Sortieren durch Mischen ("MergeSort")

Aufgabe:

Gegeben ist eine Liste **L = {a[0], a[2], a[3], ... , a[n-1]}**
von n Datenelementen, für die die Ordnungsrelationen < , > , ≤, ≥ erklärt sind. Die Inhalte dieser Datenelemente sind so anzuordnen, daß gilt:

a[0] ≤ a[2] ≤. . . . . ≤ a[n-1] .

Wir fassen die Elemente der Liste auf als Komponenten eines arrays a.

Strategie: "Divide et impera"

Eine Liste, die nur ein einziges Element enthält, ist bereits sortiert.

Die Aufgabe, die n-elementige Liste (n > 1) zu sortieren, läßt sich in 4 Schritten bewältigen:

1. **Teile die n-elementige Liste in zwei etwa gleichlange Teillisten**
2. **Sortiere die erste Teilliste gemäß den Schritten 1). - 4).**
3. **Sortiere die zweite Teilliste gemäß den Schritten 1). - 4).**
4. **Mische die sortierten Teillisten zu einer sortierten Gesamtliste**

Falls **`left < right`** wahr ist, sortiert die rekursiv definierte Funktion

`sort(array, left, right)`

die Liste

`array[left], . . . . , array[right]`

unter Verwendung der Funktion **`merge`**.

Die Funktion

`merge(array, left, middle, right)`

mischt die sortierten Teillisten

`array[left], . . . . , array[middle]`

und

`array[middle+1], . . . . , array[right]`

zu der sortierten Gesamtliste

`array[left], . . . . , array[right]` .

Quellcode der Funktion **sort** in Python:

```
def sort(array, left, right):
     if left >= right:
          return
     middle = (left + right)//2
     sort(array, left, middle)
     sort(array, middle + 1, right)
     merge(array, left, middle, right)
```

Aufruf zum Sortieren der aus den **n** Komponenten

```
a[0], a[2], a[3], . . . . , a[n-1]
```

bestehenden Liste a:

```
sort(a, 0, len(a)-1)
```

Aufwandsbetrachtung:

Mit **A(n)** werde der Aufwand (die Anzahl elementarer Verarbeitungsschritte wie z. B. Additionen, Wertzuweisungen, Vergleichsoperationen) bezeichnet, eine aus n Komponenten bestehende Liste zu sortieren.

Dann gilt:

A(n) = 2 x Aufwand zum Sortieren einer Teilliste mit n/2 Elementen
+ Aufwand zum Mischen zweier sortierter Teillisten

A(n) = A(n/2) + A(n/2)
+ Aufwand zum Mischen zweier sortierter Teillisten

Der Aufwand zum Mischen zweier sortierter Teillisten zu einer sortierten Gesamtliste wächst linear mit der Anzahl **n** der zu sortierenden Datenelemente; somit erhalten wir für den Funktionsterm **A(n)** die Funktionalgleichung (c = Konstante = Proportionalitätsfaktor)

(*) **A(n) = A(n/2) + A(n/2) + c · n** mit der Bedingung
(**) **A(1) = 0** .

Behauptung: Die Funktion

$$\mathbf{A(n) = c \cdot n \cdot \log_2(n)}$$

ist Lösung der Funktionalgleichung (*) mit der Anfangsbedingung (**).

Beweis:

$$\begin{aligned} A(n/2) + A(n/2) + c \cdot n &= 2 \cdot A(n/2) + c \cdot n \\ &= 2 \cdot c \cdot n/2 \cdot \log_2(n/2) + c \cdot n \\ &= c \cdot n \cdot (\log_2(n) - \log_2(2)) + c \cdot n \\ &= c \cdot n \cdot (\log_2(n) - 1) + c \cdot n \\ &= c \cdot n \cdot \log_2(n) \\ &= A(n) \end{aligned}$$

Damit ist (*) erfüllt; wegen $\log_2(1) = 0$ genügt A(n) auch der Bedingung (**).

Bemerkung: Mit Methoden der Analysis läßt sich die Eindeutigkeit der Lösung des Problems (*), (**) zeigen, somit ist mit $A(n) = c \cdot n \cdot \log_2(n)$ die einzige Lösung der Funktionalgleichung gefunden.

Allgemein läßt sich beweisen, daß der Aufwand zum Sortieren von n Datensätzen grundsätzlich mindestens von der Ordnung $n \cdot \log_2(n)$ wächst. In diesem Sinne kann das Sortierverfahren „MergeSort" als optimales Vefahren gelten.

Ergänzende Betrachtung zum Speicherplatzbedarf:

Nachdem wir festgestellt haben, daß der Aufwand zum Sortieren von n Datenelementen von der Ordnung $n \cdot \log_2(n)$ wächst und damit ein Optimum erreicht ist, erhebt sich die Frage, ob dieser Vorteil durch die zwar elegante, aber rekursive Formulierung des Sortieralgorithmus nicht aufgehoben wird; denn rekursive Algorithmen haben grundsätzlich den Nachteil, daß sie während der Laufzeit mehr Arbeitsspeicher beanspruchen als iterative. Daß dieser Effekt bei MergeSort nicht oder nur unwesentlich ins Gewicht fällt, zeigt folgende Überlegung:

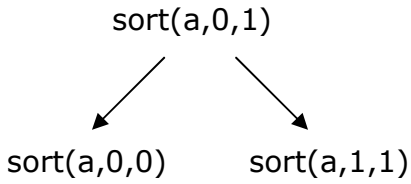
Mit **f(n)** bezeichnen wir die Anzahl der gleichzeitig aktiven Aufrufe der Funktion **`sort`**, wenn eine Liste mit **`n`** Datenelementen zu sortieren ist.

