Gütekriterien bei Algorithmen

1. **Effizienz**
   Verlangt werden Effizienz bzgl. des zeitlichen Aufwands und des Speicherbedarfs während der Laufzeit; beide Forderungen sind häufig nicht gleichzeitig erfüllbar.

2. **Korrektheit**
   Das Programm liefert die Lösung eines Problems entsprechend seiner Spezifikation, in der die Eingabedaten und die Ausgabedaten vorgeschrieben werden.

3. **Zuverlässigkeit**
   Ein zuverlässiges Programm korrigiert Fehler infolge falscher Anwendung oder falscher oder sinnloser Eingabe.

4. **Wartungsfreundlichkeit**
   Ein wartungsfreundliches Programm läßt sich leicht ändern, korrigieren oder erweitern (wichtig für upgrades!); die Wartungsfreundlichkeit setzt allerdings eine entsprechende Dokumentation des Quelltextes voraus.

5. **Benutzerfreundlichkeit**
   Der Anwender kann ohne Konsultation des Programmautors oder eines Handbuchs mit dem Programm erfolgreich umgehen; diese Fertigkeit wird selbstverständlich auch unterstützt von der Intuition und Erfahrung des Anwenders.

1. Effizienz

Sei **n := Anzahl der Datensätze**, die der Algorithmus zu verarbeiten hat

Algorithmus	**Sortieren durch direkte Auswahl**	**Sortieren durch Mischen** (mergesort)	**Türme von Hanoi**	**Erfassen von Adressen**	**Suchen in einer sortierten Liste**
Anzahl der Rechenoperationen und damit zeitlicher Bedarf zur Laufzeit des Programms proportional zu	n^2	$n \cdot \log_2(n)$	$2^n - 1$	n	$\log_2(n)$
Art des Wachstums	**polynomial**		**exponentiell**	**linear**	**logarithmisch**

Algorithmen, deren zeitlicher Aufwand exponentiell oder stärker als exponentiell (Ackermann-Funktion!) anwächst, sind in der Praxis unbrauchbar.

2. Korrektheit

Jeder Programmierer macht die Erfahrung, daß ein Programm weder bezüglich der Syntax der verwendeten Programmiersprache noch bezüglich der erwarteten Verarbeitung der Daten auf Anhieb korrekt ist.

Insbesondere gilt dies für überaus komplexe Programme wie Betriebssysteme (winXP oder win2k3; die alten winDOS-Systeme (win3.11, win95, win98, winME) erwiesen sich als besonders unzuverlässig).

In einigen Fällen, leider beschränkt auf vergleichsweise einfache Algorithmen, läßt sich sogar ein mathematischer Beweis für die Korrektheit eines Algorithmus erbringen, indem man **Schleifeninvarianten** findet und diese als korrekt verifiziert. Das hierzu benötigte Beweisverfahren ist das Verfahren der **Vollständigen Induktion** (Die Mathematik kennt bekanntlich drei Beweisverfahren: direkter Beweis, indirekter Beweis, vollständige Induktion).

Verfahren der **Vollständigen Induktion:**

Sei **A(n)** eine von der natürlichen Zahl **n** abhängige **Aussage**, $n \in \{0, 1, 2, 3, \ldots\}$.

Um zu beweisen, daß **A(n)** wahr ist für alle $n \in \{0, 1, 2, 3, \ldots\}$, verifizieren wir:

(1) **A(0) ist wahr** (Induktionsanfang)

(2) **Die Implikation [A(n) $\Rightarrow$ A(n+1)] ist wahr** (Induktionsschritt)

Bevor wir dieses Verfahren auf den Korrektheitsbeweis von Algorithmen anwenden, sollten wir es bei einfachen innermathematischen Problemen einüben und verstehen.

Aufgabe 1:

Behauptung: $1^2 + 2^2 + 3^2 + \ldots\ldots + n^2 = n(n+1)(2n+1)/6$

Beweis:

Definiere **A(n) := „$1^2 + \ldots\ldots + n^2 = n(n+1)(2n+1)/6$“**

(Beachte: A(n) ist eine Gleichung, somit insbesondere eine Aussage, die genau zwei boolesche Werte annehmen kann: TRUE oder FALSE.)

