6.0 Entwurf von Algorithmen mittels Induktion

{Lit.: U. Manber: *Using Induction to Design Algorithms*.
Communications of the ACM, Nov. 1988, 1300-1312}

6.0.1 Einleitung

Domino–Prinzip der mathem. Induktion:

1. Lösung eines ***Basisproblems***
2. Problemlösung aus Lösungen kleinerer Probleme (***Induktionsschritt***)

6.0.2 Polynomberechnung

(P1) Geg.: Reelle Zahlen $a_n, a_{n-1}, \ldots, a_0; x$.
Ges.: Wert $P_n(x) = a_n x^n + \ldots + a_1 x + a_0$.

1. induktive Lösung: Entfernen a_n.

➔ Induktionshypothese:
Können $P_{n-1}(x)$ berechnen.
Basisproblem: $P_0(x) = a_0$
Ind.-schritt: $P_n(x) = a_n * x^n + P_{n-1}(x)$
→ ½ n (n+1) Multiplikationen und n Additionen

2. induktive Lösung: $x^n = x * x^{n-1}$.

➔ Induktionshypothese:
Können $P_{n-1}(x)$ ***und*** x^{n-1} berechnen.
→ 2 n Multiplikationen und n Additionen

3. induktive Lösung: Entfernen a_0.

➔ Induktionshypothese:
Können $P'_{n-1}(x) = a_n x^{n-1} + a_{n-1} x^{n-2} + \ldots + a_1$ berechnen.
Ind.-schritt: $P_n(x) = x P'_{n-1}(x) + a_0$
→ n Multiplikationen und n Additionen

$\Rightarrow$ ges. A. durch folg. Ausdruck beschrieben:

$$a_n x^n + \ldots + a_1 x + a_0 = ((\ldots((a_n x + a_{n-1}) x + a_{n-2})\ldots) x + a_1) x + a_0$$

HORNERsche Schema

Algorithmus *Polynom_Berechnung(a, x):*

Input: **a** = (a_0, a_1, …, a_0) – Koeffizienten des Polynoms ;
x – reelle Zahl ;

Output: P – Wert des Polynoms im Punkt x;

```
BEGIN
    P := a_n;
    FOR i := 1 TO n DO P := x * P + a_{n-i}
END.
```

6.0.3 Maximal induzierter Teilgraph

(P2) Geg.: Ungerichteter Graph G=(V, E); Integer k>0.
Ges.: Subgraph H=(U, F) von G mit max. Größe, so dass alle Knoten aus H mindestens den Grad k besitzen; oder stelle fest, dass solch Subgraph nicht existiert.

Induktionshypothese: Können max. induz. Subgraph mit einem Grad der Knoten von mindestens k finden für Knotenanzahl n–1.
Basisproblem: Knotenzahl n–1 = k+1.
Ind.-schritt: Knotenzahl n > k+1.

Für Entwurf eines A. folg. ***Idee***:

Denken nicht über einen A. nach, sondern darüber, *wie ein Theorem zu beweisen ist, dass ein A. existiert*.

Zusammenfassung:

A. und Antwort auf obige Fragen ist klar →

- beliebiger Knoten mit Grad < k kann entfernt werden
- Reihenfolge des Entfernens ist unwichtig
- resultierender Graph ist die Lösung, da alle entfernten Knoten entfernt werden müssen

Bemerkung

Bester Weg, ein Problem zu reduzieren – einige seiner Elemente entfernen.
Hier – klar, welche Elemente zu entfernen sind.
In anderen Fällen folg. Vorgehensweisen:

- 2 Elemente zu einem verbinden
- Entfernung von Nebenbedingungen anstelle von Elementen
- Entwurf eines spez. A. zur Bestimmung des zu entfernenden Elementes (folg. Bsp.)