O. B. d. A. sei n eine Zweierpotenz, d. h. $n=2^k$, $k \in \{0, 1, 2, 3, \ldots\ldots\}$.

Bemerkung: Der Pfeil ⟶ *bedeutet: „ruft auf"*

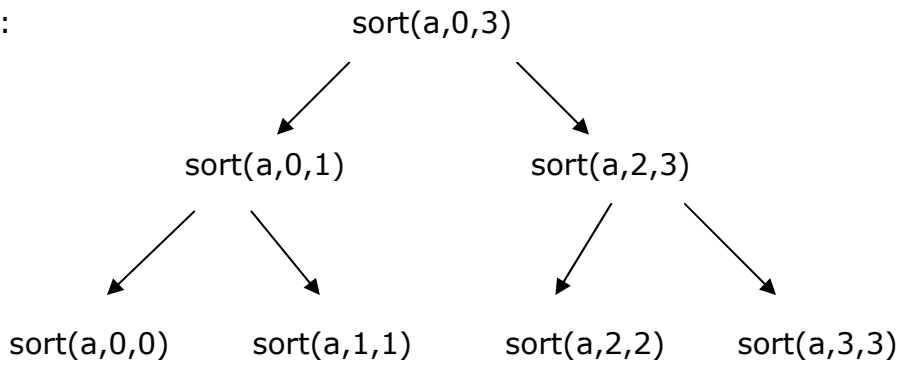
n = 1: sort(a,0,0) 1 Aufruf

n = 2:



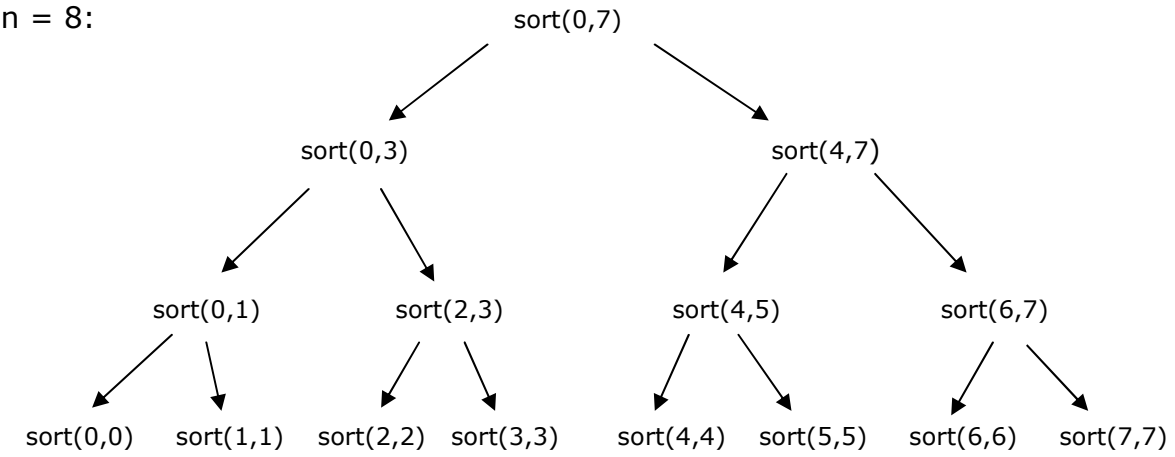
$1 + 2 \cdot 1 = 3$ Aufrufe

n = 4:



$1 + 2 \cdot 3 = 7$ Aufrufe

n = 8:



$1 + 2 \cdot 7 = 15$ Aufrufe

$f(1) = 1 = 1 = 2 \cdot 1 - 1$

$f(2) = 1 + 2 \cdot 1 = 3 = 2 \cdot 2 - 1$

$f(4) = 1 + 2 \cdot 3 = 7 = 2 \cdot 4 - 1$

$f(8) = 1 + 2 \cdot 7 = 15 = 2 \cdot 8 - 1$

$f(16) = 1 + 2 \cdot 15 = 31 = 2 \cdot 16 - 1$

$f(32) = 1 + 2 \cdot 31 = 63 = 2 \cdot 32 - 1$

allgemein:

$f(n) = 2 \cdot n - 1$

Offensichtlich ist f(n) Lösung der rekursiv definierten Funktionalgleichung

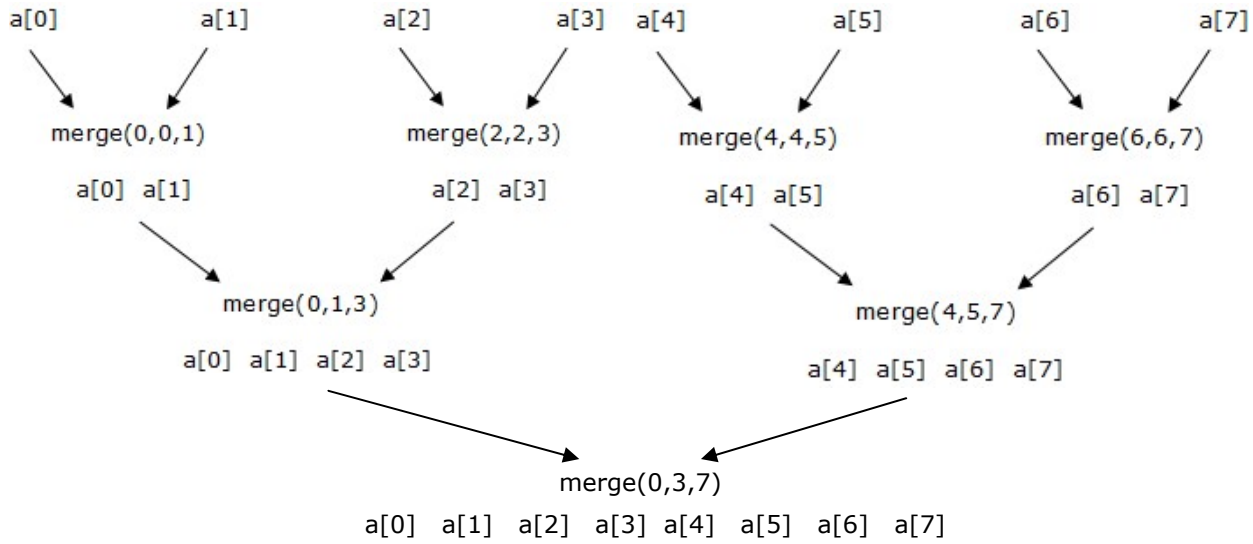
$f(n) = 1 + 2 \cdot f(n/2)$

mit der Anfangsbedingung **$f(1) = 1$** .