Induktionsanfang (n=1):

A(1)=TRUE ,
denn **A(1)** $\Leftrightarrow$ **[** $1^2 = 1\cdot(1+1)(2\cdot1+1)/6$ **]** $\Leftrightarrow$ **[** $1 = 1\cdot2\cdot3/6$ **]** $\Leftrightarrow$ **[1=1]**
die letzte Aussage hat trivialerweise den Wert TRUE.

Induktionsschritt:

Unter der Annahme, daß A(n) TRUE ist, verifizieren wir, daß dann auch A(n+1) den Wert TRUE annimmt.

Sei also A(n) TRUE, das heißt

$\mathbf{1^2 + 2^2 + 3^2 + ... + n^2 = n(n+1)(2n+1)/6}$ ist richtig für beliebiges n (diese Annahme heißt auch **Induktionsvoraussetzung**).

Wir betrachten A(n+1), also die Gleichung

$\mathbf{1^2 + 2^2 + ... + (n + 1)^2 = (n + 1)[(n+1) + 1][2(n+1) + 1]/6,}$

die wir unter der Annahme, daß A(n) TRUE ist, als TRUE qualifizieren werden.

$$\mathbf{1^2 + 2^2 + ... + (n + 1)^2 = [1^2 + 2^2 + ... + n^2] + (n + 1)^2}$$

wegen A(n) = TRUE folgt

$$\begin{aligned}
&= \mathbf{n(n+1)(2n+1)/6 + (n + 1)^2} \\
&= \mathbf{(n + 1)[n(2n+1)/6 + (n + 1)]} \\
&= \mathbf{(n + 1)[n(2n+1) + 6(n + 1)]/6} \\
&= \mathbf{(n + 1)[2n^2+n + 6n + 6)]/6} \\
&= \mathbf{(n + 1)[2n^2+ 7n + 6)]/6} \\
&= \mathbf{(n + 1)[(n + 2)(2n + 3)]/6} \\
&= \mathbf{(n + 1)[(n+1) + 1][2(n+1) + 1]/6}
\end{aligned}$$

Somit folgt unter der Annahme „A(n)=TRUE", daß „A(n+1)=TRUE" wahr ist, und in Verbindung mit dem Induktionsanfang „A(1)=TRUE" ergibt sich die Behauptung für alle Werte von n.

Als Übungsaufgabe verifiziere man die Behauptungen der Aufgaben 2 und 3:

Aufgabe 2:

Behauptung: $\mathbf{1^3 + 2^3 + 3^3 + ... + n^3 = n^2(n+1)^2/4}$

Aufgabe 3:

Behauptung: Die Bernoullische Ungleichung $\mathbf{(1 + x)^n > 1 + n \cdot x}$

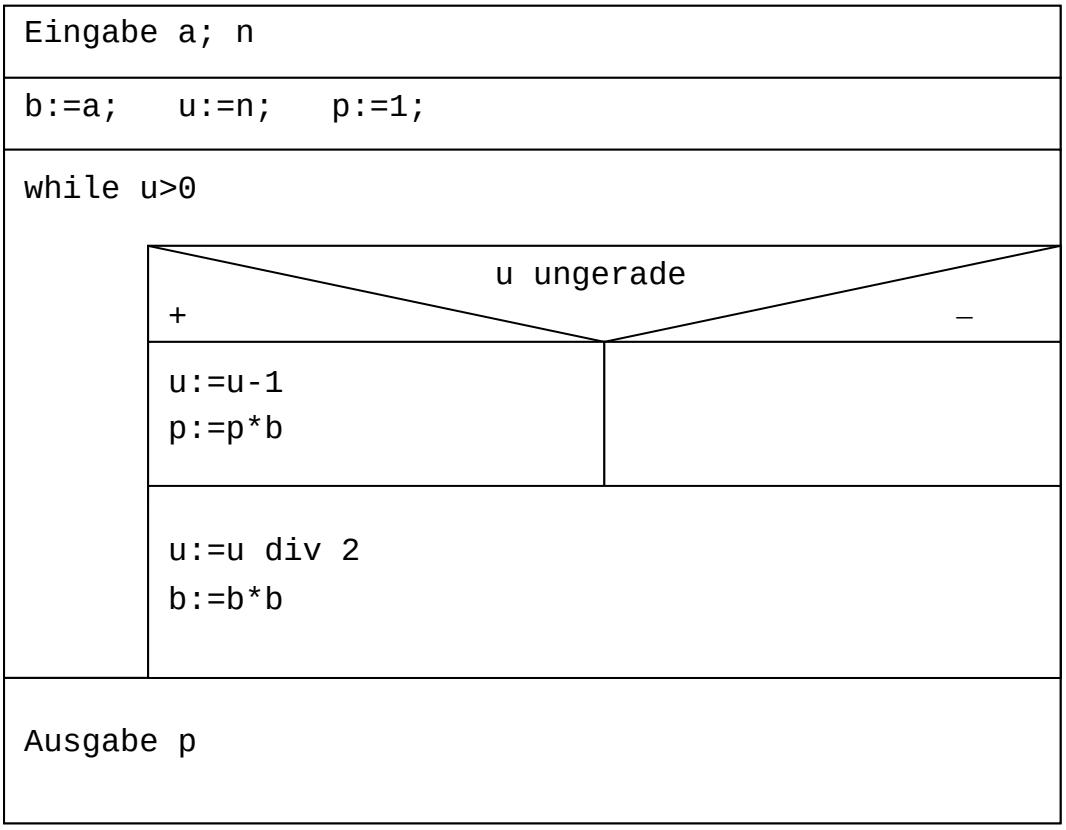
ist wahr für alle natürlichen Zahlen n mit $n \geq 2$ und für reelle Zahlen x mit $x \neq 0$ und $1+x>0$.

