# 6. Algorithmen II

* bisher : Konstruktion effektiver und sauber strukturierter Programme
u.a. bleibt noch:
→ Korrektheit als wichtiger qualitativer Aspekt;
→ Komplexität als wichtiger quantitativer Aspekt;

## 6.1. Korrektheit von Algorithmen

* Ziel der Algorithmierung: Entwicklung eines korrekten Algorithmus

korr. A : = A., bei dem bei Einhaltung der Vorbed. und strikter Befolgung der Rechenvorschriften die Gültigkeit der Nachbed. impliziert wird.

* nochmal zu Vor- und Nachbedingungen:

Vor jeder Berechnung wird best. Erg. erwartet ⇒
als Zusicherung (Nachbed.; postcondition)
! Nur Aussage über Resultat; nicht wie berechnet.

Damit Zusicherung gilt, müssen best. Vorbed. (preconditions) erfüllt sein.

z.B.: FUNCTION *doppelt* (*arg* : integer) : integer;
BEGIN *doppelt* := 2 * *arg* END;

⇒ Zusicherung: *doppelt* (*x*) = 2 * *x*
⇒ Vorbedingung: z.B. $\underbrace{x \in \text{INT} \ \wedge \ |x| \leq \text{MAXINT}/2}$

Ist eine Aussage! Kein Programm!

auch strengere Vorbed. sichern korr. Erg., z.B. $x \in \text{INT} \ \wedge \ 0 \leq x \leq \text{MAXINT}/4$
⇒ Suche nach der schwächsten Vorbedingung (weakest precondition → wp)

* Frage: Wie Korrektheit prüfen? ➔ i.w. Verifizieren + Testen;

### 6.1.1. Verifizieren

* Verifikation : = Mathematisch exakter Beweis der Korrektheit von Algorithmen und Algorithmenteilen.

- nur für A., die keine Anweis.-teile in Form der menschlichen Sprache enthalten
- benötigen best. Formalismus →

* Bezeichnungen + Definitionen:
Fkt.-vorschrift kann bezgl. ihres Effektes beschrieben werden durch:
- Zusicherung für das Ergebnis: Q ;
- Vorbedingung: P ;

{Diese Angaben gehören in Kopfkommentar eines jeglichen A., da sie u.a. beim Verifizieren zentrale Rolle spielen}

Bsp.: 1) doppelt $Q \equiv \mathit{doppelt}\,(x) = 2 * x$
$P \equiv x \in \text{INT} \wedge |x| \leq \text{MAXINT}/2$

Kommentar: Liefert das Doppelte einer Zahl, die betragsmäßig MAXINT/2 nicht übersteigt.

2) SQRT $Q \equiv \text{SQRT}\,(x) \approx \sqrt{x}$
$P \equiv x \in \text{REAL} \wedge x \geq 0$

Kommentar: Liefert Quadratwurzel für nichtneg. Parameter

{Zusicherung muß exakt formuliert auf REAL-Arithmetik Rücksicht nehmen: $Q \equiv |\text{SQRT}\,(x) - \sqrt{x}| < \frac{\sqrt{x}}{1000} \rightarrow$ relat. Fehler von 0.1%}

folg. Formalismus →

S – Progr., Teil eines Pr., Anweisungsfolge;
P – Aussage über die Var./Param. des Progr. vor Ausführung von S;
Q – Aussage über die Var./Param. des Progr. nach Ausführung von S;

$\Rightarrow$ $P \quad \{S\} \quad Q$ (*)

P: Vorbed. Q: Nachbed.

Solche Aussage (*) kann je nach Kontext wie folgt verstanden werden:
(i) S – einzelne Anweisung ⇒ (*) dient Def. dieser;
(ii) S – Programm ⇒ (*) ist Beschreibung des Pr.;
(iii) S – zu entw. Programm ⇒ (*) ist Spezifikation des Pr.;

⇒ Verifikation := Nachweis der Verträglichkeit von Spezifikation und Beschreibung

⇒ formal: Pr. ist verifiziert, wenn bewiesen, daß
$P_{SPEZ} \Rightarrow P_{BES} \wedge Q_{BES} \Rightarrow Q_{SPEZ}$

d.h.,
- die spezifizierte Vorbed. impliziert die Vorbed. des realisierten Pr.;
- die Nachbedingung des realisierten Pr. impliziert die spezifizierten Nachbed.;

In Praxis: $Q_{SPEZ}$ vorgegeben.
Oft einfacher, wenn von allgemeinerem $Q_{BES}$ ausgegangen wird und $P_{BES}$ als $wp(S, Q_{BES})$ gesucht.

⇒ Rückwärts-Vorgehen von den angestrebten Nachbed. zu den unvermeidlichen Vorbedingungen.

* schwächste Vorbed. (per. Def.)

- für die leere Anweisung $\varepsilon$ $\rightarrow$

  Semantik: $wp(\varepsilon, Q) \equiv Q$

  Informell: Was nach Ausführung von $\varepsilon$ gelten soll, muß bereits vorher gelten.

- für die Wertzuweisung $\rightarrow$

  Semantik: $wp(x := E, Q(x)) \equiv \exists\, E \in TYP(x) \wedge Q(E)$

  {meist reduziert auf $wp(x := E, Q(x)) \equiv Q(E)$}

  Informell: E muß definiert und typkompatibel mit $x$ sein. Dann ist die schwächste Vorbed. gleich der Nachbedingung wobei $x$ durch den Ausdruck E ersetzt ist.

- für die Sequenz $\rightarrow$

  Semantik: $wp(S_1; S_2, Q) \equiv wp(S_1, wp(S_2, Q))$

  Informell: Die schwächste Vorbed. der letzten Anweisung $S_2$ ist Nachbedingung der vorletzten Anweisung usw. .

- für die Alternative $\rightarrow$

  $wp(\text{IF B THEN } S_1; \text{ ELSE } S_2, Q)$

  $\equiv wp(B \Rightarrow wp(S_1, Q)) \wedge wp(\neg B \Rightarrow wp(S_2, Q))$

  $\equiv wp(B \wedge wp(S_1, Q)) \vee wp(\neg B \wedge wp(S_2, Q))$

  die unvollst. Alternative $\rightarrow$

  $S_1 := S$ und $S_2 := \varepsilon$

  $wp(\text{IF B THEN S}, Q)$

  $\equiv wp(B \wedge wp(S, Q)) \vee wp(\neg B \wedge Q)$

* Beispiele zur wp:

1. Anweisung S sei gegeben durch

   IF $x < 0$ THEN $x := -x$ ELSE $x := x - 1$;

   $x \in$ INT; $Q \equiv x \geq 0$

$\Rightarrow wp(s, Q) \equiv ((x < 0) \wedge wp(x := -x, \overbrace{x \geq 0}^{E})) \vee ((x \geq 0) \wedge wp(x := x - 1, x \geq 0))$

$\equiv ((x < 0) \wedge (-x \geq 0)) \vee ((x \geq 0) \wedge (x - 1) \geq 0)$

$\equiv (x < 0) \vee (x > 0) \equiv (x \neq 0)$

$\Rightarrow$ Vorbed. $x > 10$ wäre hinreichend, $x \leq 10$ dagegen nicht!

2. S $\Rightarrow$ $z := x * y$ ; $x := x$ DIV 2 ; $y := 2 * y$ ;

   $Q \equiv z := x * y$

Eigenschaft!