Die Anzahl **f(n)** der gleichzeitig aktiven Aufrufe von MergeSort und damit der Speicherplatzbedarf während der Laufzeit wächst somit linear mit **n**, also wesentlich schwächer als die Anzahl **A(n)** elementarer Rechenoperationen.

Die rekursiv veranlaßten Aufrufe der Funktion **`sort`** zerlegen die zu sortierende Liste in Teillisten jeweils der Länge 1, die als ein-elementige Listen bereits sortiert sind. Die Funktion **`merge`** mischt je zwei sortierte Teillisten zu jeweils einer sortierten Liste gemäß folgendem Diagramm:

Bemerkung: Der Pfeil ⟶ *bedeutet: „wird gemischt"*



Für die Anzahl g(n) der Aufrufe von merge verifiziert man unmittelbar:

g(1) = 0
g(n) = 1 + 2 · g(n/2) falls $n = 2^k$, $k > 1$

Lösung der vorstehenden Funktionalgleichung:

g(n) = n – 1

Die Anzahl f(n) der Aufrufe der Funktion **`sort`** und die Anzahl g(n) der Aufrufe der Funktion **`merge`** wachsen jeweils linear mit n.

Februar 2021

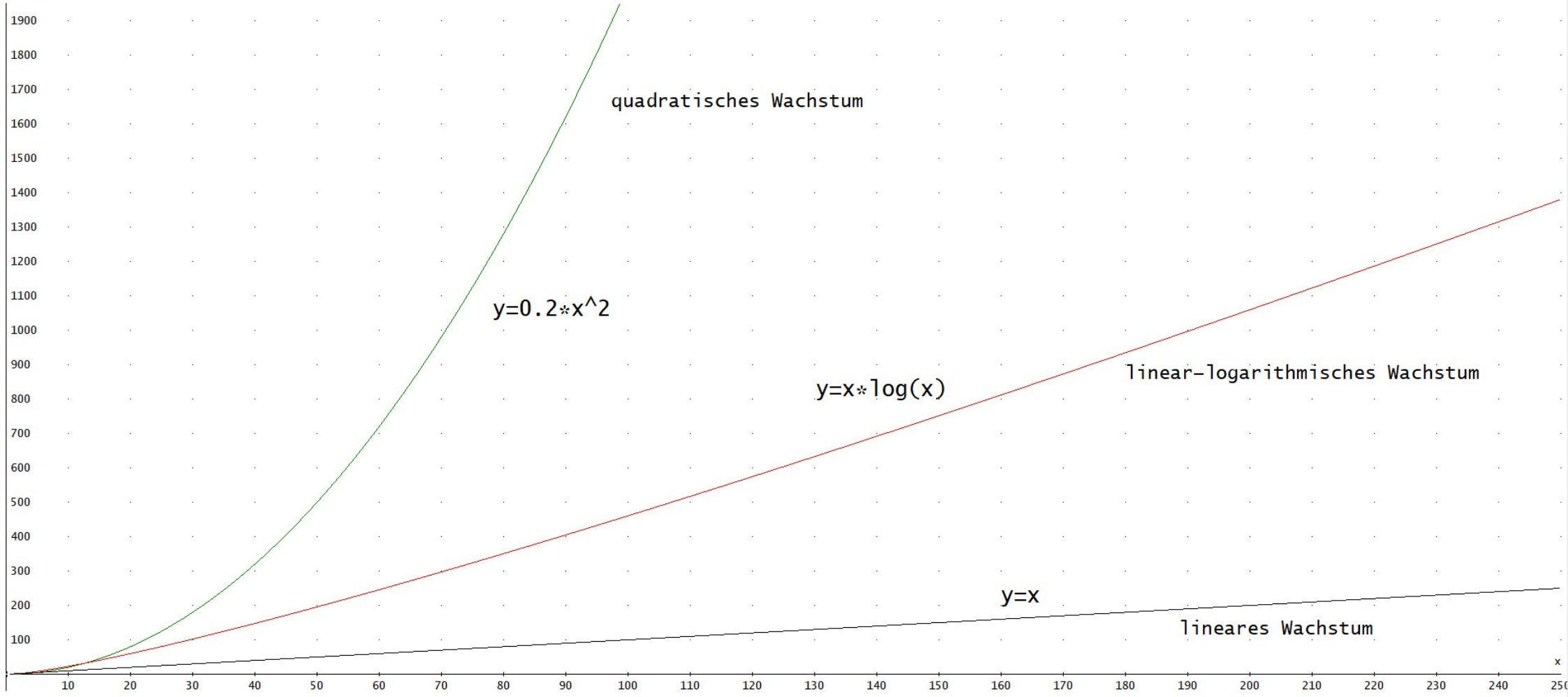
Bemerkung:
Für den Aufwand A(n) und folglich den Zeitbedarf zur Laufzeit des Algorithmus erhalten wir bei

- SelectionSort: $A(n) \sim n^2$
- MergeSort: $A(n) \sim n \cdot \log_2(n)$
- Fibonacchi-Folge: $A(n) \sim 2^n$ (bei rekursiver Berechnung)
- BinarySearch: $A(n) \sim \log_2(n)$

Entsprechend haben

- SelectionSort quadratische Komplexität,
- MergeSort linear-logarithmische Komplexität,
- die rekursive Berechnung der Fibonacchi-Folge exponentielle Komplexität,
- BinarySearch logarithmische Komplexität.

Algorithmen mit exponentieller Komplexität erweisen sich in der Praxis als unbrauchbar.


quadratisches Wachstum
y=0.2*x^2
y=x*log(x)
linear-logarithmisches Wachstum
y=x
lineares Wachstum
x
1900
1800
1700
1600
1500
1400
1300
1200
1100
1000
900
800
700
600
500
400
300
200
100
10
20
30
40
50
60
70
80
90
100
110
120
130
140
150
160
170
180
190
200
210
220
230
240

Arbeitsauftrag für die Doppelstunde am 09.09.2021 inf12

Die aus den **n** Komponenten **a[0], a[1], ... , a[n-1]** bestehende Liste **a** soll aufsteigend sortiert werden.