(Vgl. auch das Mathematikbuch; man sieht, daß Informatik und Mathematik durchaus verwandte Wissenschaften sind, was man auch nicht anders vermutet hätte.)

Korrektheitsbeweise bei Algorithmen

1. Der Algorithmus elmo

Seien n eine natürliche Zahl, a eine von 0 verschiedene reelle Zahl.
Gegeben ist folgender Algorithmus als Struktogramm:



Aufgaben:

a) Codiere den Algorithmus in Pascal (oder einer anderen Hochsprache).

b) Teste das Programm; was bewirkt der Algorithmus vermutlich?

c) Die Vermutung läßt sich anhand eines ***Trace*** erhärten; finde eine Beziehung, die sich als Schleifeninvariante erweisen könnte.

d) Beweise vermöge vollständiger Induktion, daß die in c) gefundene Beziehung tatsächlich Schleifeninvariante ist, und schließe daraus, daß die in b) aufgestellte Vermutung, was der Algorithmus bewirkt, richtig ist.

Lösungen:

zu a):

```
program elmo;
uses crt;
var  n,u  :longint;
     a,b,p:real;


begin
  clrscr;

  { Eingabe der Werte für a und n }
  write('a = '); readln(a);
  write('n = '); readln(n);

  { Initialisierung der Variablen }
  b:=a;
  u:=n;
  p:=1;

  { Verarbeitung der Daten }
  while u>0 do begin
                 if odd(u) then begin
                                  u:=u-1;
                                  p:=p*b
                                end;
                 u:=u div 2;
                 b:=sqr(b)
               end;

  { Ausgabe }
  writeln;
  write ('p = ',p);
  while not keypressed do

end.
```

zu b): Kompiliere den Quelltext und führe das Programm aus.

zu c):

Empirisches Testen des Programms anhand eines Trace

Vereinbarung: S.D. = Schleifendurchlauf

Seien n eine natürliche Zahl, a eine von 0 verschiedene reelle Zahl.

α) Trace für n=7:

	n	**a**	**b**	**u**	**p**	**u = 0**
vor dem **1.** S.D.	7	a	a	7	1	–
vor dem **2.** S.D.	7	a	a^2	3	a	–
vor dem **3.** S.D.	7	a	a^4	1	a^3	–
nach dem **3.** S.D.	7	a	a^8	0	a^7	+

β) Trace für n=18:

	n	**a**	**b**	**u**	**p**	**u = 0**
vor dem **1.** S.D.	18	a	a	18	1	–
vor dem **2.** S.D.	18	a	a^2	9	1	–
vor dem **3.** S.D.	18	a	a^4	4	a^2	–
vor dem **4.** S.D.	18	a	a^8	2	a^2	–
vor dem **5.** S.D.	18	a	a^{16}	1	a^2	–
nach dem **5.** S.D.	18	a	a^{32}	0	a^{18}	+

γ) Trace für n=52:

	n	**a**	**b**	**u**	**p**	**u = 0**
vor dem **1.** S.D.	52	a	a	52	1	–
vor dem **2.** S.D.	52	a	a^2	26	1	–
vor dem **3.** S.D.	52	a	a^4	13	1	–
vor dem **4.** S.D.	52	a	a^8	6	a^4	–
vor dem **5.** S.D.	52	a	a^{16}	3	a^4	–
vor dem **6.** S.D.	52	a	a^{32}	1	a^{20}	–
nach dem **6.** S.D.	52	a	a^{64}	0	a^{52}	+

Vermutung:

Die Beziehung

$$\mathbf{p \cdot b^u = a^n}$$

ist vor und nach jedem Schleifendurchlauf erfüllt, also invariant gegenüber Schleifendurchläufen. Eine solche Gleichung heißt auch ***Schleifeninvariante***.