$\Rightarrow$ $wp(S, Q) \equiv wp(z := x * y ;\ x := x \text{ DIV } 2 ;\ y := 2 * y ,\ z := x * y)$

$\equiv wp(z := x * y ;\ x := x \text{ DIV } 2 ,\ \underbrace{wp(y := 2 * y , z := x * y))}_{z := x * 2 * y}$

$\equiv wp(z := x * y ,\ \underbrace{wp(x := x \text{ DIV } 2 , z := x * 2 * y))}_{z := (x \text{ DIV } 2) * 2 * y}$

$\equiv wp(x * y = (x \text{ DIV } 2) * 2 * y)$

Ausdruck ist wahr, wenn

(a) $y = 0 \ \wedge \ x$ – beliebig,

(b) $x = 0 \ \wedge \ y$ – beliebig,

(c) $y \neq 0 \ \wedge \ (x = (x \ \text{DIV} \ 2) * 2) \equiv y \neq 0 \ \wedge \ x$ – gerade

$\Rightarrow P \equiv (y = 0) \vee (x = 0) \vee (x$ – gerade$)$

3. S $\Rightarrow$ $t := t * x$ ; $i := i + 1$ ;

$i, n \in$ INT; $Q \equiv i \leq n \ \wedge \ t = x^i$

daraus folgt mit der Nebenbed. $x \neq 0$, daß

$$\begin{aligned} wp(S_1; S_2, Q) &\equiv wp(S_1, wp(S_2, Q)) \\ &\equiv wp(t := t * x, wp(i := i + 1, 1 \leq n \ \wedge \ t = x^i)) \\ &\equiv wp(t := t * x, i + 1 \leq n \ \wedge \ t = x^{i+1})) \\ &\equiv (i + 1 \leq n) \wedge (t * x = x^{i+1}) \\ &\equiv (i < n) \wedge (t = x^i) \end{aligned}$$

! Bem.: Der Teil $(t = x^i)$ der Vorbed. stimmt mit jenem Teil der Nachbedingung exakt überein. Der Zusammenhang $t = x^i$ wird also durch S nicht verändert, obwohl $t$ und $i$ verändert werden.

$\Rightarrow$ Bez. für derart. Prädikat (Bed.): Invariante $\rightarrow$

## 6.1.2. Verifikation iterativer Programme

* im Prinzip wie oben, aber sehr mühevoll;
noch anderer Ansatz

$\rightarrow$ Lit.: Appelrath/Ludwig: Skriptum Informatik 3. Aufl.; Teubner 1995;

dazu folg. Pr.-Teil:

$x, n, t, i \in$ INT; $x, n$ – const. ;

```
WHILE  i <= n  DO  BEGIN
                       t := t * x ;   }
                       i := i + 1;    }  S
                   END;
```

Nachbedingung lautet $Q(t, i) \equiv t = x^n$

* Schleifenrumpf als Variablentransformator:
sei geg. WHILE B(V) DO S ;

V – Vektor der Variablen, die von S verändert;
B – von V abhängige Wiederholbedingung;

$V' = h(V) = [h_1(V), h_2(V), ..., h_k(V)]$ – Änderung von V nach 1 Ausführ. von S (bei $k$ Var.);

$\Rightarrow$ h – durch Schleifenkörper realisierte Variablentransformation ;

für unser Bsp. $\rightarrow$ $V = (t, i)$ und $h(t, i) = [t \cdot x, i+1]$

j-malige Ausführung von S $\rightarrow$

$V_j = h(V_{j-1}) = ... = h(h(...h(V_0))) =: H(V_0 ; j)$ mit $V_0 = V$; $H(V_0 ; 0) = V_0$

für unser Bsp. $\rightarrow$ $H(t, i ; j) = [t \cdot x^i, i + j]$

* Prinzip der (Schleifen-) Invariante:

Schleifeninvariante INV ist eine Aussage über V, die stets gültig ist, wenn die Wdh.-bed. geprüft wird.

$\Rightarrow$ $INV(V_0) \equiv TRUE$

unter der Bed. $B(V_0) \wedge B(V_1) \wedge ... \wedge B(V_m)$ auch

$INV(h(V_m)) \equiv INV(V_{m+1}) \equiv TRUE$

Klar: „$\equiv$ TRUE“ ist überflüssig, somit kann die Inv. charakterisiert werden durch

$INV(V_0) \wedge B(V_0) \wedge B(V_1) \wedge ... \wedge B(V_m) \Rightarrow INV(h(V_m))$

$\Rightarrow$ Inv. beschreibt einen Zush. der Variablen, der erhalten bleibt, solange Schleife durchlaufen wird. Bei Abbruch ist B(V) nicht mehr gegeben, so daß

$INV(V_j) \wedge \neg B(V_j)$.

Ist Inv. richtig gewählt, so folgt hieraus

$Q \equiv (INV(V_j) \wedge \neg B(V_j)) \Rightarrow Q(V_j)$

Zusf.: geeignete Invariante muß 3 Bed. erfüllen

(1) Sie muß zu Beginn gelten.
(2) Sie muß, sofern B erfüllt war, unter Anwendung von h tatsächlich invariant sein.
(3) Sie muß hinr. aussagekräftig sein, um nach Abbruch den Schluß auf 0 zu erlauben.

$\hookrightarrow$ z.B. für $INV(V) \equiv TRUE$ nicht erfüllt; ist trivialerweise eine Invariante jeder Schleife;

für unser Bsp. $\rightarrow$ $\boxed{INV(t, i) \equiv i \leq n + 1 \wedge t = x^{i-1}}$

Beweis: induktiv $\rightarrow$

I) Induktionsbeginn:

Sei Vorbed. geg. als $P \equiv (i = 1) \cap (t = 1) \cap (n \geq 0)$.

Weiterhin gilt $B(V) \equiv (i \leq n)$

a) $n = 0 \Rightarrow$ $\neg B(V_0) \Rightarrow$

$(i = 1) \cap (t = 1) \equiv (i \leq n + 1) \cap (t = x^{i-1}) \equiv INV(t, i)$

b) $n > 0 \Rightarrow$ $B(V_0) \Rightarrow$

$(i = 2) \cap (t = x) \equiv (i \leq n + 1) \cap (t = x^{i-1}) \equiv INV(t, i)$

II) Induktionsübergang:
Seien Inv. und Wdh.-bed. gültig für die Werte $t_m$ und $i_m$ nach dem m. Durchlauf. Dann gilt:

$$\mathrm{INV}(t_m, i_m) \cap \mathrm{B}(i_m) \equiv (i_m \leq n+1) \cap (t_m = x^{i_m - 1}) \cap (i_m \leq n).$$

Mit $h(t, i) = [t \cdot x,\ i+1]$ ergibt sich

$$t_{m+1} = t_m \cdot x = x^{i_m - 1} \cdot x = x^{i_m} \quad \text{sowie} \quad i_{m+1} = i_m + 1, \text{ so daß}$$

$$(t_{m+1} = x^{i_{m+1} - 1}) \wedge i_{m+1} = n+1 \equiv \mathrm{INV}(t_{m+1}, i_{m+1}) \qquad \blacksquare$$

Bem.: Als Nachbedingung erhält man

$$\begin{aligned} \mathrm{INV}(V) \cap \neg \mathrm{B}(i_m) &\equiv (i \leq n+1) \cap (t = x^{i-1}) \cap (i > n) \\ &\equiv (i_m = n+1) \cap (t = x^{i-1}) \\ &\equiv (t = x^n) \\ &\equiv Q(t, i) \end{aligned}$$

* Konstruktion iterativer Programme:

am schwierigsten bei Verifikation eines A. → Auffinden der Schleifeninvarianten;
die anderen Formeln → einfach „ausrechnen“;

folg. Schema, um Schleifen korrekt zu konstruieren (Appelrath/Ludewig):

1. Festlegung der ***Nachbedingung*** Q (Ziel der Konstruktion).
2. „Entdeckung“ einer geeigneten ***Invariante*** INV.
3. Festlegung der ***Vorbedingung*** P und Ableitung der notwendigen ***Initialisierung*** (Eingangsprüfung/Anforderungen an vorangehende Pr.-teile), d.h., ***Nachweis***, daß P die Invariante impliziert.
4. Festlegung der ***Wiederholbedingung*** B.
5. ***Nachweis***, daß INV aus 2. eine Invariante ist (d.h., bei gültigem B auch nach Durchlauf des Schleifenkörpers gilt).
6. ***Nachweis***, daß bei ¬ B die Nachbedingung Q gilt.

