale Injektive Abbildung*

Gegeben: Eine Menge A und eine Abbildung $f : A \rightarrow A$

Gesucht: Eine Teilmenge $S \subseteq A$ so, dass:

1.) $\forall\ a \in S: f(a) \in S$

2.) $\forall\ a, b \in S: a \neq b \Rightarrow f(a) \neq f(b)$

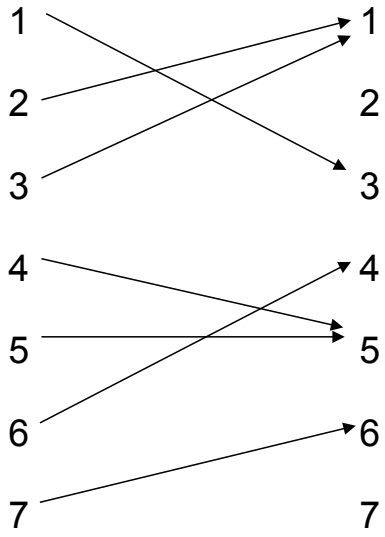
3.) S ist die größte solche Teilmenge von A

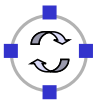


Suchen Maximaler Injektiver Abbildungen (2)

- Ist f eine injektive Funktion, so erfüllt A die Bedingung des Problems.
- Gilt $f[i] = f[j]$ für beliebige $i \neq j$, so kann S nicht beide Elemente i und j enthalten:
- Im nebenstehenden Beispiel ist etwa $f[2] = f[3]$
- Die abgebildete Funktion ist demnach nicht injektiv
- Es ist jedoch nicht beliebig, welche Elemente entfernt werden müssen, damit S maximal wird.
- In diesem Fall ist die gesuchte maximale Teilmenge:

$S = \{1, 3, 5\}$





Suchen Maximaler Injektiver Abbildungen (3)

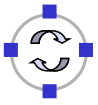
- Glücklicherweise können wir selber entscheiden, wie wir ein Problem der Größe *n* auf ein kleineres Problem reduzieren:
  - Wir können ein Element aus *A* entnehmen, dass in *S* liegt.
  - Wir können ein Element aus *A* entnehmen, dass nicht in *S* liegt. (Wir entscheiden uns für die zweite Variante.)

Induktionshypothese: Wir wissen wie das Problem für Mengen mit maximal n-1 Elementen gelöst werden kann.

Induktionsanfang: A injektiv: S := A (nichts zu entnehmen)

Induktionsschritt: $n\text{-}1 \rightsquigarrow n$:

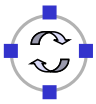
- Angenommen $|A| = n$ und wir suchen ein $S \subseteq A$, dass die Bedingung erfüllt.
- Hilfsbehauptung: $\forall\, a \in A$: $(\forall\, b \in A: f(b) \neq a \Rightarrow a \notin S)$ (ein Element *a*, auf das kein Element $b \in A$ abgebildet wird, kann nicht zu *S* gehören)



Suchen Maximaler Injektiver Abbildungen (4)

Induktionsschritt (Fortsetzung):

- Beweis der Hilfsbehauptung: Angenommen $a \in S$ und $|S| = k$, dann werden diese k Elemente auf $k-1$ Elemente abgebildet und die Abbildung f kann eingeschränkt auf S somit nicht injektiv sein.
- Somit gibt es zwei Fälle
- 1. Fall: Es existiert *kein* solches $a \in A$
  - Dann gilt $S = A$ und wir sind fertig (Induktionshypothese nicht benötigt)
  - Beweis: würde $f(b) = f(c) \in A$ für $b, c \in A, b \neq c$ gelten, dann würden n Elemente auf höchstens $n-1$ Elemente in A abgebildet werden (Widerspruch zur Ausgangssituation des 1. Falls)
- 2. Fall: Es existiert ein solches $a \in A$: Dies können wir aus der noch zu betrachtenden Teilmenge A' entfernen, da es nicht Teil der Lösung sein kann:
  - $A' := A \setminus \{a\}$
  - Da $|A'| = n-1$ wissen wir, wie eine Lösung für A' berechnet werden kann.



Suchen Maximaler Injektiver Abbildungen (5)

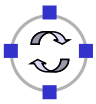
- Der rekursiv beschriebene Algorithmus kann auch iterativ implementiert werden:
  - Zunächst wird ein Array von Zählern $c[i]$ berechnet, dass angibt, wie viele Elemente auf das Element i abgebildet werden.
  - In jedem Schritt werden die Elemente i entfernt, deren Zähler $c[i] = 0$ ist.

```
algorithm MaxInjectiveMapping (int f[], n)
   Eingabe: ein Feld f[1, n] mit n Werten ∈ {1...n}
   Ausgabe: S, eine Menge so dass f injektiv auf S ist
   S := {1...n}; T := Ø;
   for j := 1 to n do c[j] := 0; od
   for j := 1 to n do c[f[j]]++; od
   for j := 1 to n do
                 if c[j] = 0 then Insert j in T; fi od
   while T ≠ Ø do
                 Remove arbitrary i from T;
                 S := S \ {i};
                 c[f[i]]--;
                 if c[f[i]] = 0 then Insert f[i] in T; fi
   od
```



Suchen Maximaler Injektiver Abbildungen (6)

- Aufwandsanalyse:
  - Die Initialisierung der Zähler erfordert *O(n)* Schritte.
  - Jedes Element kann maximal einmal in die Menge *T* eingefügt und maximal einmal aus den Mengen *T* und *S* entnommen werden; hierbei fällt jeweils konstanter Aufwand an.
  - Die Gesamtzahl der Schritte ist daher *O(n)*
- Bemerkungen:
  - In diesem Beispiel wurde das Problem dadurch reduziert, dass aus einer Menge zu betrachtender Elemente jeweils ein Element entnommen wurde.
  - Wir haben dabei die einfachste Methode gewählt, ein Element auszusuchen, dass entfernt werden kann, ohne dass die Bedingungen des ursprünglichen Problems verändert werden.
  - Da *f* die Menge *S* in sich selbst abbilden soll, lag es auf der Hand, ein Element auszuwählen, auf das kein anderes Element abgebildet wird.