6.0.4 Das VIP–Problem

➔ folg. *graphentheoretische Formulierung*:

Bauen gerichteten Graph mit Pfeil von Knoten A zu Knoten B, falls Person A die Person B kennt.
Berühmtheit = *Senke* des Graphen

P3:	Geg.:	Adjazenz–Matrix $A_{n\times n}$.
	Ges. :	i* mit $a_{i^* j} = 0$, $j \neq i^*$, und $a_{i^* i^*} = 1$.

Induktionshypothese: Können VIP–Problem für n–1 Personen lösen.

a) *Basisfall:* $\rightarrow$ n = 2 Personen

b) *Induktionsschritt:* Da maximal 1 VIP $\Rightarrow$ 3 Fälle

(1) VIP ist unter den ersten n–1.
(2) VIP ist die n. Person.
(3) Kein VIP vorhanden.

➔ brauchen andere Lösung

TRICK:

Lösen das „umgekehrte“ Problem – ob eine Person keine VIP ist.

WENN A gefragt wird, ob er B kennt $\rightarrow$
BEI „Ja“, DANN „A ist kein VIP“;
BEI „Nein“, DANN „B ist kein VIP“.

➔ ***Mit einer Frage eine Person eliminiert!***

➔ A.:

Fragen A, ob mit B bekannt.
Entfernen A oder B in Abhängigkeit von Antwort.
Wenn z. B. A entfernt, so

(i) bestimmen VIP unter den restlichen n–1 Personen;
(ii) prüfen, ob A die VIP kennt und ob A der VIP unbekannt ist.

➔ 2 Phasen: (für iterative Implementierung)
Phase 1 $\rightarrow$ eliminieren alle Kandidaten bis auf einen;
Phase 2 $\rightarrow$ testen, ob letzte Kandidat eine VIP ist.

Bemerkung zur Komplexität:

$\Rightarrow$ Lösungsaufwand proportional zur Problemgröße n;
für die apriori–Lösung war er proportional zu n^2;

Algorithmus *VIP(A):*

Input: Adjazenz–Matrix A als Boole'sche n×n Matrix

Output: VIP

```
BEGIN
  i:=1;
  j:=2;
  next:=3;        { Stack implementiert über Variablen i, j, next. }

  WHILE  next ≤ n+1  DO
            BEGIN     { Entweder i oder j wird entfernt. }
              IF  A[i, j]  THEN  i:= next
                           ELSE   j:= next;
              next := next+1
            END;

  IF  i = n+1  THEN  kand_VIP:= j
               ELSE   kand_VIP:= i;

  { Nun folgt Test, ob kand_VIP wirklich VIP ist. }
  kein_VIP := false;
  k := 1;
  A[kand_VIP, kand_VIP] := false;
                  {Dummy–Variable, um den Test zu bestehen.}
  WHILE  NOT kein_VIP  AND  k≤n  DO
      BEGIN
        IF  A[kand_VIP, k]  THEN  kein_VIP := true;
                  {kand_VIP kennt k}
        IF  NOT A[k, kand_VIP]    {k kennt kand_VIP nicht}
            THEN  IF  kand_VIP ≠ k  THEN  kein_VIP := true;
        k := k+1
      END;

  IF  NOT  kein_VIP
      THEN  VIP := kand_VIP
      ELSE   VIP := 0        {kein VIP vorhanden}
END.
```

6.0.5 Bestimmung der maximalen Teilfolge

P4:	Geg.:	Folge $x_1, x_2, \ldots, x_n$ reeller Zahlen.
	Ges.:	Teilfolge $x_i, x_{i+1}, \ldots, x_j$ mit maximaler Summe.

z.B.: x = (2, -3, 1.5, -1, 3, -2, -3, 3)
=> max. Teilfolge maxt = (1.5, -1, 3) mit Summe = 3.5

Vereinbarung: Leere Teilfolge hat Summe 0.

Induktionshypothese:
Können maxt in Folge mit n–1 Elementen finden.

a) *Basisfall:* n = 1

b) *Induktionsschritt:* $x = (x_1, x_2, \ldots, x_n)$, $n > 1$;
können maxt in $x' = (x_1, x_2, \ldots, x_{n-1})$ finden;

➔ ***Grundidee:*** verschärfen die Induktionshypothese

Strengere Induktionshypothese:
Wir können maxt in Folge mit < n Elementen finden, und wir können die max. Teilfolge, die ein Suffix ist, finden.