Bem.: Falls Schritt 6 ohne Erfolg, so Wdh. Ab Schritt 2.

Bsp.: Iterative Berechnung der Summe einer endlichen Folge reeller Zahlen

1° $Q \equiv \text{Res} = \sum_{j=1}^{n} a_j$

2° $\text{INV} \equiv \text{Res} = \sum_{j=1}^{i} a_j \ \wedge \ 0 \le i \le n$

Implementieren des Körpers S:

$i := i + 1;\ \text{Res} := \text{Res} + a_i$ ;

wp(S, Q) $\equiv$ wp($i := i + 1$, wp (Res := Res + $a_i$ , Q($i$, Res)))

$\equiv$ wp($i := i + 1$, Q($i$, Res + $a_i$))

$\equiv$ Q($i + 1$, Res + $a_{i+1}$)

$\Rightarrow$ h($i$, Res) = [$i + 1$, Res + $a_{i+1}$] als Transformation;
garantiert mit der noch zu def. Wiederholbed. B die Invariante;

Bem.: Die umgekehrte Reihenfolge in S ist falsch, da dort h(V)
Die Invariante verletzt. Beweis als **HA** !

3° $P \equiv \text{Res} = 0 \ \wedge \ i = 0 \ \wedge \ n \ge 0$

$P \Rightarrow \text{INV}$ ?

Antwort: $\text{Res} = 0 = \sum_{j=1}^{0} a_j \ \wedge \ 0 = i \le n \quad \Rightarrow$ Ja!

4° $B \equiv i < n$

5° $\text{INV}(V) \cap B(V) \Rightarrow \text{INV}(V')$ ?

Antwort:

INV(V') $\equiv$ INV(h($i$, Res)) $\equiv$ INV($i + 1$, Res + $a_{i+1}$)

$\equiv \ \text{Res} + a_{i+1} = \sum_{j=1}^{i} a_j + a_{i+1} = \sum_{j=1}^{i+1} a_j \ \wedge \ 0 \le i + 1 \le n$

$\text{INV}(V) \cap B(V) \equiv \ \text{Res} = \sum_{j=1}^{i} a_i \ \wedge \ 0 \le i \le n \cap i < n$

$\Rightarrow \text{INV}(V')$

6° $\text{INV}(V) \cap \neg B(V) \quad \Rightarrow Q(V)$ ?

Antwort:

$\text{Res} = \sum_{j=1}^{i} a_j \ \cap \ 0 \le i \le n \ \cap \ i \ge n \equiv \text{Res} = \sum_{j=1}^{i} a_j \cap \ i = n$

$\equiv \text{Res} = \sum_{j=1}^{n} a_j \equiv Q$ ■

$\rightarrow$ Schleife ist erfolgreich konstruiert.

$\Rightarrow$ folg. Implementierung

S
```
Res := 0;
   i := 1;
WHILE (i < n) DO BEGIN
                    i := i +1;
                    Res := Res +a_i
                  END;
```

Jedoch: In P ist $n \geq 0$ enthalten $\Rightarrow$
IF $n$ >= 0
THEN S
ELSE < Fehlermeldung >

* Definitionen: Korrektheit von Prog. unter verschiedenen Aspekten;

- zunächst – immer wenn A. endet, so das erwartete Erg.

  **Partielle Korrektheit** := Ein A. mit Vorbed. P und Nachbedingung Q heißt partiell korrekt g. d., wenn jede Eingabe, welche die Vorbedingung P erfüllt und für welche die Ausführung des A. terminiert, die Nachbedingung Q impliziert.

  z. B.

  $P \equiv k \in \mathbb{N} \cap a \in \mathbb{N}$
  S : WHILE ($k$ MOD $a$) > 0 DO $k$ := ($k$+2) MOD $a$ ;
  $Q \equiv (k \text{ MOD } a) = 0$

  $\Rightarrow$ Abbruch nur, wenn Q erfüllt!
  Q nicht erfüllbar, sobald $k$ gerade $\wedge$ $a$ ungerade bzw.
  $k$ ungerade $\wedge$ $a$ gerade für $k \neq a$.

  **HA**: Beweis!
  Hinweis: ungerade Zahl $\rightarrow \exists\, n \geq 0 : k = 2 \cdot n + 1$
  3 Fälle $k = a$ ; $k < a$ ; $k > a$ ;

$\Rightarrow$ einzige Garantie hier: wenn ein Erg. erzeugt wird, so ist es korrekt;

- zweiter wichtiger Aspekt – Terminiertheit

  **Totale Korrektheit** := Ein A. mit Vorbed. P und Nachbed. Q heißt (total) korrekt g. d., wenn für jede Eingabe, welche die Vorbed. P erfüllt, die Ausführung des A. anhält und die Ausgabe erfüllt die Nachbedingung Q.

* Terminiertheit von Schleifen:

grundlegende Technik, die Endlichkeit von Schleifen zu untersuchen

$\rightarrow$ beobachten Werte, die sich in best. Richtung mit der Anzahl der Schleifendurchläufe ändern;

Bsp.: $A$: $\mathbb{N}^2 \to \mathbb{N}$ gemäß

$$A(x, y) = \begin{cases} y + 1, & x = 0; \\ A(x - 1, 1), & y = 0; \\ A(x - 1, A(x, y - 1)), & \text{sonst}; \end{cases}$$

modifizierte (große) Ackermannfunktion;

```
FUNCTION  A (x, y : nichtnegative ganze zahl) : Integer;
BEGIN
        IF  x = 0
         THEN   A := y +1
         ELSE    IF  y = 0
                        THEN A := A(x – 1, 1)
                        ELSE  A := A(x – 1, A(x,  y – 1))
END;
```

Terminiert? Ja, da Bewegung auf Terminierbedingung $x = 0$ zu.

**HA**: Bestimmen Sie $A(2,2)$ und $A(3,2)$!

Endlichkeitsbeweise oft trivial, sofern A. richtig verstanden.
Nun anderes Bsp. – einf. Aussehen, aber nicht zu beweisen (z.Z.) →

Bsp.: $F$: $\mathbb{N}_0 \to \mathbb{N}_0$ gemäß

$$F(x) = \begin{cases} 1, & x = 1; \\ F(x / 2), & x > 1 \text{ und gerade}; \\ F(3x + 1), & x > 1 \text{ und ungerade}; \end{cases}$$

```
FUNCTION  F (x, y : Integer) : Integer;
BEGIN
        IF    x = 1      THEN F:=1
                         ELSE  IF  NOT  ODD(x)
                                        THEN   F := F(x / 2)
                                        ELSE    F := F(3 * x + 1)
END;
```

besser ⇒

```
PROGRAM   Trivial;
        VAR   x : Integer;
BEGIN
        WRITE('Eingabe x:  '); READLN(x);
        WHILE  x > 1 DO
                                IF  NOT  ODD(x)
                                        THEN   x := x / 2
                                        ELSE    x := 3 * x +1;
        WRITE('f(x) =  ', x);
        READLN;
END.
```

Bem.: Graf. Darstellung hilfreich!