Identifizieren einer Berühmtheit (1)

- Gegeben sei eine Menge von n Personen:
  - Eine *Berühmtheit* ist eine Person, die alle anderen kennen, die selber jedoch niemanden kennt.
  - Es können Fragen der Form „Verzeihen Sie, kennen Sie diese Person dort?“ gestellt werden (alle antworten und niemand lügt).
  - Gesucht ist die Berühmtheit bzw. die Antwort, dass keine Berühmtheit unter den *n* Personen ist.
  - Wir repräsentieren diese Aufgabe durch eine $n \times n$-Matrix *Know*, mit *Know*[*i, j*] = 1 wenn Person *i* die Person *j* kennt; und *Know*[*i, j*] = 0 sonst

Problem *Berühmtheit*

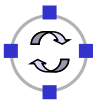
Gegeben: $n \times n$-Matrix *Know*

Gesucht: Index $i \in \{1...n\}$ (bzw. Entscheidung dass kein *i* existiert),
so dass: 1.) $\forall\, j \in \{1...n\} \setminus \{i\}$: *Know*[*j, i*] = 1
2.) $\forall\, j \in \{1...n\} \setminus \{i\}$: *Know*[*i, j*] = 0



Identifizieren einer Berühmtheit (2)

- Für den durch die Matrix *Know* beschriebenen Graphen *G* mit |*G*| = n suchen wir demnach eine *Senke*, d.h. einen Knoten, dessen Eingangsgrad gleich *n-1* und dessen Ausgangsgrad gleich 0 ist.
- Das Ziel bei dem Entwurf eines geeigneten Algorithmus ist es, die Anzahl der zu stellenden Fragen zu minimieren:
  - Da $n \cdot (n-1) / 2$ mögliche Paare existieren, reichen $n \cdot (n-1)$ Fragen aus.
  - Der Basisfall |*G*| = 2 ist einfach.
  - Wir konzentrieren uns daher auf den Unterschied des Problems mit *n-1* und des Problems mit *n* Personen:
    - Wir nehmen an, dass wir eine Berühmtheit unter *n-1* Personen identifizieren können (Induktionshypothese).
    - Da es maximal eine Berühmtheit geben kann, existieren drei Fälle:
      1. Die Berühmtheit ist unter den *n-1* Personen
      2. Die Berühmtheit ist die *n*-te Person
      3. Es ist keine Berühmtheit unter den *n* Personen



Identifizieren einer Berühmtheit (3)

- Der erste Fall ist einfach zu behandeln: Wir müssen lediglich prüfen, ob die *n*-te Person die Berühmtheit der *n-1* Personen kennt, und dass diese Berühmtheit die *n*-te Person nicht kennt (erfordert 2 Fragen).
- Der zweite Fall erfordert $2 \cdot (n-1)$ Fragen!
  $\Rightarrow T(n) = T(n-1) + 2 \cdot (n-1) \Rightarrow T(n) = n \cdot (n-1)$:o(

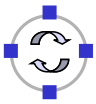
- Geht das nicht besser?
  - Es scheint aufwändig zu sein, eine Berühmtheit zu identifizieren.
  - Vielleicht ist es einfacher, jemanden zu identifizieren, der keine Berühmtheit ist?
    - Wir können schließlich irgendeine Person auswählen (es gibt viele!)
    - Wir fragen also Alice ob sie Bob kennt:
      - Falls ja, ist Alice keine Berühmtheit
      - Falls nein, ist Bob keine Berühmtheit
    - Somit können wir mit jeder Frage jeweils eine Person entfernen!



Identifizieren einer Berühmtheit (4)

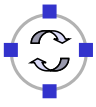
- Nachdem das Problem der Größe *n-1* per Induktion gelöst wurde, müssen wir es zu einer Lösung für das Problem der Größe *n* erweitern.
- Wir betrachten wieder die drei Fälle:
  1. Berühmtheit unter *n-1* Personen: Zwei Fragen genügen.
  2. Berühmtheit ist *n*-te Person: Dieser Fall kann nicht auftreten!
     (die *n*-te Person wurde extra so gewählt)
  3. Keine Berühmtheit: Keine weitere Frage erforderlich.
- Aufwandsanalyse: $T(n) = T(n-1) + 3; \quad T(1) = 0$

$$\Rightarrow T(n) = 3 \cdot (n-1)$$



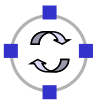
Identifizieren einer Berühmtheit (5)

```
algorithm Celebrity (int know[][], n)
  Eingabe: nxn-Matrix know, Dimension n der Matrix
  Ausgabe: celebrity // englisch für Berühmtheit
  i := 1; j := 2; next := 3;
  while next ≤ n+1 do
    if know[i, j] = 1 then i := next; else j := next; fi
    next++; od  // entweder i oder j wird entfernt
  if i = n + 1 then cand := j else cand := i; fi
  wrong := 0; k := 1;  // we now check the candidate
  while wrong = 0 and k ≤ n do
    if know[cand, k] = 1 and k ≠ cand then wrong = 1; fi
    if know[k, cand] = 0 and k ≠ cand then wrong = 1; fi
    k++;
  od
  if wrong = 0 then celebrity := cand;
               else celebrity := 0; fi
```



Identifizieren einer Berühmtheit (6)

- Bemerkungen:
  - Die wesentliche Idee an diesem Algorithmus ist es, das Problem der Größe *n* in „geschickter Weise“ auf ein Problem der Größe *n-1* zu reduzieren
  - Der Algorithmus liefert ein Beispiel dafür, dass es sich manchmal lohnt, Aufwand in die Reduktion des Problems zu investieren (hier: eine Frage), da dann die Lösung des kleineren Problems einfacher erweitert werden kann.
  - Es wird somit nicht ein beliebiges kleineres Problem gewählt, sondern das kleinere Problem wird gezielt aus dem größeren Problem hergeleitet.
  - Dieser Gedanke wird auch bei *QuickSort* angewendet: der Algorithmus *Partition* reduziert das Problem in geschickter Weise so auf zwei kleinere Teilprobleme, dass die Lösungen der Teilprobleme einfach zusammengefügt werden können.