1. Gegeben ist der Quelltext **SelectionSort.txt** zum Algorithmus „Sortieren durch direkte Auswahl"; nach Eingabe einer natürlichen Zahl **n** wird eine aus **n** Zufallszahlen bestehende Liste **a** erzeugt und anschließend aufsteigend sortiert.

   Wir modifizieren diesen Quelltext so, daß das Sortieren nach dem Algorithmus „MergeSort" erfolgt; ersetze hierzu denjenigen Programmteil, der den Sortiervorgang veranlaßt, in geeigneter Weise durch die Funktionen **sort** und **merge**. Benutze hierzu das Skriptum **MergeSort_01-09-2021.pdf** und den Quelltext **function_merge.txt** der Funktion **merge**.
   Aufruf der Funktion **sort** zum Sortieren der Liste **a**: **sort(a, 0, len(a)-1)**

2. Vergleiche die Algorithmen **SelectionSort** und **MergeSort** experimentell hinsichtlich ihrer zeitlichen Effizienz.

3. Implementiere Variable **x** und **y**, um die Anzahl der Aufrufe der Funktionen **sort** und **merge** jeweils zu zählen, und bestätige die diesbezüglichen Ergebnisse aus dem Skriptum.

4. **Hausaufgabe:**
   Um den Aufwand bei **SelectionSort** zu ermitteln, betrachten wir denjenigen Programmteil, der das Sortieren ausführt:

```
j = 0
while j <= n-2:
    i = j + 1
    min = a[j]
    while i < n:
        if a[i] < min:
           min = a[i]
           a[i] = a[j]
           a[j] = min
        i = i + 1
    j = j + 1
```

   Wir fassen die Anweisungen aus dem Schleifenrumpf der inneren Schleife dieses Programmauszugs gedanklich zum Anweisungsblock **A** zusammen
   (markiere Block **A** im obenstehenden Programmtext).
   Um den Aufwand zu ermitteln, ein aus **n** Komponenten bestehendes array zu sortieren, fragen wir, wie oft Block **A** in Abhängigkeit von **n** abgearbeitet wird.

   a) Vervollständige die Einträge in folgender Tabelle, wobei z(j) angibt, wie oft Block **A** in Abhängigkeit von j abgearbeitet wird.

Index j	**Index i**	**z(j)**
j = 0	≤ i ≤	
j = 1	≤ i ≤	
j = 2	≤ i ≤	
....	....	....
j = n-3	≤ i ≤	
j = n-2	≤ i ≤	

   b) Die Gesamtzahl **z** der Abarbeitungen von Block **A** ergibt sich als

   z = z(0) + z(1) + z(2) + z(3) + ... + z(n-3) + z(n-2)

   Vereinfache diese Summe und zeige so, daß **z** quadratisch mit **n** wächst!

   Hinweis: Für die Summe der ersten n natürlichen Zahlen gilt bekanntlich:
   $1 + 2 + \ldots\ldots\ldots + n = \frac{1}{2} \cdot n \cdot (n + 1)$

Komplexität zur Laufzeit (Sorting and Searching)

SelectionSort - BinarySearch

Array mit 10.000 Komponenten:

```
Laenge des arrays: 10000

Wieviele Elemente sollen angezeigt werden? 3

5016012
6439926
6543928

Sortierte Liste:

428
1045
1894

SelectionSort
Zeitaufwand zum Sortieren von 10000 Elementen:  17.687 s
Anzahl Durchlaeufe der inneren Schleife:   49995000

gesuchtes Element: 6543339

6543339 wurde nicht gefunden

Zeitaufwand BinarySearch bei 10000 Elementen:   0.000 s
Anzahl Aufrufe binarysearch:  14
```

Array mit 20.000 Komponenten:

```
Laenge des arrays: 20000

Wieviele Elemente sollen angezeigt werden? 3

6910309
3265111
5248564

Sortierte Liste:

312
549
555

SelectionSort
Zeitaufwand zum Sortieren von 20000 Elementen:  65.406 s
Anzahl Durchlaeufe der inneren Schleife:   199990000

gesuchtes Element: 4268819

4268819 wurde nicht gefunden

Zeitaufwand BinarySearch bei 20000 Elementen:   0.000 s
Anzahl Aufrufe binarysearch:  14
```

MergeSort - BinarySearch

Array mit 20.000 Komponenten:

```
Laenge des arrays: 20000

Wieviele Elemente sollen angezeigt werden? 3

7563549
5638990
5059315

Sortierte Liste:

484
896
1319

MergeSort
Zeitaufwand zum Sortieren von 20000 Elementen:   0.118 s
Anzahl Aufrufe sort:   39999
Anzahl Aufrufe merge:  19999

gesuchtes Element: 7374881

7374881 wurde nicht gefunden

Zeitaufwand BinarySearch bei 20000 Elementen:   0.000 s
Anzahl Aufrufe binarysearch:  14
```

Array mit 1.000.000 Komponenten:

```
Laenge des arrays: 1000000

Wieviele Elemente sollen angezeigt werden? 3

4393058
1949575
2241821

Sortierte Liste:

1
7
12

MergeSort
Zeitaufwand zum Sortieren von 1000000 Elementen:   8.841 s
Anzahl Aufrufe sort:   1999999
Anzahl Aufrufe merge:  999999

gesuchtes Element: 6338229

6338229 wurde nicht gefunden

Zeitaufwand BinarySearch bei 1000000 Elementen:   0.037 s
Anzahl Aufrufe binarysearch:  20
```

Array mit 4.000.000 Komponenten:

```
Laenge des arrays: 4000000

Wieviele Elemente sollen angezeigt werden? 8

9320507
2318498
2925900
542557
7231914
3414556
8504507
8492286

Sortierte Liste:

4
10
12
16
22
24
24
30

MergeSort
Zeitaufwand zum Sortieren von 4000000 Elementen:  39.388 s
Anzahl Aufrufe sort:   7999999
Anzahl Aufrufe merge:  3999999

gesuchtes Element: 3456789

3456789 wurde nicht gefunden

Zeitaufwand BinarySearch bei 4000000 Elementen:   0.197 s
Anzahl Aufrufe binarysearch:  22
```