* induktives Beweisen:

z.B. McCarthy's sogen. 91-Algorithmus →

```
FUNCTION  MC (x : nichtnegative ganze zahl) : Integer;
BEGIN
        IF  x > 100
         THEN   MC := x – 10
         ELSE   MC := MC(MC(x + 11))
END;
```

Induktionsbehauptung:

$$MC(x) = \begin{cases} x - 10 & \text{bei } x > 100; \\ 91 & \text{sonst;} \end{cases}$$

Induktionsanfang:

Für $x > 100 \rightarrow$ trivial;
Also $x = 100 \Rightarrow MC(100) = MC(MC(111)) = MC(101) = 91$.

Induktionsübergang:

Sei für ein $x < 100$ gezeigt, daß $MC(y) = 91$ für $x \le y \le 100$.
Zeigen für $x - 100$.
a) Falls $\overbrace{x + 10}^{(x-1)+11} > 100 \Rightarrow MC(x - 1) = MC(MC(x + 10)) = MC(x) = 91$ laut Induktionsvoraussetzung

b) Falls $x + 10 \le 100 \Rightarrow MC(x - 1) = MC(MC(x + 10)) = MC(91)$,
da laut Induktionsvoraussetzung $MC(y) = 91$ für $y = x + 10 \le 100$.

Mit $x + 10 \le 100$ gilt $x \le 90 \le 91$ und somit laut Induktionsvoraussetzung wieder $MC(91) = 91$. ■

### 6.1.3. Test

* formale Beweise → nur für kleinere Programme;
meistens Tests; {weil weniger gedanklicher Aufwand}

Testziel: Fehler zu entdecken!

Ist Test erfolgreich → Symptom (Spur) eines Fehlers entdeckt;
danach Spurenverfolgung: Fehlersuche;
schließlich: Fehlerbehandlung;

{Fehler – bug; ~ suche – debugging}

Programm(-teil) enthält Fehler, wenn es

(i) nicht tut, was es soll;
(ii) tut, was es nicht soll.

* Test := gezieltes und systematisches Erproben eines A. oder A.-teils mit Hilfe ausgewählter Testdaten (Testfälle).

$\Rightarrow$ repräsentat. Eingaben durch A. bearbeitet und die Ausgaben mit den laut Spezifikationen erwarteten Ausgaben verglichen

in Appelrath/Ludewig 4 Korrektheitsebenen:

Ebene 1: Pr. „läuft“ $\rightarrow$ syntakt. Korr.;
Ebene 2: Pr. liefert für einige ausgew. Eingabewerte korr. Ergebnisse;
Ebene 3: Pr. liefert für irgendwelche Eingabewerte korr. Ergebnisse;
Ebene 4: Pr. liefert für alle zulässigen Eingabewerte korr. Ergebnisse;

z.B.: a) Algor. MULT1 aus § 2.2.1 $\Rightarrow$

1. $m = 0 \wedge n > 0$ 2. $m = 0 \wedge n = 0$
3. $m$ – groß $\wedge$ $n$ – klein ...;

b) Suche $\Rightarrow$

1. $a \notin$ F 2. F = $\varnothing$ 3. $a = a_1 \vee a_n$
4. $a = a_i$ für $1 < i < n$

$\Rightarrow$ i.w. zu Ebene 2;

zu Ebene 4:

Für völlige Sicherheit ist A. auf ges. Eingabemenge $E$ anzuwenden.
$\rightarrow$ prakt. unmöglich (vgl. PROGRAM Trivial)
{Austesten unrealistischer als Suche der Stecknadel im Heuhaufen!}

! Testen kann nur Anwesenheit von Fehlern nachweisen, nicht aber deren Abwesenheit !!!

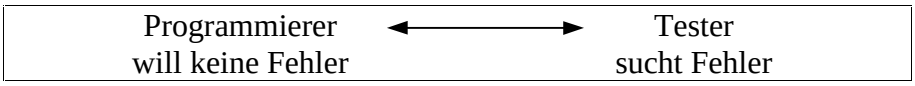
Wichtig dabei:

a) P und Q
b) Einblick in Anwendungsdomäne eines A.

„Auswege“:

(i) konstruktive Lösungen wo möglich
(ii) Prinzip der Lokalität (Prozed., Module, Kapsel)
(iii) Parallelität von A.-Entwurf und Korrektheitsbeweis
(iv) Test von „Klassen“
(v) grafische Darstellung des/der A.

* Aufgabenstellung und Zielsetzung




⇒ 2 versch. Personen; Gegenstand der SWE (SWT)

* Konstruktion von Testdaten (⇒ Testplan)

Spurensuche → unsystematisch oder gezielt

besser

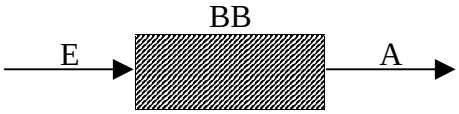
Auswahl der Testfälle unter folg. Gesichtspunkten:

a) Anforderungen der Spezifikation werden leicht mißverstanden;
b) Grenzfälle sind bei Implementation oft nicht korrekt. behandelt;
c) Fehlersituation i.d.R. weniger sorgfältig bearbeitet als der Normalfall;
d) Fehler in best. Anw./Verzweig. nur entdeckt, wenn diese durchlaufen:
e) Jede Änderung des Pfades, auf dem ein Pr. durchlaufen wird, kann Fehler offenbaren.

* Auswahlverf. für Testfälle ⇒

• Black-Box-Test (Schnittstellentest):

wenn kein Quellcode verfügbar, so nur an Schnittstellen Test möglich;
(bzgl. der Spezifikation);

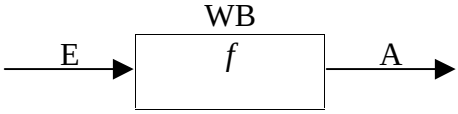



Spezifikation => $f: E \rightarrow A$

$\{ A_n = f(E_n),\ n = 1\ (1)N \} \rightarrow f$ implementiert oder nicht?

• White-Box-Test (programmabh. Test):

Testfälle auch auf Basis des Quelltextes ausgewählt;
nur hier Aspekte (d) und (e) zu beachten;



## 6.2. Komplexität

### 6.2.1. Einleitung

* bisher qualitative Merkmale: Überschaubarkeit; Korrektheit;

∃ auch quantitative Merkmale: Effizienz; Ressourcenverbrauch

hier: Aufwand zur Abarbeitung eines A. durch einen Prozessor;

Jedoch: unterschiedl. Prozessoren + Algorithmen ⇒
„langsamer“ A. auf Pentium vielleicht mehr Rechenzeit als „schneller“ auf 386er;
Oder: Ein „erstbester“ A. ist gut für kleinere Probleme und schrecklich für große Probleme.

→ Frage 1: Wie Aufwand messen?
→ Frage 2: Was sind „kleine“ bzw. „große“ Probleme?

1 Antwort: Wählen eine Basisoperation und best. bzw. schätzen deren Anzahl ab.

* Bsp. MULT1:

Basisoperation → Add. oder Subtr.;
Aufwand := $A_{MULT1}(m, n) \approx$ Anz. Durchläufe = $2 \cdot n$

Aber: Aufwand von Eingabe $n$ abh. →
wenn $m$ klein und $n$ groß ⇒ vertausche $m$ und $n$ :

MULT2 mit vorbereitendem Schritt
(0) k ← $n$; n ← min($m$, $k$); $m$ ← max($m$, $k$);

⇒ $A_{MULT2}(m, n) \approx 2 \cdot \min(m, n)$; !