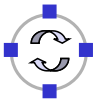
Suchen von Teilsequenzen mit Maximaler Summe (1)

Problem *MaxSumSubsequence*

Gegeben: Eine Sequenz $x_1, x_2, \ldots, x_n$ reeller Zahlen (inkl. negative)

Gesucht: Eine zusammenhängende Teilsequenz $x_i, x_{i+1}, \ldots, x_j$ so dass die Summe ihrer Elemente maximal ist.

- Wir nennen eine solche Sequenz eine *maximale Teilsequenz*
- Beispiel:
  - Sequenz: (2, -3, 1.5, -1, 3, -2, -3, 3)
  - Lösung: (1.5, -1, 3); die Summe ist 3.5
- Sind alle Zahlen negativ, so ist die maximale Teilsequenz leer.
- Wir suchen nach einem Algorithmus, der die Sequenz nur einmal lesen muss (linearer Aufwand).



Suchen von Teilsequenzen mit Maximaler Summe (2)

- Erster Ansatz:

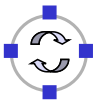
  Induktionshypothese 1:

  - Wir wissen wie in Sequenzen mit $n - 1$ Elementen eine maximale Teilsequenz gefunden werden kann.

  Induktionsanfang: $n = 1$: trivial

  Induktionsschritt: $n\text{-}1 \rightsquigarrow n$:

  - Wir betrachten eine Sequenz $S = (x_1, ..., x_n)$ mit $n > 1$ Elementen
  - Nach Induktionshypothese können wir eine maximale Teilsequenz S'_M in der Folge $S' = (x_1, ..., x_{n-1})$ finden:
    - Ist S'_M leer, dann sind alle Zahlen in S' negativ, und wir müssen lediglich prüfen, ob x_n negativ ist.
    - Angenommen S'_M ist nicht leer, d.h. $S'_M = (x_i, x_{i+1}, ..., x_j)$ für geeignete $1 \leq i \leq j \leq n\text{-}1$



Suchen von Teilsequenzen mit Maximaler Summe (3)

Induktionsschritt (Fortsetzung):

- Falls $j = n\text{-}1$, dann ist die maximale Subsequenz von S´ ein *Suffix,* d.h. sie erstreckt sich ab einem Element bis zum Ende von S´
  - In diesem Fall kann die Lösung einfach erweitert werden:
    - Ist $x_n \geq 0$, so erweitert x_n diese Lösung und $S_M = S'_M \,||\, (x_n)$
    - Ist $x_n < 0$, so ist die bisherige Lösung maximal und $S_M = S'_M$
- Falls $j < n\text{-}1$, dann existieren zwei Möglichkeiten:
  - S'_M bleibt maximal, oder
  - Es gibt eine andere Teilsequenz, die maximal wird, wenn x_n dazu addiert wird.
- Hier stoßen wir auf das Problem, dass unsere Induktionshypothese nicht ausreicht, um eine Lösung für das kleinere Problem auf eine Lösung für das größere Problem zu erweitern.
- Wir werden die Induktionshypothese daher stärken.



Suchen von Teilsequenzen mit Maximaler Summe (4)

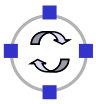
Induktionshypothese 2:

- Wir wissen wie in Sequenzen mit < *n* Elementen eine maximale Teilsequenz und die maximale Teilsequenz, die ein Suffix ist, gefunden werden können.

Induktionsanfang: *n = 1*: trivial

Induktionsschritt: *n-1* ⇝ *n* :

- Damit ist die Erweiterung der Lösung des kleineren Problems einfach:
  - Wir addieren x_n zum maximalen Suffix. Ist der resultierende Wert größer, als die Summe der bisherigen maximalen Teilsequenz, so haben wir eine neue maximale Teilsequenz und ein neues maximales Suffix gefunden.
  - Andernfalls ist die bisherige maximale Teilsequenz auch die Neue.
  - Wir müssen jedoch auch das neue maximale Suffix berechnen. Wird dieses negativ durch Hinzufügen von x_n, so ist das neue maximale Suffix eine leere Teilsequenz.



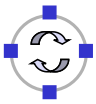
Suchen von Teilsequenzen mit Maximaler Summe (5)

```
algorithm MaxSumSubsequence (double S[]; int n)
  Eingabe: Ein Feld mit n reellen Zahlen, Feldlänge n
  Ausgabe: globalmax die Summe der maximalen Subsequenz
  globalmax := 0; suffixmax = 0;
  for i := 1 to n  do
    if S[i] + suffixmax > globalmax
      then suffixmax := suffixmax + S[i];
           globalmax := suffixmax;
      else if suffixmax + S[i] > 0
           then suffixmax := suffixmax + S[i];
           else suffixmax := 0;
      fi
    fi
  od
// Merken der entsprechenden Indizes ist eine einfache Übung
```



Reflexion: Stärken der Induktionshypothese

- Das Stärken der Induktionshypothese ist eine der wichtigsten Techniken beim Beweisen mathematischer Sätze per Induktion:
  - Angenommen wir wollen den Satz *P* per Induktion beweisen.
  - Die Induktionshypothese bezeichnen wir mit *P(<n)*.
  - Der Induktionsschritt muss demnach zeigen: *P(<n)* $\Rightarrow$ *P(n)*
  - In vielen Fällen wird dieser Schritt einfacher, wenn wir eine zweite Annahme *Q* hinzunehmen:
    - In diesen Fällen ist es einfacher zu zeigen: [*P and Q*](<n) $\Rightarrow$ *P(n)*
    - Damit der Induktionsbeweis jedoch richtig wird, müssen wir nun *Q* in die Induktionshypothese aufnehmen und zeigen, dass auch *Q* in jedem Schritt erhalten bleibt
    - Wir zeigen also insgesamt: [*P* and *Q*](<n) $\Rightarrow$ [*P* and *Q*](*n*)
  - Achtung: Wird *Q* im Induktionsschritt vergessen, so wird der Beweis falsch!
- Diese Technik ist auch für den Entwurf von Algorithmen zentral
  - In unserem Beispiel war *Q* gleich dem Berechnen des maximalen Suffix



Ein Einfaches Rucksackproblem (1)

- Angenommen, wir haben einen Rucksack mit einem bestimmten Fassungsvermögen und möchten wissen, ob wir diesen mit einer Auswahl aus gegebenen Gegenstände exakt füllen können.