Informatik 12
September 2021

Boolesche Terme und Schaltalgebra

1. Datentyp boolean

Eine Boolesche Variable oder ein Boolescher Ausdruck (Term) nimmt nur zwei Werte an: **`True`** oder **`False`**
(abkürzend: 1 oder 0; in Python sind **`True`** oder **`False`** zu verwenden)

Insbesondere sind folgende Terme Boolesche Ausdrücke, deren Wert sich auch einer Variablen zuweisen läßt:

`8 > 5` hat den Wert **`True`**
`7 == 8` hat den Wert **`False`**
`7 != 8` hat den Wert **`True`**
`x` hat den Wert **`True`** nach der Wertzuweisung **`x = 7 < 12`**
`x` hat den Wert **`False`** nach der Wertzuweisung **`x = (0 == 6)`**
`a or b` hat den Wert **`True`** genau dann, wenn mindestens eine der Variablen **`a`**, **`b`** den Wert **`True`** hat; andernfalls hat **`a or b`** den Wert **`False`**.

Mit **`a = 7 != 8`** oder **`a = (7 != 8)`** wird in Python der Booleschen Variablen **`a`** der Wert des Booleschen Terms **`7 != 8`** (hier: **`True`**) zugewiesen.

Wir definieren die Verknüpfungen **`and`** und **`or`** sowie die Operation **`not`** jeweils über eine Wahrheitstafel:

a	b	a or b
False	False	False
False	True	True
True	False	True
True	True	True

a	b	a and b
False	False	False
False	True	False
True	False	False
True	True	True

a	not a
False	True
True	False

Abkürzende Schreibweisen (a, b, c sind Boolesche Variable oder Boolesche Terme):

`a and b` = $a \wedge b$ = $a \cdot b$ = **`a b`**
`a or b` = $a \vee b$ = **`a + b`**
`not a` = $\neg a$ = $\overline{a}$

Dabei gelte auch die aus der Algebra bekannte Vereinbarung "Punkt vor Strich", d. h.

`a + (b · c) = a + b · c = a + b c`

Die **AND**-Verknüpfung nennen wir auch **Konjunktion**,
die **OR**-Verknüpfung **Disjunktion**.

2. Rechenregeln für Boolesche Variable

Kommutativgesetz

(1) $a + b = b + a$ (1′) $a \cdot b = b \cdot a$

Assoziativgesetz

(2) $a + (b + c) = (a + b) + c$ (2′) $a \cdot (b \cdot c) = (a \cdot b) \cdot c$

Distributivgesetz

(3) $a \cdot (b + c) = a \cdot b + a \cdot c$ (3′) $a + b \cdot c = (a + b) \cdot (a + c)$

Absorptionsgesetz

(4) $a(a + b) = a$ **(4′)** $a + ab = a$

Tautologie

(5) $a \cdot a = a$ **(5′)** $a + a = a$

Gesetz über die Negation

(6) $\bar{a} \cdot a = 0$ **(6′)** $\bar{a} + a = 1$

Doppelte Negation

(7) $\bar{\bar{a}} = a$

Gesetz von De Morgan

(8) $\overline{a \cdot b} = \bar{a} + \bar{b}$ **(8′)** $\overline{a+b} = \bar{a} \cdot \bar{b}$

Operationen mit 0 und 1

(9.1) $a \cdot 1 = a$ **(9.1′)** $a + 0 = a$

(9.2) $a \cdot 0 = 0$ **(9.2′)** $a + 1 = 1$

(9.3) $\text{not } 0 = 1$ **(9.3′)** $\text{not } 1 = 0$

Bemerkung: Die jeweils in einer Zeile stehenden Gesetze sind duale Gesetze voneinander; Beispiel: (3′) ist das duale Gesetz von (3), (3) das duale Gesetz von (3′).

Beweis von Rechengesetz (3):

a	b	c	b + c	a(b + c)	ab	ac	ab + ac
0	0	0	0	0	0	0	0
0	0	1	1	0	0	0	0
0	1	0	1	0	0	0	0
0	1	1	1	0	0	0	0
1	0	0	0	0	0	0	0
1	0	1	1	1	0	1	1
1	1	0	1	1	1	0	1
1	1	1	1	1	1	1	1

Da die Spalten zu **a(b + c)** und **ab + ac** übereinstimmen, gilt: **a(b + c) = ab + ac** .

Aufgaben:

1. Beweise das Distributivgesetz (3′).
2. Beweise die Gesetze von De Morgan.
   Hinweis: Wahrheitstafel; außer den Spalten für a und b (4 Zeilen) erstelle Spalten für $a \cdot b$, $\overline{a \cdot b}$, $\bar{a}$, $\bar{b}$, $\bar{a} + \bar{b}$ für Regel (8).
3. Unter der Disjunktion **a or b** versteht man das nichtausschließende **oder** („non-exclusive or"), d. h., **a or b** ist genau dann **True**, falls **a** oder **b** oder sowohl **a** als auch **b** **True** sind („oder" im Sinne von lat. vel).
   Unter der Verknüpfung **a xor b** (andere Schreibweise: **a** $\oplus$ **b**) versteht man das ausschließende **oder** (**ex**clusive **or**), d. h., **a** $\oplus$ **b** ist genau dann **True**, falls entweder **a** oder **b** den Wert **True** hat.
   Zeige: **a** $\oplus$ **b** = $\bar{a} \cdot b + a \cdot \bar{b}$

BEISPIEL 1

Die Boolesche Funktion
z = f(a,b,c)
ist durch nebenstehende Wahrheitstafel gegeben:

a	b	c	z
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

a) Ermittle die disjunktive Normalform (**DNF**; Disjunktion von Konjunktionen) für **z**.

b) Vereinfache den Funktionsterm unter Anwendung der Booleschen Rechengesetze.

c) Zeichne den Schaltplan für die optimierte Funktion **z**.