Der Algorithmus bricht ab, sobald u den Wert 0 hat; da u bei jedem Schleifendurchlauf um 1 vermindert wird, falls u ungerade ist, in jedem Fall aber durch 2 dividiert wird, ist die Abbruchbedingung nach endlich vielen Schleifendurchläufen mit Sicherheit erfüllt.

Für u=0 schreibt sich die Schleifeninvariante:

$$\mathbf{p \cdot b^0 = a^n}$$

$$\Leftrightarrow \qquad \mathbf{p = a^n}$$

Damit ist gezeigt, daß bei Abbruch des Algorithmus die Zahl $\mathbf{a^n}$ ausgegeben wird, falls die Beziehung $\mathbf{p \cdot b^u = a^n}$ sich als Schleifeninvariante erweist.

Zu d):

Wir führen den Beweis vermöge vollständiger Induktion über den Index i, der den i-ten Schleifendurchlauf bezeichnet.

Mit $\mathbf{p_i}$, $\mathbf{b_i}$ und $\mathbf{u_i}$ bezeichnen wir die Werte der Variablen **p** , **b** und **u** vor dem i-ten Schleifendurchlauf.

Induktionsanfang (i=1):

Wegen $p_1 = 1$, $b_1 = a$ und $u_1 = n$ gilt:

$p_1 \cdot b_1^{u_1} = 1 \cdot a^n = a^n$, somit ist die Beziehung $\mathbf{p \cdot b^u = a^n}$ für i=1 erfüllt.

Induktionsschritt:

Wir nehmen an, daß die Beziehung $\mathbf{p \cdot b^u = a^n}$ vor dem i-ten Schleifendurchlauf erfüllt ist, daß somit gilt:

$$p_i \cdot b_i^{u_i} = a^n \quad (*)$$

Wir werden verifizieren, daß unter dieser Annahme (*) die Beziehung $\mathbf{p \cdot b^u = a^n}$ auch nach dem (i + 1)-ten Schleifendurchlauf erfüllt ist.

Dazu drücken wir die Werte $\mathbf{p_{i+1}}$, $\mathbf{b_{i+1}}$ und $\mathbf{u_{i+1}}$ der Variablen **p** , **b** und **u** durch die Werte $\mathbf{p_i}$, $\mathbf{b_i}$ und $\mathbf{u_i}$ aus. Da die Eigenschaft von u, gerade oder ungerade zu sein, auf die Berechnung der neuen Werte von **p** , **b** und **u** Einfluß hat, müssen wir eine Fallunterscheidung vornehmen:

α. u sei ungerade vor dem i-ten Schleifendurchlauf, also odd(u_i) = TRUE

$$p_{i+1} = p_i \cdot b_i \quad \Leftrightarrow \quad p_i = p_{i+1} / b_i$$
$$b_{i+1} = b_i \cdot b_i \quad \Leftrightarrow \quad b_i = \sqrt{b_{i+1}}$$
$$u_{i+1} = (u_i - 1) / 2 \quad \Leftrightarrow \quad u_i = 2 \cdot u_{i+1} + 1$$

Wenn wir in die Gleichung (*) die für p_i , b_i und u_i erhaltenen Werte einsetzen, folgt:

$$(p_{i+1} / b_i) \cdot (\sqrt{b_{i+1}})^{(2 \cdot u_{i+1} + 1)} = (p_{i+1} / \sqrt{b_{i+1}}) \cdot (\sqrt{b_{i+1}})^{(2 \cdot u_{i+1} + 1)}$$
$$= p_{i+1} \cdot b_{i+1}^{u_{i+1}}$$

β. u sei gerade vor dem i-ten Schleifendurchlauf, also odd(u_i) = FALSE

Übungsaufgabe!