Problem *ExactKnapsack*

Gegeben: Fassungsvermögen *k* eines Rucksacks und eine Sequenz $w_1, w_2, \ldots, w_n$ von Gewichten (alles positive ganze Zahlen)

Gesucht: Eine Untermenge S von Indizes, so dass $k = \sum_{i \in S} w_i$

- Wir bezeichnen dieses Problem als *P(n, k)*: wir können aus *n* Gegenständen auswählen und müssen exakt das Gewicht *k* erhalten.
- Die Gewichte der Gegenstände werden nicht mit in die Notation des Problems aufgenommen (ähnlich wie die Inhalte zu sortierender Folgen), und somit bezeichnet *P(i, k)* das Problem, den Rucksack mit einer Auswahl aus den ersten *i* Gegenständen exakt zu füllen.



Ein Einfaches Rucksackproblem (2)

- Um die Diskussion einfach zu halten, konzentrieren wir uns auf das Problem, zu entscheiden, ob eine solche Lösung existiert.
  - Eine Lösung dieses Problems kann leicht um die Berechnung der entsprechenden Indizes erweitert werden.

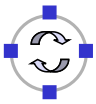
Induktionshypothese 1: Wir wissen wie P(n-1, k) gelöst werden kann.

Induktionsanfang: $n = 1$:

- In diesem Fall kann nur dann eine Lösung existieren, wenn der einzige Gegenstand das Gewicht k hat.

Induktionsschritt: $n-1 \rightsquigarrow n$:

- Wenn $P(n-1, k)$ eine Lösung hat, dann brauchen wir den n-ten Gegenstand nicht und haben auch eine Lösung für $P(n, k)$.
- Angenommen, $P(n-1, k)$ hat keine Lösung: Können wir dieses negative Resultat verwenden?
  - Ja, in diesem Fall kann nur eine Lösung mit dem n-ten Gegenstand existieren, d.h. wir müssen prüfen ob $P(n-1, k-w_n)$ eine Lösung hat.



Ein Einfaches Rucksackproblem (3)

- Wir müssen unsere Induktionshypothese daher stärken:

Induktionshypothese 2: Wir wissen wie $P(n-1, j)$ für alle $0 \leq j \leq k$ gelöst werden kann.

Induktionsanfang: $n = 1$:

- In diesem Fall existiert nur für $w_1 = j$ bzw. im Fall $j = 0$ eine Lösung.

Induktionsschritt: $n-1 \rightsquigarrow n$:

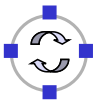
- Wie bereits erläutert reduzieren wir $P(n, k)$ auf die beiden Teilprobleme $P(n-1, k)$ und $P(n-1, k-w_n)$. Wenn $k - w_n < 0$ ist, können wir das zweite Teilproblem ignorieren.
- Diese Reduktion hängt von keinem bestimmten Wert von k ab.
- Die Reduktion ist somit gültig und wir können den zugehörigen Algorithmus einfach „ablesen“.



Ein Einfaches Rucksackproblem (4)

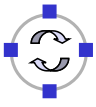
- Aufwandsanalyse:
  - Wir betrachten die Rekurrenz für den Aufwand des Problems in Abhängigkeit der Anzahl zu untersuchender Gegenstände
  - $T(n) = 2 \cdot T(n-1) + 1$ mit $T(1) = 1 \Rightarrow T(n) = \Theta(2^n)$
  - Der Algorithmus fordert einen exponentiell mit der Problemgröße anwachsenden Aufwand! :o(
- Geht das nicht besser?
  - Können überhaupt so viele unterschiedliche Teilprobleme auftreten?
  - Oder lösen wir einige identische Probleme mehrfach?
  - Wir haben das Problem mit *P(n, k)* bezeichnet. Da *n* und *k* ganze positive Zahlen sind, können insgesamt maximal $n \cdot k$ unterschiedliche Probleme auftreten.
  - Wir sollten uns daher die Lösung einmal berechneter Teilprobleme merken, um die mehrfache Berechnung zu vermeiden.

Induktionshypothese 3: Wir wissen, wie *P(<n, j)* für alle $0 \leq j \leq k$ gelöst werden kann.



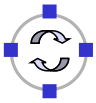
Ein Einfaches Rucksackproblem (5)

- Implementierung:
  - Wir speichern alle Zwischenergebnisse in einer $n \times k$-Matrix *M*.
  - Der Eintrag *(i, j)* enthält:
    - die Entscheidung, ob das Problem *P(i, j)* lösbar ist: $M(i, j).exist \in \{0, 1\}$
    - die Information, ob der *i*-te Gegenstand Teil der Lösung ist: $M(i, j).belong \in \{0, 1\}$
  - Die Reduktion aus unserem Induktionsbeweis berechnet somit die Elemente der *n*-ten Zeile dieser Matrix
  - Falls *P(n, k)* eine Lösung hat, können die zu dieser Lösung gehörenden Gegenstände anschließend durch „intelligentes Ablaufen“ der Ergebnismatrix ausgehend von *M(n, k)* und Auslesen der *belong*-Einträge ermittelt werden (Übungsaufgabe).