⇒ MULT2 ≻ MULT1

* Bsp. SUCHE:

Wie hier Aufwand? Basisoperation: Vergleich; Wertzuweisung;

Wenn Suche bei 1. Element begonnen →

$$A(n) = \begin{cases} 2, & \text{falls } a_1 = a; \\ 2 \cdot n, & \text{falls } a_n = a; \end{cases}$$

$$A(n) = 2 \cdot i, \quad \text{falls } a_i = a, \; i = 1(1)n;$$

⇒ Nutzen ist gering;

$\Rightarrow$ 3 übliche Maße:

| | | |
|---|---|---|
| $A_{min}(n)$ | – | Aufwand im günstigsten Fall (best-case-Analyse) |
| $A_{max}(n)$ | – | Aufwand im ungünstigsten Fall (worst-case-Analyse) |
| $\overline{A}(n)$ | – | Aufwand im Mittel (average-case-Analyse) |

$\downarrow$
?

Nur klar: Abh. von Werten der Eingabeparameter. Benötigen Wahrscheinlichkeitsverteilung für Position des gesuchten Elements $\rightarrow$ vereinfachte Annahme: Alle Positionen sind gleich wahrscheinlich.

$$\Rightarrow \qquad \overline{A}(n) = \sum_{i=1}^{n} 2 \cdot i \cdot P(a_i = a) = \frac{1}{n} \cdot 2 \cdot \sum_{i=1}^{n} = \frac{2}{n} \cdot \frac{n}{2}(n+1) = \underline{\underline{n+1}}$$

* Vergleich von A. $\Rightarrow$ i. d. R. bzgl.

a) $\overline{A}(n)$
b) $A_{max}(n)$ und
c) asymptotischer Aufwand

Warum c) ?

Seien z.B. zwei Algorithmen I und II gegeben
mit $\overline{A_I}(n) = 2n^2 + 50$ und
$\overline{A_{II}}(n) = 100 \cdot n$
Welcher ist „besser“ ?

Für I ignorieren wir 2 und 50 und sagen: Aufwand ist proportional $n^2$.
Da $n^2$ größer n sagen wir: I benötigt höheren (asymptot.) Aufwand als II.
! Ungeachtet, daß für $n \leq 49$ I $\succ$ II.

$\Rightarrow$ Auch Antwort: Große/kleine Probleme?
Relation hier: Für kleine Werte, d.h. $n \leq 49$ ist I $\succ$ II;
für große Werte ab $n = 50$ ist II $\succ$ I.

$\Rightarrow$ Vergleich nie (i. d. R.) absolut, sondern in Abhängigkeit von Problemgröße.

### 6.2.2. Die O – und die Ω – Notation

* brauchen best. Formalismus, da es um quant. Aussagen geht;

- ein Problem P $\rightarrow$ unendlich viele Fragestellungen *p* ;
  z.B.: Eine Fragestellung des Maximumproblems wäre, die größte der vier Zahlen 13, 6, 31, 12 zu bestimmen;

- $\forall\, p \in$ P wird eine nat. Zahl *g(p)* als Größe zugeordnet (ergibt sich meist in nat. Weise $\rightarrow$ 4 Zahlen zu sortieren; ...);

- T( $p$, A) – Rechenzeitaufwand für einen Algorithmus A zur Lösung des Problems P an der Fragestellung $p \in P$;

- im konkreten Fall durch Messungen; interessanter (und objektiver) sind globale Aussagen zu beliebigen Fragestellungen der Größe $n$
  $\Rightarrow$ 3 Abstraktionen:

  (a) Verhalten im ungünstigsten Fall (worst case)
  $T^*(n, A) := sup \{ T(p, A): p \in P \text{ und } g(p) = n \}$

  (b) Verhalten im besten Fall (best case)
  $T_*(n, A) := inf \{ T(p, A): p \in P \text{ und } g(p) = n \}$

  (c) Verhalten im Mittel (average case)
  $\overline{T}(n, A) := E \{T(\underline{p}, A) \mid \underline{p} \text{ verteilt in P und konzentriert auf } g(p) = n \}$

  Klar: $\overline{T}(n, A) \leq T^*(n, A)$.

  Aber: Welche Wahrscheinlichkeitsverteilung?
  $\Rightarrow$ meist: (i) worst-case Analyse;
  (ii) average-case Analyse mit Gleichverteilung;

- bei Analyse $\rightarrow$ nur Größenordnung der Laufzeit in Abhängigkeit von der Größe der Eingabe

  für Analyse solcher oberer und unterer Schranken bzw. Wachstumsordnungen $\Rightarrow$ Groß – O und Groß – Omega – Notation

a) $f: \mathbb{N} \rightarrow \mathbb{N}$ sei

$O(f) := \{ h: \exists c_1 > 0, c_2 > 0, n_0 \geq 0$ so, daß
$h(n) \leq c_1 \cdot f(n) + c_2$ für $n \geq n_0 \}$

{für hinreichend großes $n$ ist die Fkt. $h$ nicht mehr als das $c_1$ – fache der Fkt. $f$ }

$\Rightarrow$ Sprechweise: Statt „ Die Laufzeit $\overline{T}(n, A)$ erfüllt für alle $n \in \mathbb{N}$:
$\overline{T}(n, A) \leq c_1 \cdot n + c_2$ !“
sagt man
$\overline{T}(n, A)$ ist von der Ordnung $n$ oder
$\overline{T}(n, A)$ ist $O(n)$ oder $\overline{T}(n, A)$ ist in $O(n)$;
schreibt man
$\overline{T}(n, A) = O(n)$ oder $\overline{T}(n, A) \in O(n)$;
{ Exakt wäre: $\overline{T}(n, A) \in O(f)$ mit $f(n) = n$. }

Bem.: Die Konstanten $c_1$ und $c_2$ interessieren nicht!
$\Rightarrow$ Abschätzung nach oben!

b) Abschätzung von unten $\Rightarrow$

$\Omega(g) := \{h : \exists\ c > 0$ so, daß für unendlich viele $n \in \mathbb{N}$ gilt $h(n) \geq c \cdot g(n)\}$

* Bem.: Bisher Komplexität von Algorithmen.
In Praxis auch Komplexität von Problemen:

Problem P hat (Zeit-)Komplex. $O(n^2)$ heißt, es $\exists$ ein A. zur Lösung von P, dessen Laufzeit durch eine quadratische Funktion beschränkt ist.

Klar: Für obere Schranke $\rightarrow$ 1 Algorithmus benötigt.
Für untere Schranke $\rightarrow$ muß alle beachten (schwer; wenige Fälle bisher)

* häufigsten Funktionen zur Effizienzmessung von Algorithmen:

| Wachstum | | | |
|---|---|---|---|
| logarithmisch | : | log n | (i. d. R. zur Basis 2) |
| linear | : | n | |
| n – log n | : | $n \cdot \log n$ | |
| quadratisch | : | $n^2$ | ($n^k \rightarrow$ polynomiales Wachstum) |
| exponential | : | $c^n$ | |

z. Z. allg. Überzeugung $\rightarrow$

praktikabel höchstens A. mit polynomialem Wachstum; A. mit exponentialem Wachstum sind schon für kleine Problemgrößen nicht mehr ausführbar.

## 6.2.3. Einige Prinzipien zur Berechnung der Komplexität von Algorithmen

(Lit.: U. Manber: Introduction to algorithms / Addison Wesley 1989)

* A. bestehe aus verschiedenen Teilen $\Rightarrow$ seine Komplexität ist gleich der Summe seiner Teile; nicht so einfach bei Schleifen oder Rekursion.