➔ A. ist offensichtlich:
addieren x_n zum Suffix;
wenn die entsprechende Summe > maxt–Summe,
dann haben wir neue maxt (und neuen Suffix)
sonst bleibt alte maxt erhalten;

Was noch offen? Brauchen noch neuen *max. Suffix*!

Algorithmus *Max_Teilfolge(x, n):*

Input: x (als 1–dimensionales Feld der Länge n)

Output: Max (Summe der max. Teilfolge)

```
BEGIN
      Max := 0;
      Suffix := 0;     {max. Suffix}


      For i := 1 TO n DO

            IF  x[i] + Suffix  >  Max

                  THEN
                        BEGIN
                              Suffix := Suffix + x[i];
                              Max := Suffix
                        END

                  ELSE    IF  x[i] + Suffix  >  0
                                 THEN  Suffix := Suffix + x[i]
                                 ELSE   Suffix := 0

END.
```

6.0.6 Das Rucksackproblem

Rucksack–Problem: (knapsack problem)

Geg.: Integer V und
n Gegenstände mit ganzzahligen Volumen $v_1, v_2, \ldots, v_n$.
Ges.: Menge von Gegenständen mit Gesamtvolumen V.

P(n, V) := Rucksack–Problem mit n Gegenständen und Volumen V
(nehmen an, dass v_i als Input gegeben)

P(i, v) – entspr. Problem mit den ersten i Gegenständen und einem Rucksack mit Volumen v;

* ***Entscheidungsproblem***: Gibt es eine Lösung oder nicht.

Induktionshypothese: (1. Versuch)
Können P(n–1, V) lösen.

Basisfall: Für P(1, V) ex. Lösung nur, wenn $v_1 = V$.

Ind.-schritt: Für P(n, V) ➔
a) P(n–1, V) hat Lösung, so auch P(n, V).
b) P(n–1, V) hat keine Lösung =>
2 kleinere Probleme: P(n–1, V) und P(n–1, V–v_n)

Induktionshypothese: (2. Versuch)
Können P(n–1, V) für ***alle*** $0 \le v \le V$ lösen.

Basisfall: Für P(1, v) =>
v = 0 – immer eine triviale Lösung;
v > 0 – g.d. Lösung, wenn $v_1 = v$.

Induktionsschritt: P(n, V) ➔

2 Probleme P(n–1, v) und P(n–1, v–v_n) ➔
{bei v<v_n wird 2. Problem ignoriert}

=> A. mit expon. Aufwand, aber nur n · V Probleme

Gleiche Probleme mehrfach erzeugt!

➔ **Idee:** Schon erreichte Lösungen merken bzw. speichern!

Algorithmus *Knapsack(S, K):*

Input: S – Feld der Länge n für Volumen der Gegenstände
K – Volumen des Rucksackes

Output: P – 2-dim. Feld, wo P[i, k].exist = true, falls Lösung für P(i, k) ex., sowie P[i, k].dazu = true, falls Gegenstand i zur Lösung gehört

```
BEGIN
    P[0, 0].exist := true;
    FOR  i := 1  TO  K  DO  P[0, k].exist:=false;
    {brauchen P[i, 0] für i > 0 nicht zu initialisieren, da aus P[0, 0] berechnet}
    FOR  i := 1  TO  n  DO
       FOR  k := 0  TO  K  DO
          BEGIN
            P[i, k].exist := false;    {Default–Wert}
            IF  P[i–1, k].exist
                THEN BEGIN
                            P[i, k].exist := true;
                            P[i, k].dazu := false
                          END
                ELSE   IF  k–S[i] >= 0
                             THEN   IF  P[i–1, k–S[i]].exist
                                          THEN BEGIN
                                                 P[i, k].exist := true;
                                                 P[i, k].dazu := true
                                                   END
          END
END.
```