Ein Einfaches Rucksackproblem (6)

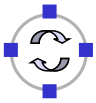
```
algorithm SimpleKnapsack (int S[]; int n, k)
   Eingabe: Feld S mit n ganzen Zahlen, Rucksackgröße k
   Ausgabe: P ein 2-dimensionales Array mit Einträgen
            .exist und .belong wie zuvor erklärt

   P[0,0].exist := 1; P[0,0].belong := 0;
   for j := 1 to k  do P[0, j].exist := 0; od

   for i := 1 to n  do
     for j := 0 to k  do
      P[i, j].exist := 0; // the default
      if P[i-1, j].exist = 1
        then P[i, j].exist := 1; P[i, j].belong := 0;
        else if ((j-S[i] ≥ 0) and (P[i-1, j-S[i]].exist = 1))
             then P[i, j].exist := 1; P[i, j].belong := 1;
             fi fi
   od od
```



Ein Einfaches Rucksackproblem (7)

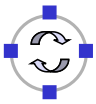
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$w_1 = 2$	0	–	1	–	–	–	–	–	–	–	–	–	–	–	–	–	–
$w_2 = 3$	0	–	0	1	–	1	–	–	–	–	–	–	–	–	–	–	–
$w_3 = 5$	0	–	0	0	–	0	–	1	1	–	1	–	–	–	–	–	–
$w_4 = 6$	0	–	0	0	–	0	1	0	0	1	0	1	–	1	1	–	1

- Ein Beispiel der Tabelle *M* für *S* = (2, 3, 5, 6) für alle *k* bis 16:
  - Ein Eintrag „–“ bedeutet, dass das entsprechende P(i, j) keine Lösung hat.
  - Bei Eintrag „0“ hat P(i, j) eine Lösung ohne den *i*-ten Gegenstand.
  - Bei Eintrag „1“ hat P(i, j) eine Lösung mit dem *i*-ten Gegenstand.
  - Befindet sich ein „–“ in Spalte *j* der letzten Zeile, hat das Problem P(n, j) keine Lösung.



Ein Einfaches Rucksackproblem (8)

- Bemerkungen:
  - Die hier angewendete Methode ist ein Beispiel für die allgemeine Technik der sogenannten *„Dynamischen Programmierung“*:
    - Dabei werden große Tabellen mit allen bisher berechneten Ergebnissen konstruiert, da die Lösungen häufig wiederverwendet werden müssen, um weitere Lösungen zu berechnen
    - Die Tabellen werden dabei iterativ konstruiert und jeder Eintrag wird mit Hilfe von Einträgen berechnet, die entweder oberhalb von ihm oder in der gleichen Zeile links von ihm stehen.
  - Komplexität:
    - Es müssen $n \cdot k$ Einträge berechnet werden. Jeder Eintrag kann in konstanter Zeit berechnet werden $\Rightarrow \Theta(n \cdot k)$
    - Das Auslesen der Indizes für eine Lösung $P(n, k)$ benötigt $\Theta(n)$ Schritte.
    - Wie benötigen $n \cdot k$ Tabelleneinträge (Speicheraufwand)
  - Ist k groß oder eine reelle Zahl, so ist diese Technik nicht einsetzbar.



Fallstricke beim Algorithmenentwurf per Induktion

- Analog zu den Fallstricken bei Induktionsbeweisen, existieren auch beim Entwurf von Algorithmen typische Fehlerquellen:
  - Vergessen bzw. unvollständige Behandlung des Induktionsanfangs:
    - Der Basisfall einer rekursiven Prozedur wird nicht korrekt behandelt, und der Algorithmus wird somit falsch.
  - Erweitern einer Lösung für den allgemeinen Fall des Problems der Größe n auf einen Spezialfall des Problems der Größe $n+1$:
    - Diesen Fehler macht man leicht, wenn z.B. nicht alle möglichen Fälle korrekt erkannt und behandelt werden.
  - Ändern der Induktionshypothese:
    - Tritt z.B. auf, wenn die Induktionshypothese eine bestimmte Eigenschaft des Problems voraussetzt, und der Reduktionsschritt diese Eigenschaft nicht erhält
    - Beispiel: Bei einem Graphenalgorithmus wird vorausgesetzt, dass der Graph zusammenhängend ist, und im Reduktionsschritt wird eine beliebige Kante (oder Knoten) entfernt.



Suchen Minimaler und Maximaler Elemente (1)

Problem *MinMaxElements*

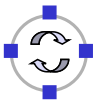
Gegeben: Ein Folge *F* mit *n* Zahlen

Gesucht: Das kleinste und das größte Element in *F*

- Eine einfache Vorgehensweise besteht darin, beide Probleme unabhängig voneinander zu lösen (z.B. nacheinander):
  - Das Minimum kann in *n-1* Schritten gefunden werden
  - Das Maximum kann anschließend in *n-2* Schritten gefunden werden
  - Gesamtaufwand: *2n - 3* Schritte
- Geht es besser?

  Induktionshypothese: Wir wissen wie in einer Folge mit n-1 Elementen das maximale und minimale Element gefunden werden können.

  Induktionsanfang: *n = 1*: trivial



Suchen Minimaler und Maximaler Elemente (2)

- Geht es besser? (Fortsetzung)

  Induktionsschritt:
  - Angenommen, wir kennen das Minimum und das Maximum aus n-1 Zahlen, und wir wollen diese Lösung auf das Problem inkl. der n-ten Zahl ausdehnen.
  - Hierfür benötigen wir zwei weitere Vergleiche.
  - $T(n) = T(n-1) + 2;\quad T(1) = 0;\quad T(2) = 1 \Rightarrow T(n) = 2n - 3$:o(
- Geht es nicht doch besser?
  - Versuchen wir nun, die per Induktion gefundene Lösung um mehr als ein Element in einem Schritt zu erweitern, also z.B. um zwei Elemente
  - Wir benötigen jetzt zwei „Anker“ für unsere Induktion (*n=1, n=2*)
  - Nun können wir mit drei Vergleichen die Lösung für *n-2* erweitern:
    - Wir vergleichen x_n mit x_{n-1}
    - Wir vergleichen den größeren Wert mit Max_{n-2}
    - Wir vergleichen den kleineren Wert mit Min_{n-2}



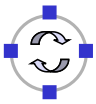
Suchen Minimaler und Maximaler Elemente (3)

- Geht es nicht doch besser? (Fortsetzung)
  - Anstelle von vier Vergleichen bei zweimaliger Induktion von $n\text{-}1 \rightarrow n$, benötigen wir bei einmaliger Induktion von $n\text{-}2 \rightarrow n$ nur drei Vergleiche
  - $T(n) = T(n\text{-}2) + 3;\ T(1) = 0;\ T(2) = 1; \quad \Rightarrow T(n) = 3n / 2$

- Kann man das durch gleichzeitiges Hinzunehmen von noch mehr Elementen weiter verbessern?