Bsp.: Schleife; $n$ Durchläufe; Durchlauf $i$ erfordere $i$ Operationen $\Rightarrow$
Gesamtzahl der Operationen: $1 + 2 + \ldots + n = \frac{n}{2}(n+1)$
Wie aber, wenn $i$. Durchlauf $i^2$ Operationen ?

**Lemma 6.1:** Für $s_2(n) = \sum_{i=1}^{n} i^2$ gilt $s_2(n) = \frac{n(n+1)(2n+1)}{6}$ .

Beweis: Klar: $s_2(n) \leq n^3 = n \cdot n^2$ $\Rightarrow$ Differenz ist endlich.
Prüfen diese Vermutung mittels Induktion.

Vermutung $\Rightarrow$

$s_2(n) = P(n) = an^3 + bn^2 + cn + d$ mit

$P(1) = 1$ und

$P(n+1) = P(n) + (n+1)^2$

$a(n+1)^3 + b(n+1)^2 + c(n+1) + d - an^3 - bn^2 - cn - d = n^2 + 2n + 1$

$\Rightarrow$ $3a + b - b = 1$ (für $n^2$)

$3a + 2b + c - c = 2$ (für $n$)

$a + b + c + d - d = 1$ (für 1)

$P(1) = 1 \Rightarrow a + b + c + d = 1$

$\Rightarrow$ d = 0

$\underline{\underline{a = \frac{1}{3}}}$ $\underline{\underline{b = \frac{1}{2}}}$ $\underline{\underline{c = \frac{1}{6}}}$

$\Rightarrow s_2(n) = \frac{n^3}{3} + \frac{n^2}{2} + \frac{n}{6} = \frac{n(n+1)(2n+1)}{6}$

■

$\exists$ weiteres Beweisverfahren, das allg. anwendbar:

Sei $s_3(n) = \sum_{i=1}^{n} i^3$. Nun transformieren wir die Summationsgrenzen $\rightarrow$

$$s_3(n) = \sum_{i=1}^{n} i^3 = \sum_{i=0}^{n-1} (i+1)^3 = \sum_{i=0}^{n-1} (i^3 + 3i^2 + 3i + 1)$$

gleiche Glieder weg ergibt

$$n^3 = 0^3 + \sum_{i=0}^{n-1} (3i^2 + 3i + 1) = 3 \cdot \left[ s_2(n) - n^2 \right] + 3\frac{n(n-1)}{2} + n$$

$$\Rightarrow \quad s_3(n) = \frac{n^3}{3} + n^2 - \frac{1}{2}n^2 + \frac{1}{2}n - \frac{n}{3} = \underline{\underline{\frac{n^3}{3} + \frac{n^2}{2} + \frac{n}{6}}}$$

$\Rightarrow$ Idee/Trick: Nutzen 2 spezielle Summen, und zwar so, daß sich ihre meisten Glieder gegenseitig aufheben.

* weitere Beispiele

**<u>Lemma 6.2:</u>** Für $F(n) = \sum_{i=0}^{n} 2^i$ gilt $F(n) = 2^{n+1} - 1$.

<u>Beweis:</u> $2F(n) = \sum_{i=1}^{n+1} 2^i$

$2F(n) - F(n) = F(n) = 2^{n+1} - 1$

■

**<u>Lemma 6.3:</u>** Für $G(n) = \sum_{i=1}^{n} i \cdot 2^i$ gilt $G(n) = n \cdot 2^{n+1} - 2^{n+1} + 2$.

<u>Beweis:</u>

$$G(n) = 2G(n) - G(n) = \left[ 1 \cdot 2^2 + 2 \cdot 2^3 + \ldots + n \cdot 2^{n+1} \right] - \left[ 1 \cdot 2 + 2 \cdot 2^2 + \ldots + n \cdot 2^n \right]$$

$$= n \cdot 2^{n+1} - \underbrace{(1 \cdot 2 + 1 \cdot 2^2 + \ldots + 1 \cdot 2^n)}_{\text{L. 6.2: } 2^{n+1} - 2} = (n-1) \cdot 2^{n+1} + 2$$

■

**Lemma 6.4:** Für $G(n)=\sum_{i=1}^{n} i\cdot 2^{n-i}$ gilt $G(n)=2^{n+1}-2-n$.

Beweis: $G(n)=2G(n)-G(n)=\sum_{i=1}^{n} i\cdot(2^{n+1-i}-2^{n-i})$

$=1\cdot(2^n-2^{n-1})+2\cdot(2^{n-1}-2^{n-2})+\ldots+n\cdot(2-1)$

$=2^n+2^{n+1}+\ldots+2-n=2^{n+1}-2-n$ ■

* rekursive Beziehungen:

Fibonacci → $F(n)=F(n-1)+F(n-2),\quad n>2;\quad (*)$
$F(2)=F(1)=1$

Frage: ∃ geschlossene Darstellung? Wenn ja, so viel Ressourcenersparnisse.

(1) Intelligentes „Raten“:

Fibonacci ⇒

- da $F(n)=F(n-1)+F(n-2)\Rightarrow F(n)=c\cdot 2^n$ als Vermutung
  setzen in (*) ein ⇒ $c\cdot 2^n=c\cdot 2^{n-1}+c\cdot 2^{n-2}$
  ⇒ keine Lösung, da $2^n > c\cdot 2^{n-1}+c\cdot 2^{n-2}$ für $n\geq 2$

- 2. Versuch mit anderen Exponentialfkt, aber mit kleinerer Basis → $F(n)=a^n$
  setzen in (*) ein ⇒ $a^n=a^{n-1}+a^{n-2}$ bzw. $a^2=a+1$

⇒ 2 Lösungen $\quad a_{1,2}=\frac{1}{2}\pm\sqrt{\frac{1+4}{4}}=\frac{1}{2}\pm\sqrt{\frac{5}{4}}=\frac{1}{2}\pm\frac{1}{2}\sqrt{5}$

Wenn 2 Lösungen, so auch auch deren Linearkombination:

$c_1\cdot a_1^n+c_2\cdot a_2^n$ ;

$c_1, c_2$ aus $F(2)=F(1)=1 \Rightarrow c_1=\frac{1}{\sqrt{5}};\quad c_2=-\frac{1}{\sqrt{5}}$

$$\Rightarrow \boxed{F(n)=\frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^n-\frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^n}$$

(2) Divide – and – Conquer Beziehungen:

Divide – and – Conquer Algorithmen sehr häufig und auch recht elegant;
z.B. beim Sortieren; 3 Schritte ⇒

| | |
|---|---|
| Divide – Schritt: | Input in kleine Teilmengen aufgeteilt. {Folge in 2 Teilfolgen} |
| Conquer –Schritt: | Problem für jede der Teilmengen gelöst. {Teilfolgen werden sortiert} |

Merge – Schritt: Teillös. zur Gesamtlösung zusammengemischt.
{sort. Teilfolgen zur sort. Folge}

allgemein:

Divide – Schritt $\Rightarrow$ $a$ Teilmengen(-probleme)

Conquer –Schritt $\Rightarrow$ jede hat Aufwand $\frac{n}{b}$ des Ausgangsproblems

Merge – Schritt $\Rightarrow$ $c \cdot n^k$

$$\Rightarrow T(n) = a \cdot T\left(\frac{n}{b}\right) + c \cdot n^k \quad \text{für } a, b, c, k = \text{const.}$$

vereinfach. Ann.: $n = b^m$, $b > 1$ sei Integer;