Lösung:

a) $z = \bar{a} \cdot \bar{b} \cdot \bar{c} + \bar{a} \cdot \bar{b} \cdot c + \bar{a} \cdot b \cdot c + a \cdot b \cdot c$

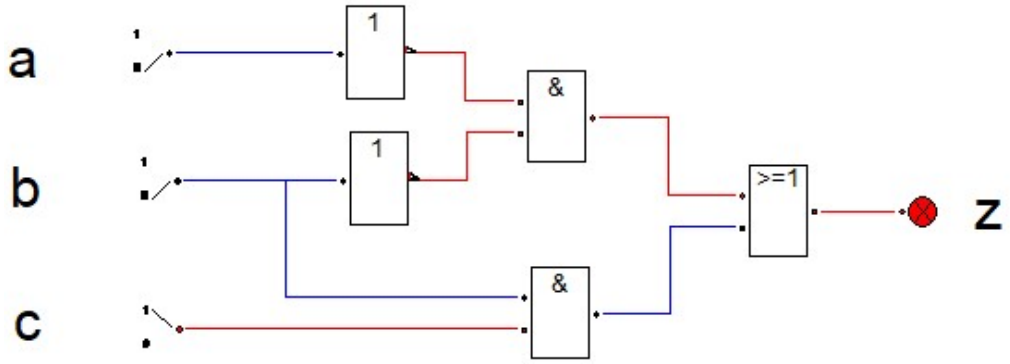
b) $z = \bar{a} \cdot \bar{b} \cdot (\bar{c} + c) + b \cdot c \cdot (\bar{a} + a)$ *Kommutativ- und Distributivgesetz*

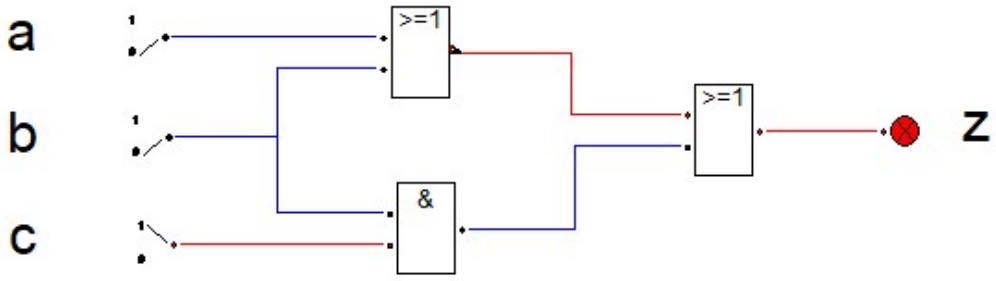
$= \bar{a} \cdot \bar{b} \cdot 1 + b \cdot c \cdot 1$

$= \bar{a} \cdot \bar{b} + b \cdot c$

$= \overline{a+b} + b \cdot c$ *de Morgan's Gesetz*

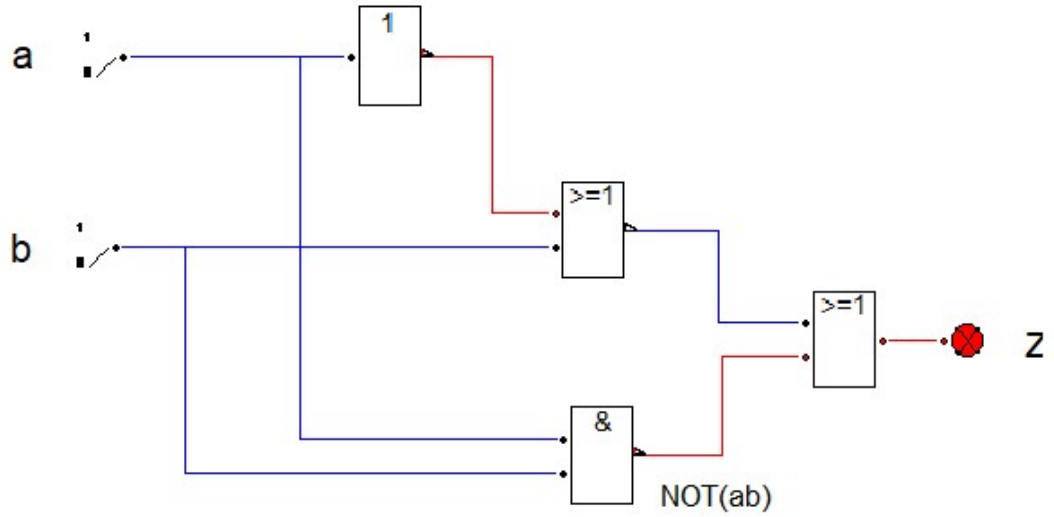
c) $z = \bar{a} \cdot \bar{b} + b \cdot c$ (oben) $\qquad z = \overline{a+b} + b \cdot c$ (unten)





BEISPIEL 2

Gegeben ist folgende digitale Schaltung mit den Eingangsvariablen **a**, **b** und der Ausgangsvariablen **z**:



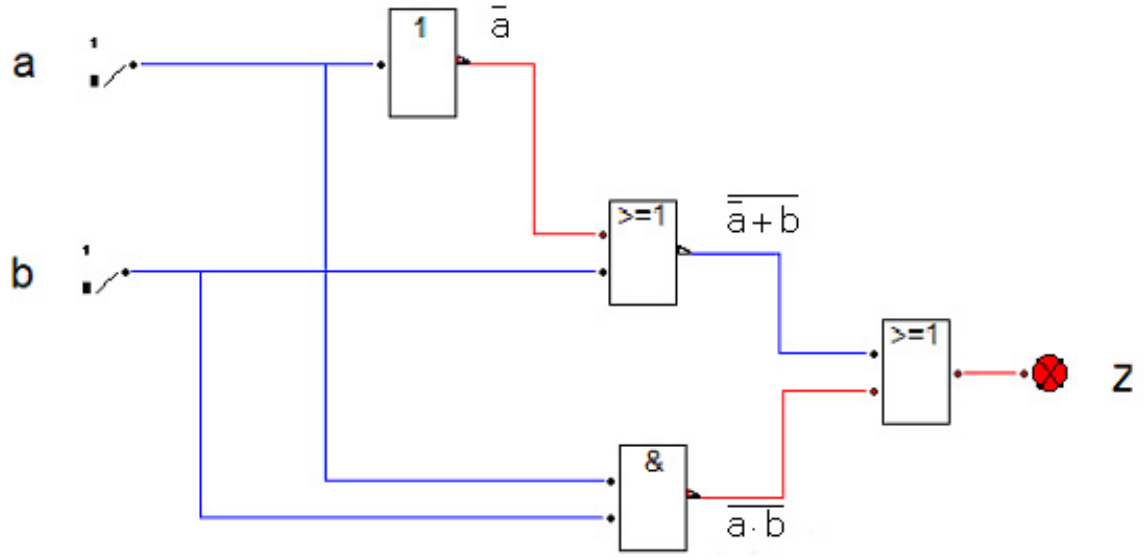
a) Ermittle den Booleschen Term für die Boolesche Funktion **z = f(a,b,c)**. Hinweis: Notiere am Ausgang jedes Gatters jeweils den Booleschen Term (Beispiel: $\overline{a \cdot b}$ am Ausgang des NAND-Gatters).