  → Nein, die Schranke von *3n/2* kann für dieses Problem nicht unterschritten werden.

- In der Übung wird ein Divide-und-Conquer-Algorithmus behandelt, der auch *3n/2* Vergleiche benötigt



Suchen des *k*-Kleinsten Elements (1)

Problem *k-KleinstesElement*

Gegeben: Folge $S = s_1, s_2, ..., s_n$ mit n Zahlen und eine Zahl $1 \le k \le n$

Gesucht: Das k-kleinste Element in S

- Liegt k nahe bei 1 oder nahe bei n, können wir das Problem dadurch lösen, dass wir den Algorithmus für das minimale bzw. maximale Element k-mal anwenden. Aufwand: $T(n) = n \cdot k$
- Im Fall, dass k mehr als $\log n$ von beiden Werten entfernt ist, liefert Sortieren und anschließendes Zugreifen auf das k-te Element einen effizienteren Algorithmus. Aufwand: $T(n) = n \cdot \log n$
- Es gibt jedoch noch einen (im mittleren Fall) effizienteren Algorithmus, der ähnlich wie QuickSort vorgeht:
  - Grundidee: Zerlege wie bei Quicksort um ein Pivot-Element herum und suche jedoch nur in dem Teil weiter, der das Element enthalten muss.



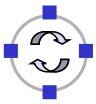
Suchen des *k*-Kleinsten Elements (2)

Induktionshypothese: Wir wissen wie in einer Folge mit <*n* Elementen das k-kleinste Element gefunden werden kann.

Induktionsanfang: *n = 1*: trivial (korrekt, da *1 ≤ k ≤ n)*

Induktionsschritt:

- Wir wollen das Problem *P(n, k)*, in einer Folge *S* mit *n* Elementen das *k*-kleinste Element zu finden, auf ein kleineres Problem reduzieren.
- Hierzu wählen wir das erste Element von *S* als Pivot-Element und Zerlegen die Folge wie bei QuickSort mit dem Algorithmus *Partition*, der uns auch die neue Position *middle* des Pivot-Elements zurückgibt.
- Anschließend prüfen wir, ob *k* < *middle* ist:
  - Wenn ja, suchen wir per Induktion in der Teilfolge $(s_1, ..., s_{middle})$ das *k*-kleinste Element; das ist ja ein identisches Problem *P(<n, k)*
  - Wenn nein, suchen wir per Induktion in der Teilfolge $(s_{middle+1}, ..., right)$ das *(k-middle)*-kleinste Element; das ist auch ein identisches Problem *P(<n, <k)*



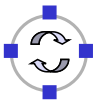
Suchen des *k*-Kleinsten Elements (3)

```
algorithm k-SmallestElement (int S[]; int n, k)
  Eingabe: Feld S mit n ganzen Zahlen, Zahl k
  Ausgabe: ks, das k-kleinste Element
  if (k < 1) or (k > n) then print(„Error!");
                       else ks := Select(S, 1, n, k); fi


  procedure Select(S, left, right, k)
  { if left = right then return left;
    else m = Partition(S, left, right);
         if k < m-left+1
         then return Select(S, left, m, k);
         else return Select(S, m+1, right, k-(m-left+1));
         fi
    fi
  }
```



Suchen des *k*-Kleinsten Elements (4)

- Aufwand:
  - Wie bei QuickSort führt eine „unglückliche“ Wahl des Pivot-Elements zu einem quadratischen Aufwand (schlechtester Fall).
  - Da in jedem rekursiven Aufruf nur ein Teilproblem gelöst werden muss, ist der Algorithmus im mittleren Fall schneller als QuickSort.
  - Im mittleren Fall wird das Problem in zwei ungefähr gleich große Teile geteilt, von denen nur eines weiter bearbeitet werden muss:
    - $T(n) = T(n/2) + n$
    - Abschätzung mit Master-Theorem: $a = 1, b = 2, f(n) = n$
    - Da $n^{\log_b a} = n^{\log_2 1} = n^0$, gilt $f(n) = \Omega(n^{\log_b a + \varepsilon})$
    - Weiterhin ist: $a \cdot f(n/b) = 1 \cdot f(n/2) = n/2 \leq c \cdot f(n)$ für $1/2 \leq c < 1$
    - Es kann somit Fall 3 des Master-Theorems angewendet werden und es gilt: $T(n) = \Theta(f(n)) = \Theta(n)$



Suchen von Zeichenketten in Zeichenketten (1)

- Seien $A = a_1\ a_2\ ...\ a_n$ und $B = b_1\ b_2\ ...\ b_m$, $m \leq n$ zwei Zeichenketten über einem endlichen Alphabet.
- Ein Teilzeichenkette (engl. Substring) der Zeichenkette A ist eine konsekutive Sequenz von Zeichen $a_i\ a_{i+1}\ ...\ a_j$ von A.
- Mit $A(i)$ und $B(i)$ bezeichnen wir die speziellen Substrings $a_1\ a_2\ ...\ a_i$ und $b_1\ b_2\ ...\ b_i$.

Problem *FindSubstring*

Gegeben: Zeichenketten A und B

Gesucht: Das erste Vorkommen der Zeichenkette B in A (falls B in A vorkommt), d.h. das kleinste k so dass gilt $\forall 1 \leq i \leq m: a_{k+i} = b_i$

- Beispiele für Anwendungen:
  - Textverarbeitung: Suchen von Wörtern oder Mustern in Texten
  - Molekularbiologie: Suchen von Mustern in RNA oder DNA Molekülen



Suchen von Zeichenketten in Zeichenketten (2)

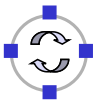
- Das Problem sieht auf den ersten Blick einfach aus:
  - Wir fangen beim ersten Zeichen a_1 von A an und vergleichen nacheinander alle b_i mit den a_i.
- Bezeichne $P(n, m)$ das Problem in A der Länge n die Position des ersten Vorkommens von B der Länge m zu bestimmen.