$\Rightarrow \frac{n}{b}$ ist immer Integer

sehen einige Schritte an $\rightarrow$

$$T(n) = a \cdot \left[ a \cdot T\left(\frac{n}{b^2}\right) + c\left(\frac{n}{b}\right)^k \right] + c \cdot n^k$$

$$= a \cdot \left\{ a \cdot \left[ a \cdot T\left(\frac{n}{b^3}\right) + c\left(\frac{n}{b^2}\right)^k \right] + c\left(\frac{n}{b}\right)^k \right\} + c \cdot n^k = \ldots$$

Nehmen an, daß $T(1) = c$ .
Anderenfalls würde sich Endergebnis um eine Konstante ändern.

$$\Rightarrow T(n) = c \cdot a^m + c \cdot a^{m-1} \cdot b^k + c \cdot a^{m-2} \cdot b^{2k} + \ldots + c \cdot b^{mk}$$

$$\Rightarrow T(n) = c \cdot \sum_{i=0}^{m} a^{m-i} \cdot b^{ik} = c \cdot a^m \cdot \underbrace{\sum_{i=0}^{m} \left(\frac{b^k}{a}\right)^i}$$

geom. Reihe

3 Fälle

Fall 1: $a > b^k$

$\rightarrow$ Summe konvergiert bei $m \rightarrow \infty$ :

$\Rightarrow T(n) = O(a^m)$

wegen $n = b^m \rightarrow m = \log_b n$ bzw. $a^m = a^{\log_b n} = n^{\log_b a}$

$\Rightarrow T(n) = O(n^{\log_b a})$

Fall 2: $a = b^k$

$\rightarrow$ Summe ist gleich $m + 1$

$\Rightarrow T(n) = O(a^m \cdot m)$

wegen $a = b^k$ gilt $\log_b a = k$ und $m = O(\log n)$

$\Rightarrow T(n) = O(n^k \cdot \log n)$

Fall 3: $a < b^k$

$$\rightarrow \text{Summe ist } \frac{\left(\frac{b^k}{a}\right)^{m+1} - 1}{\left(\frac{b^k}{a} - 1\right)}$$

$$\Rightarrow T(n) = \mathrm{O}\left(a^m \cdot \left(\frac{b^k}{a}\right)^m\right) = \mathrm{O}\left(b^{k \cdot m}\right) = \mathrm{O}\left(n^k\right) \quad \Rightarrow$$

## Satz 6.1:

Für die Lösung der Rekurrenzgleichung

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + c \cdot n^k$$

mit ganzz. Konstanten $a \geq 1$ und $b \geq 1$ sowie Konstanten $c, k > 0$ gilt

$$T(n) = \begin{cases} \mathrm{O}(n^{\log_b a}) & \text{bei} \quad a > b^k; \\ \mathrm{O}(n^k \log n) & \text{bei} \quad a = b^k; \\ \mathrm{O}(n^k) & \text{bei} \quad a < b^k. \end{cases}$$

Bem.: $a = 2b \Rightarrow T(n) = \mathrm{O}(n^2)$!

Anwendungsbeispiele:

1***) Sortieren*** von $n$ Elementen (Namen, Zahlen, ...)

Algorithmus *Sortiere* (*Liste*);
IF $n > 1$
THEN BEGIN
*Sortiere* (*1. Listenhälfte*);
*Sortiere* (*2. Listenhälfte*);
*Mische* (*1. und 2. Listenhälfte*)
END;

$$\Rightarrow \quad T(n) = 2 \cdot T\left(\frac{n}{2}\right) + \underbrace{c \cdot n}_{\text{Aufwand für Mischen } \sim n} \quad \text{mit } T(1) = c$$

$\rightarrow$ $a = 2$; $b = 2$; $c = c$; $k = 1$

$\Rightarrow$ $\boxed{T(n) = \mathrm{O}(n \cdot \log n)}$

2) ***Multiplikation*** von zwei n-stelligen Zahlen

Algorithmus 1:

| 1 2 3 | · | 4 5 6 |
|---|---|---|
| | 4 9 2 | |
| | 6 1 5 | |
| | 7 3 8 | |
| | 5 6 0 8 8 | |

$\Rightarrow$ *n* Zeilen zu add.; jede Zeile *n* oder *n*+1 Ziffern;
je Zeile *n* Multiplikationen

$\hookrightarrow$ $n \cdot n = n^2$ Operationen $T(n, \text{A1}) = \text{O}(n^2)$

Algorithmus 2: Suchen einen besseren A.!
Gehen von folgender Betrachtung aus $\rightarrow$

$$\overbrace{A \quad B}^{n} \times \overbrace{C \quad D}^{n} =$$
$$= 10^{\frac{n}{2}} \cdot A \times \left( 10^{\frac{n}{2}} C + D \right) + B \times \left( 10^{\frac{n}{2}} C + D \right) \quad \Rightarrow$$

$$\left.\begin{array}{ll} 10^{n}: & A \times C \\ 10^{\frac{n}{2}}: & A \times D + B \times G \\ 10^{0}: & B \times D \end{array}\right\} \; 4 \text{ Mult.} + 4 \text{ Add.}$$

$$T(n) = 4 \cdot T\left(\frac{n}{2}\right) + c \cdot n = \text{O}(n^2) \qquad \Rightarrow \text{Fehlanzeige!}$$

Aber:

$10^{\frac{n}{2}}: \; (A+B) \cdot (C+D) - AC - BD$

$\Rightarrow$ 3 Multiplikationen + 5 Add.

$$T(n) = 3 \cdot T\left(\frac{n}{2}\right) + c \cdot n = \text{O}(n^{\log_2 3}) = \text{O}(n^{1.58496}) \qquad \{\text{Bem.: } \log_b a = \frac{\ln a}{\ln b}\}$$

| 1 2 3 · 4 5 6 | $\Rightarrow$ |
|---|---|
| $A \cdot G$: | 400 |
| $(A+B) \cdot (C+D) - AC - BD$: | 148 |
| $B \cdot D$: | 1288 |
| | 56088 |

Bem.: *n* muß gerade sein $\rightarrow$ 0123 · 0456 $\Rightarrow$ n = 4 und n/2 = 2

$\Rightarrow$ nach $A \cdot G = 4$ alle folg. Zeilen um $2 = \frac{n}{2}$ Stellen nach hinten

## 6.3. Suchalgorithmen

* Suche nach best. Datum / Daten → häufige Operation in Programmen

Suchverfahren := A., der für einen best. Schlüssel (das Suchargument) die zugehörige Inf., den eigentlich interess. Inhalt ermittelt, sofern ein Obj. mit diesem Schlüssel ex.; anderenfalls wird die Suche erfolglos abgebrochen.

formale Def. des Suchproblems:

Gegeben sei eine Folge $S = (s_1, s_2, ..., s_n)$ von $n$ Objekten $s_1$ bis $s_n$. Jedes Objekt besteht aus einem Schlüssel (Namen) $s_i \cdot key$ und bestimmten Informationen, $i = 1(1)n$.

Die Schlüssel stammen aus einer Vielfalt $U$.
Gegeben ist weiter ein Element $z \in U$.
Gesucht ist ein $i_0$ derart, daß $s_{i_0} \cdot key = z$ .

### * *lineare Suche*:

Keine Information über Ordnung der Schlüssel ⇒ sequentielles Durchlaufen der Folge

$$\Rightarrow T^*(n) = n;\ T_*(n) = 1;\ \overline{T}(n) = \frac{n+1}{2}$$

Im weiteren: $U = \mathbb{R}^1$ mit lin. Ordnung. Schreiben $x_i$ statt $s_i \cdot key$ .