b) Vereinfache den in a) erhaltenen Term unter Verwendung der Rechenregeln für Boolesche Ausdrücke;

c) Erstelle die Wahrheitstafel und zeichne das Schaltbild für den vereinfachten Funktionsterm; teste beide Schaltungsvarianten mit einem Digitalsimulationsprogramm.

Lösung:

zu a):

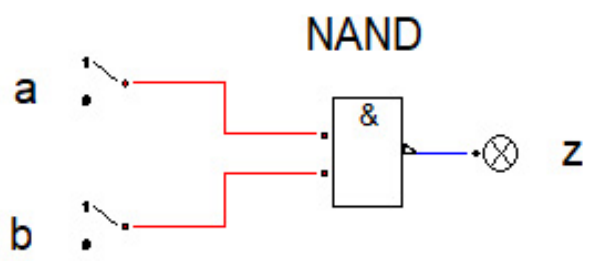


zu b):

$$
\begin{aligned}
z &= \overline{\bar{a} + b} + \overline{a \cdot b} \\
&= \bar{\bar{a}} \cdot \bar{b} + (\bar{a} + \bar{b}) && \textit{(2-mal de Morgan)} \\
&= a \cdot \bar{b} + \bar{a} + \bar{b} && \textit{(wegen } \bar{\bar{a}} = a\textit{)} \\
&= \bar{a} + \bar{b} \cdot a + \bar{b} && \textit{(Kommutativgesetze)} \\
&= \bar{a} + \bar{b} \cdot a + \bar{b} \cdot 1 && \textit{(wegen } a = a \cdot 1\textit{)} \\
&= \bar{a} + \bar{b} \cdot (a + 1) && \textit{(Distributivgesetz)} \\
&= \bar{a} + \bar{b} && \textit{(wegen } a + 1 = 1\textit{)} \\
&= \overline{a \cdot b} && \textit{(de Morgan)}
\end{aligned}
$$

zu c):

optimierte Schaltung:

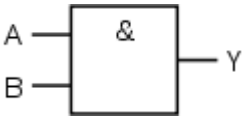
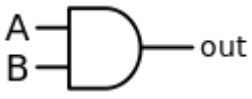
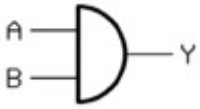
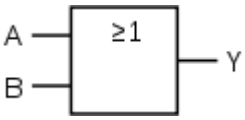
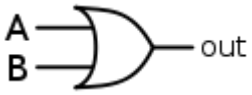
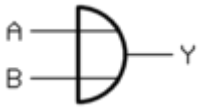
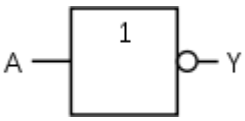
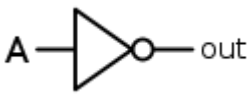
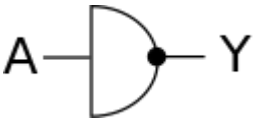
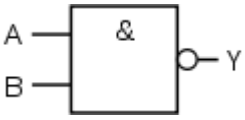
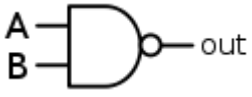
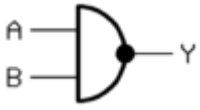
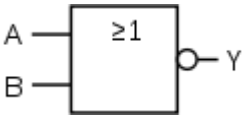
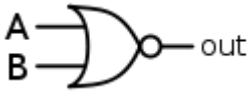
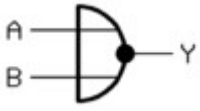
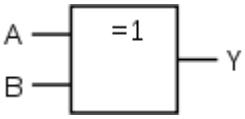
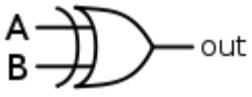
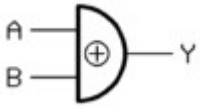


Wertetabelle:

a	b	z
0	0	1
0	1	1
1	0	1
1	1	0

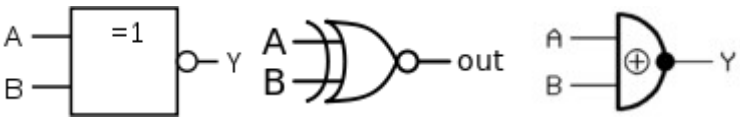
Typen von Logikgattern und Symbolik

Logikgatter werden mit Schaltsymbolen bezeichnet, die nach unterschiedlichen, mehr oder weniger parallel existierenden Standards definiert sind.