  Induktionshypothese: Das Problem $P(n, m)$ kann gelöst werden.

  Induktionsanfang: $n = 1$: trivial

  Induktionsschritt: $n \rightsquigarrow n + 1$
  - Sei $|A| = n+1$. Wir vergleichen das erste Element a_1 mit b_1:
    - Wenn $a_1 \neq b_1$ und $m < n+1$, dann reicht es, das Problem $P(n, m)$ zu lösen. Bei $a_1 \neq b_1$ und $m \geq n+1$ kann es keine Lösung geben.
    - Wenn $a_1 = b_1$ dann brauchen wir nur noch das Problem $P(n, m-1)$ zu lösen.

      Achtung: Das ist keine gültige Reduktion, da die Zeichenkette B ja zusammenhängend in A vorkommen soll!



Suchen von Zeichenketten in Zeichenketten (3)

- Wir benötigen daher eine andere Problemformulierung:
- Bezeichne $PM(n, m)$ das Problem, zu bestimmen, ob die Zeichenkette B in A ab der Position 1 vorkommt (PM steht für *PrefixMatch*).

  Induktionshypothese: Die Probleme $P(n, m)$ und $PM(n, \leq m)$ können gelöst werden.

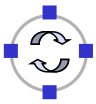
  Induktionsanfang: $n = 1$: trivial

  Induktionsschritt: $n \rightsquigarrow n + 1$
  - Sei $|A| = n+1$. Wir vergleichen wieder a_1 mit b_1:
    - Wenn $a_1 \neq b_1$ siehe vorige Folie.
    - Wenn $a_1 = b_1$ versuchen wir zuerst das Problem $PM(n, m-1)$ zu lösen.
    - Ergibt hierbei $PM(n, m-1)$, das $b_2 \ldots b_m$ kein Prefix von $a_2 \ldots a_{n+1}$ ist, so müssen wir noch versuchen, das Problem $P(n, m)$ zu lösen.
    - Alle auftretenden Teilprobleme sind laut Induktionshypothese lösbar.
  - Aufwand (schlechtester Fall): Es ist „fast“ immer $a_1 = b_1$ aber $PM(n, 1)$ ergibt stets, dass das letzte Zeichen nicht übereinstimmt:
    - $T(n, m) = T(n-1, m) + T(n-1, m-1);\ T(n, 1) = 1; \Rightarrow T(n, m) = n \cdot m$



Suchen von Zeichenketten in Zeichenketten (4)

- Bei diesem Problem hat uns die Technik der vollständigen Induktion zwar einen Algorithmus finden lassen, der das Problem löst, aber:
  - Im schlechtesten Fall ist der Algorithmus nicht sehr effizient.
  - Der Induktionsbeweis war schwieriger als der Algorithmus selbst.
- Es gibt einen effizienteren Algorithmus, der das Problem auch im schlechtesten Fall in *O(n)* löst:
  - Der Algorithmus wurde von *Knuth, Morris* und *Pratt* entwickelt und ist unter dem Namen *KMP* bekannt.
  - Die wesentliche Idee ist eine geeignete Vorverarbeitung von B, um bei einem „Mismatch“ mehr als ein Zeichen nach vorne rücken zu können
  - Bei „üblichen“ Texten werden sogar weniger als *n* Vergleiche benötigt
  - Dieser Algorithmus ist allerdings per Induktion nicht einfach herzuleiten.
- Dennoch können wir an diesem Beispiel zwei Dinge lernen:
  - Bei der Reduktion darf man das Problem nicht verändern.
  - Induktion ist keine „Allzweckwaffe“ beim Algorithmenentwurf.



Vergleiche von Sequenzen (1)

- Seien $A = a_1\ a_2\ ...\ a_n$ und $B = b_1\ b_2\ ...\ b_m$ wieder zwei Zeichenketten über einem endlichen Alphabet (hier keine Bedingung an *n* und *m*).
- Wir würden A gerne so ändern, dass es gleich B wird.
- Hierfür haben wir drei Operationen zur Verfügung:
  - *Insert:* Fügt ein Zeichen in die Zeichenkette ein
  - *Delete:* Löscht ein Zeichen aus der Zeichenkette
  - *Replace:* Ersetzt ein Zeichen in der Zeichenkette durch ein anderes
- Beipiel: *A = abbc B = babb*
  - Lösung 1: Lösche das erste *a* → *bbc*
Füge *a* zwischen die beiden *b* ein → *babc*
Ersetze *c* durch b → *babb*
  - Lösung 2: Füge *b* am Anfang ein → *babbc*
Lösche *c* → *babb*
- Das Ziel ist es wieder, so wenig Schritte wie möglich zu benötigen.
- Das Problem hat zahlreiche Anwendungen in der Molekularbiologie.



Vergleiche von Sequenzen (2)

- Es bezeichnen *A(i)* und *B(i)* wieder Prefix-Substrings $a_1\ a_2\ ...\ a_i$ bzw. $b_1\ b_2\ ...\ b_i$

Problem *MinEditDistance*

Gegeben: Zeichenketten *A(n)* und *B(m)*

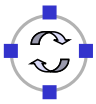
Gesucht: Die minimale Anzahl von Änderungen mit Operationen *Insert*, *Delete* und *Replace*, so dass *A(n)* in *B(m)* überführt wird.

Induktionshypothese: Wir können die MinEditDistance von *A(n-1)* und *B(m)* berechnen.

Induktionsanfang: $n = 1$: trivial

Induktionsschritt: $n\text{-}1 \rightsquigarrow n$

- Wir suchen die Lösung für *A(n)* und *B(m)*. Hierzu berechnen wir per Induktion die Lösung für *A(n-1)* und *B(m)*. Anschließend löschen wir a_n.