### * *binäre Suchverfahren*:

binäre Suche ist für Algorithmierung, was das Rad in der Mechanik;

Grundidee: Durch 1 Frage ca. ½ des Suchraumes verwerfen.

a) Reine binäre Suche:

Problem:

Sei $S = (x_1, ..., x_n)$ eine Folge reeller Zahlen mit der Eigenschaft $x_1 \le x_2 \le ... \le x_n$ . Zu gegebenem reellem $z$ ist ein Index $i$ mit $x_i = z$ auszugeben, wenn $z$ in der Folge ist, sonst ist $0$ auszugeben.

$$\Rightarrow i = \min \{k:\ x_k = z\} \vee 0 ;$$

Als Grundidee ⇒ sehe ein „mittleres“ El. $x_{\lceil n/2 \rceil}$ oder $x_{\lceil n/2 \rceil + 1}$ an;

$$\left.\begin{array}{l}\text{ist } z \text{ kleiner, so alle } x_i \text{ mit } i \geq \lceil n/2 \rceil \text{ weg;} \\ \text{ist } z \text{ gleich, so Ende } \{z := \lceil n/2 \rceil\}; \\ \text{ist } z \text{ größer, so alle } x_i \text{ mit } i \leq \lceil n/2 \rceil \text{ weg;}\end{array}\right\} \Rightarrow$$

Algorithmus ***binäre_Suche***(X, n, z);

Input: X – sortiertes Feld mit Komponenten x[1] bis x[n]
z – Suchschlüssel

Output: *Position* – Index *i* mit x[*i*] = z oder 0, wenn solch Index nicht existiert

```
BEGIN
    Position := Finde(z, 1, n)
END;

FUNCTION Finde(e, Links, Rechts): Integer;
VAR Mitte: Integer;
  BEGIN
     IF Links = Rechts
          THEN  IF  x[Links] = z  THEN   Finde := Links
                                    ELSE    Finde := 0
          ELSE  BEGIN
                     Mitte := ⌈ 1/2 * (Links+Rechts) ⌉;
                     IF  z < x[Mitte]
                          THEN  Finde := Finde(z, Links, Mitte-1)
                          ELSE   Finde := Finde(z, Mitte, Rechts)
                END
  END;
```

Komplexität: Basisoperation sei ein Vergleich $\Rightarrow$

$$T(n) = 1 + T\left(\frac{n}{2}\right) \quad \text{mit } T(1) = 1$$

Satz 6.1. $\Rightarrow$ $a = 1 \quad b = 2 \quad c = 1 \quad k = 0 \quad \Rightarrow a = b^k$

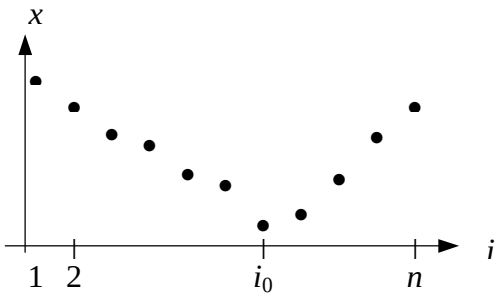
$\Rightarrow$ $\boxed{T(n) = O(\log n)}$

b) Binäre Suche des Minimums einer unimodalen Folge:

Problem:

Sei $S = (x_1, ..., x_n)$ derart, daß $x_1 > x_2 > ... > \underline{\underline{x_{i_0}}} < x_{i_0+1} < ... < x_{n-1} < x_n$.
für ein $i_0$. Bestimme den Index $i_0$.

unimodale Folge z.B. $\rightarrow$




wichtig für viele Minimierungsprobleme, z.B. folg. Lagerhaltungsproblem:

- 1 Lager, zufälliger. Bedarf entsprechend Verteilungsgesetz

$$B = \begin{pmatrix} 0 & 1 & \dots & n \\ p_0 & p_1 & \dots & p_n \end{pmatrix}$$

- Vorrat $x \in \mathbb{N}$

- Kosten: $k(x,b) = \begin{cases} h \cdot (x-b), & x \geq b; \\ g \cdot (b-x), & x < b; \end{cases}$

Kostenerwartungswert

$$K(x) := E[k(x,B)] = \sum_{b=0}^{n} p_b \cdot k(x,b) =$$

$$= h \cdot \sum_{b=0}^{n} p_b \cdot (x-b) + g \cdot \sum_{b=x+1}^{n} p_b \cdot (b-x)$$

- gesucht: $x^*$ mit $K(x^*) \leq K(x)$ für $x \in \mathbb{N}$

Lösung 1:

Berechne $K(x)$ für $x = 1 .. n$ und wähle $x^*$.

Lösung 2:

Binäre Suche nach folg. Grundidee:

berechne Fkt.-werte für „mittl." El. $x_{\lfloor n/2 \rfloor}$ und $x_{\lfloor n/2 \rfloor + 1}$;

ist $K\left(x_{\lfloor n/2 \rfloor}\right) < K\left(x_{\lfloor n/2 \rfloor + 1}\right)$, so ist Min. in $[1,...,\lfloor n/2 \rfloor]$, also links von $x_{\lfloor n/2 \rfloor + 1}$;

ist $K\left(x_{\lfloor n/2 \rfloor}\right) > K\left(x_{\lfloor n/2 \rfloor + 1}\right)$, so rechts von $x_{\lfloor n/2 \rfloor}$;

*Komplexität* : Basisoperation = Berechnung eines Funktionswertes;

➔ $T(n) = 2 + T(n/2) \quad \rightarrow \quad a=1 \quad b=2 \quad c=2 \quad k=0 \quad \rightarrow \quad a = b^k$

➔ $T(n) = O(\log n)$

* Bsp.: Lagerhaltungsproblem

$n = 10$ $p_b = 1/10$ für $b=1(1)10$ $h = 1$ $g = 3$

➔

| x | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| K(x) | 16.5 | 13.5 | 10.9 | 8.7 | 7.0 | 5.5 | 4.5 | 3.9 | 3.7 | 3.9 | 4.5 |

Bei binären Suche →

L = 0, R = 10 → Mitte = 5 , so daß
K(5) = 5.5 und K(6) = 4.5 zu berechnen sind
K(5) > K(6) ➔

L = 6, R = 10 → Mitte = 8 , so daß
K(8) = 3.7 und K(9) = 3.9 zu berechnen sind
K(8) < K(9) ➔

L = 6, R = 8 → Mitte = 7 , so daß
K(7) = 3.9 zu berechnen ist
K(7) > K(8) ➔

L = R = 8 ➔ Minimum wird unter x = 8 erreicht
➔ 5 Funktionswerte statt 11

**Bem.**:
Analoges Vorgehen bei Bestimmung des Minimums (Maximums) einer Funktion oder des Null-Punktes einer stetigen Funktion und bei ähnlichen Problemen.

Algorithmus *binäre_Suche*(X, n, z);

Input: X – sortiertes Feld mit Komponenten x[1] bis x[n]
z – Suchschlüssel

Output: *Position* – Index *i* mit x[*i*] = z oder 0, wenn solch Index nicht existiert

```
BEGIN
      Position := Finde(z, 1, n)
END;

   FUNCTION Finde(e, Links, Rechts): Integer;
   VAR Mitte: Integer;

     BEGIN
        IF Links = Rechts

              THEN  IF  x[Links] = z  THEN   Finde := Links
                                      ELSE    Finde := 0

              ELSE  BEGIN

                          Mitte := ⌊1/2 * (Links+Rechts)⌋;

                          IF  z <= x[Mitte]

                                THEN  Finde := Finde(z, Links   , Mitte   )

                                ELSE   Finde := Finde(z, Mitte+1, Rechts)
                    END
     END;
```