2. Der Algorithmus merlin

x und **y** seien natürliche Zahlen mit **x** ≥ **0** und **y > 0**.
Gegeben ist folgender Algorithmus als Struktogramm:

Eingabe x; y
q:=0; r:=x;
while r ≥ y
q:=q + 1
r:=r – y
Ausgabe q
Ausgabe r

Aufgaben:

a) Codiere den Algorithmus in Pascal (oder einer anderen Hochsprache).

b) Teste das Programm; was bewirkt der Algorithmus vermutlich?

c) Läßt sich der Algorithmus auch mit einer repeat-Schleife formulieren?

d) Die Vermutung aus b) läßt sich anhand eines ***Trace*** erhärten; finde eine Beziehung, die sich als Schleifeninvariante erweisen könnte.

e) Beweise vermöge vollständiger Induktion, daß die in d) gefundene Beziehung tatsächlich Schleifeninvariante ist, und schließe daraus, daß die in b) aufgestellte Vermutung, was der Algorithmus bewirkt, richtig ist.

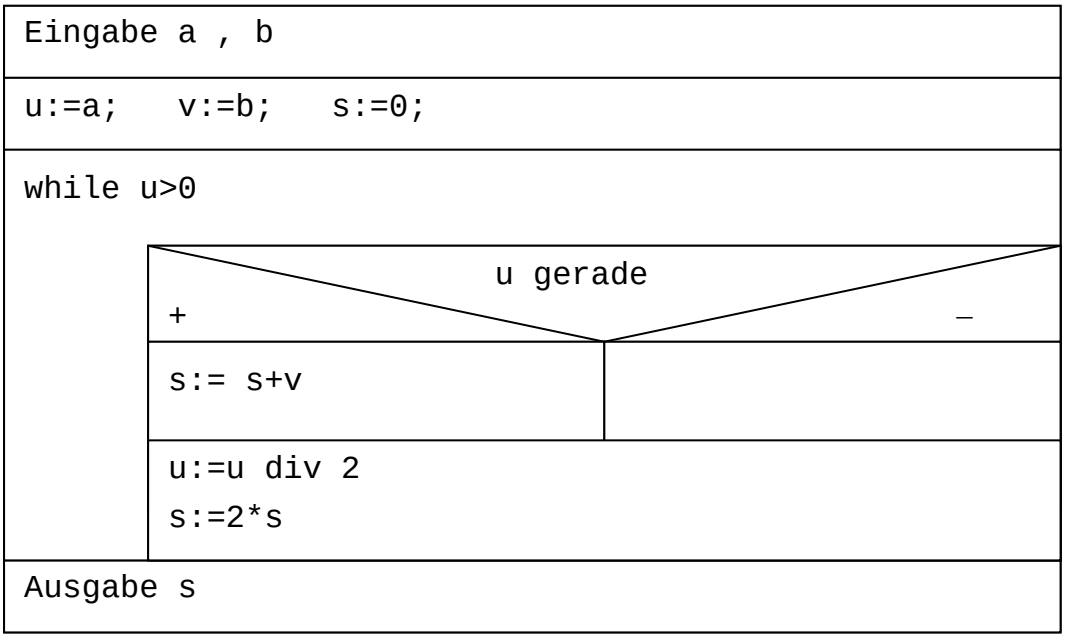
3. Den Potenzierungsalgorithmus „elmo“ kann man modifizieren, indem man die while-Schleife durch folgende Befehlssequenz ersetzt:

```
while u>0 do begin
               while not odd(u) do begin
                                     u:=u div 2;
                                   b:=b*b
                                   end;
               u:=u–1;
               p:=P*b
             end;
```

a) Integriere diese Befehlssequenz in den vorhandenen Programmtext und teste das Programm empirisch.

b) Beweise die Korrektheit des auf diese Weise modifizierten Algorithmus!

4. In einem Buch ist das Struktogramm des folgenden Algorithmus abgedruckt, von dem behauptet wird, daß er das Produkt der natürlichen Zahlen a und b berechne (dieses – im übrigen nicht schlechte – Buch gibt’s tatsächlich!):



a) Verifizieren anhand eines Trace (oder indem man das Pascal-Programm schreibt und dieses testet), daß der Algorithmus das verlangte nicht leistet.

b) Korrigiere den Algorithmus, so daß er korrekt im Sinne der Spezifikation arbeitet; beweise dessen Korrektheit vermöge vollständiger Induktion.