Vergleiche von Sequenzen (3)

Induktionsschritt: (Fortsetzung)

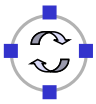
- Aber: diese Lösung ist unter Umständen nicht minimal!
  - Vielleicht wäre es besser, a_n durch b_m zu ersetzen?
  - Vielleicht gilt sogar bereits $a_n = b_m$?
- Daher müssen alle unterschiedlichen Möglichkeiten betrachtet werden, wie mit Hilfe der besten Lösungen für kleinere Teilsequenzen von A und B die minimale Folge von Änderungsschritten um A in B zu überführen konstruiert werden kann.

- Wir bezeichnen mit *C(i, j)* die minimalen Kosten, um *A(i)* in *B(j)* zu überführen und interessieren uns nur für die Kosten (die dazu gehörenden erforderlichen Schritte können leicht durch einfache Ergänzung ermittelt werden).
- Wir wollen zunächst eine Beziehung zwischen *C(n, m)* und den *C(i, j)* für Kombinationen kleinerer *i* und *j* herleiten.



Vergleiche von Sequenzen (4)

- Es ergeben sich die folgenden Möglichkeiten:
  1. *delete:* wenn a_n gelöscht werden soll in der minimalen Änderungsfolge, so ergeben sich die Kosten zu: $C(n, m) = C(n-1, m) + 1$
  2. *insert:* wenn die minimale Änderungsfolge erfordert, dass ein Zeichen hinzugefügt werden muss, ergibt sich: $C(n, m) = C(n, m-1) + 1$
  3. *replace:* wenn a_n das Zeichen b_m ersetzen soll, muss zunächst die minimale Änderungsfolge von $A(n-1)$ zu $B(m-1)$ gefunden werden. Es ergibt sich somit: $C(n, m) = C(n-1, m-1) + 1$
  4. *match:* wenn $a_n = b_m$ ergibt sich: $C(n, m) = C(n-1, m-1)$
- Es sei $d(i, j) = \begin{cases} 0 \text{ falls } a_i = b_j \\ 1 \text{ falls } a_i \neq b_j \end{cases}$



Vergleiche von Sequenzen (5)

- Damit kann die folgende Rekurrenzrelation aufgestellt werden:

$$C(n, m) = \min \begin{cases} C(n-1, m) + 1 & (\text{delete } a_n) \\ C(n, m-1) + 1 & (\text{insert for } b_m) \\ C(n-1, m-1) + d(n, m) & (\text{replace/match } a_n) \end{cases}$$

$$\text{mit } C(i, 0) = i \text{ für } 0 \leq i \leq n \text{ und } C(0, j) = j \text{ für } 0 \leq j \leq m$$

- Problematisch an diesem Ansatz ist, dass wir die Induktion zu häufig anwenden:
  - So müssen für das Problem $C(n, m)$ drei nur unwesentlich kleinere Probleme gelöst werden.
  - Das Resultat ist ein Algorithmus mit exponentiell steigendem Aufwand! :o(
- Glücklicherweise, müssen die drei Teilprobleme nicht unabhängig voneinander gelöst werden:
  - Wir berechnen für jedes Teilproblem $C(i, j)$ für $0 \leq i \leq n$ und $0 \leq j \leq m$
  - Insgesamt kann es nur $n \cdot m$ Kombinationen solcher $C(i, j)$ geben.



Vergleiche von Sequenzen (6)

- Das gleiche Phänomen haben wir bereits bei unserem einfachen Rucksack-Problem beobachtet.
- Wir benutzen daher wieder *starke Induktion*.

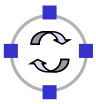
Induktionshypothese: Wir können die MinEditDistance für Probleme <(n, m) berechnen.

- Hierbei bedeutet <(n, m) dass mindestens eine der beiden Parameter kleiner als *n* bzw. *m* ist und keiner der Parameter größer ist als *n* bzw. *m*.

Induktionsanfang: trivial (→ Rekurrenzrelation für *C(i, 0)* und *C(0, j)*)

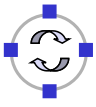
Induktionsschritt:

- Die starke Induktionshypothese erlaubt uns, jedes der auftretenden Teilprobleme in der Rekurrenzrelation zu berechnen.
- Wenn wir die Ergebnisse der vorgehenden Probleme in einer Tabelle speichern, benötigen wir für jedes neue Teilproblem konstanten Aufwand.
- Insgesamt ergibt sich für die Laufzeit somit $O(n \cdot m)$



Vergleiche von Sequenzen (7)

```
algorithm MinEditDistance(char A[], B[]; int n, m)
  Eingabe: Zwei Zeichenketten A, B; Ihre Längen n, m
  Ausgabe: C die Kostenmatrix der MinEditDistance
  for i = 0 to n do C[i, 0] = i; od
  for j = 0 to m do C[0, j] = j; od
  for i = 1 to n do
    for j = 1 to m do
      x := C[i-1, j] + 1;
      y := C[i, j-1] + 1;
      if A[i] = B[j]
        then z := C[i-1, j-1];
        else z := C[i-1, j-1] + 1; fi
      C[i, j] := min(x, y, z);
      // hier könnte auch die korrespondierende Operation
      // zusätzlich in einer zweiten Matrix vermerkt werden
    od od
```



Zusammenfassung für den Anhang

- Vollständige Induktion ist ein mächtiges Werkzeug beim Entwurf von Algorithmen:
  - Oft ist es einfacher, sich zunächst darauf zu konzentrieren, zu beweisen, dass das Problem überhaupt gelöst werden kann, als zu versuchen, direkt eine ausführbare Anweisungsfolge für den Algorithmus anzugeben.
  - In Kombination mit Rekurrenzrelationen kann so oft nicht nur die Existenz eines Algorithmus bewiesen, sondern zugleich sein Aufwand bestimmt werden.
  - Es gibt aber auch Probleme, bei der andere Herangehensweisen einfacher sind (das merkt man, wenn man es mit Induktion versucht).
- Nützliche Techniken:
  - *Stärken der Induktionshypothese:* hilft oft weiter, wenn die bisher angenommene Hypothese nicht ausreicht, um die Lösung zu erweitern. Hierbei muss darauf geachtet werden, dass der Induktionsschritt wiederum die gestärkte Hypothese erhält.
  - *Starke Induktion:* Alle Probleme <n können gelöst werden (Hypothese).