Name	Funktion	Symbol in Schaltplan			Wahrheits-tabelle
		IEC 60617-12 : 1997 & ANSI/IEEE Std 91/91a-1991	**ANSI/IEEE Std 91/91a-1991**	**DIN 40700 (vor 1976)**	
Und-Gatter (AND)	$Y=A \cdot B$				**A B Y** 0 0 0 0 1 0 1 0 0 1 1 1
Oder-Gatter (OR)	$Y=A+B$				**A B Y** 0 0 0 0 1 1 1 0 1 1 1 1
Nicht-Gatter (NOT)	$Y=\overline{A}$				**A Y** 0 1 1 0
NAND-Gatter (NICHT UND) (NOT AND)	$Y=\overline{A \cdot B}$				**A B Y** 0 0 1 0 1 1 1 0 1 1 1 0
NOR-Gatter (NICHT ODER) (NOT OR)	$Y=\overline{A+B}$				**A B Y** 0 0 1 0 1 0 1 0 0 1 1 0
XOR-Gatter (Exklusiv-ODER, Antivalenz) (eXclusiveOR)	$Y=A \oplus B$				**A B Y** 0 0 0 0 1 1 1 0 1 1 1 0

XNOR-Gatter
(Exklusiv-Nicht-ODER, Äquivalenz)
(eXclusive Not OR)

$Y = \overline{A \oplus B}$



A	B	Y
0	0	1
0	1	0
1	0	0
1	1	1

Früher waren auf dem europäischen Kontinent die deutschen Symbole (rechte Spalte) verbreitet; im englischen Sprachraum waren und sind die amerikanischen Symbole (mittlere Spalte) üblich. Die IEC-Symbole sind international auf beschränkte Akzeptanz gestoßen und werden in der amerikanischen Literatur (fast) durchgängig ignoriert.

JK-Flipflop

Ein Flip-Flop (bistabile Kippstufe oder bistabiler Multivibrator) hat zwei stabile Zustände am Ausgang Q; die Zustände heißen „gesetzt“ (set) oder „zurückgesetzt“ (reset). Ein 1-Bit-Speicher läßt sich somit als FlipFlop realisieren.

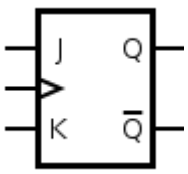
Ein JK-FlipFlop ist ein taktgesteuertes FlipFlop: die an den Eingängen J und K liegende Information wird mit einer Flanke (hier: der steigenden Flanke) des an C liegenden Taktsignals auf die Ausgänge Q und $\overline{Q}$ übernommen.

Mit dem Taktsignal (clock, C) und der Eingangsbelegung J = 1 und K = 0 wird am Ausgang Q eine 1 erzeugt und gespeichert, alternativ eine 0 bei J = 0 und K = 1.

Bei der Realisierung des JK-Flipflops als taktflankengesteuertes Flipflop kann der Eingang C für steigende Flanken (Wechsel von 0 auf 1) oder für fallende Flanken (Wechsel von 1 auf 0) ausgelegt sein.

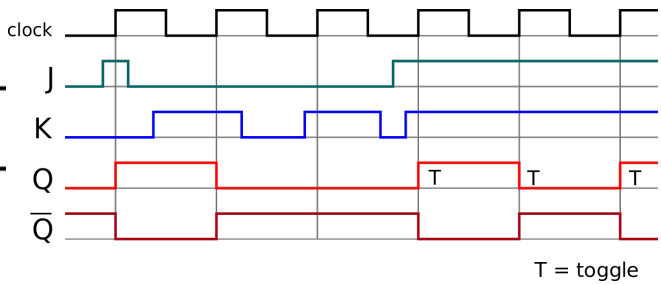
Name und Schaltzeichen

Flanken-gesteuertes JK-Flipflop



Signal-Zeit-Diagramm

Übernahme der Eingangsinformation durch steigende Flanke an C (clock)



Funktionstabelle

bis zur … n-ten Taktflanke		nach der … n-ten Taktflanke
J	**K**	**Q_n**
0	0	Q_{n-1} (unverändert)
0	1	0 (zurückgesetzt)
1	0	1 (gesetzt)
1	1	NOT Q_{n-1} (gewechselt)

(Wikipedia)

Halbaddierer und Volladdierer

Die Ziffern einer im Dezimalsystem geschriebenen Zahl a ergeben sich als Aneinanderreihung der Koeffizienten aus der Dezimalzerlegung (Summe von Zehnerpotenzen) von a; entsprechend erhalten wir die Darstellung von a im Dualsystem als Aneinanderreihung der Koeffizienten aus der Dualzerlegung (Summe von Zweierpotenzen).

$87_{dezimal} = 8 \cdot 10^1 + 7 \cdot 10^0$

$87_{dezimal} = 1 \cdot 2^6 + 0 \cdot 2^5 + 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 1010111_{dual}$

Addition der Dualzahlen
$a = a_3 \cdot 2^3 + a_2 \cdot 2^2 + a_1 \cdot 2^1 + a_0 \cdot 2^0$ und $b = b_3 \cdot 2^3 + b_2 \cdot 2^2 + b_1 \cdot 2^1 + b_0 \cdot 2^0$:

		a_3	a_2	a_1	a_0
+		b_3	b_2	b_1	b_0
	s_4	s_3	s_2	s_1	s_0

		1	1	0	1
+		1	0	1	1
	1	1	0	0	0

Den Übertrag („carry"), der sich aus der i-ten Stelle ergibt und der bei der Addition in der (i + 1)-ten Stelle zu berücksichtigen ist, bezeichnen wir mit c_{i+1}; $i \geq 0$.

Für die 0-te Stelle genügt ein Halbaddierer mit den Eingängen a_0 und b_0 und den Ergebnissen s_0 und c_1; die Addition in der i-ten Stelle, $i \geq 1$, erfordert einen Volladdierer mit den Eingängen a_i, b_i, c_i und den Ergebnissen s_i und c_{i+1}.

Halbaddierer HA

Wahrheitstafel:

a_0	b_0	s_0	c_1
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Wir ermitteln für $\mathbf{s_0}$ und $\mathbf{c_1}$ jeweils die disjunktive Normalform („Disjunktion der Konjunktionen"):

$$s_0 = \overline{a_0} \cdot b_0 + a_0 \cdot \overline{b_0} = a_0 \oplus b_0$$

$$c_1 = a_0 \cdot b_0$$

Volladdierer VA

Wahrheitstafel:

a_i	b_i	c_i	s_i	c_{i+1}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Wir ermitteln für $\mathbf{s_i}$ und $\mathbf{c_{i+1}}$ jeweils die disjunktive Normalform („Disjunktion der Konjunktionen") und vereinfachen ggf. die booleschen Funktionsterme:

$$s_i = \overline{a_i} \cdot \overline{b_i} \cdot c_i + \overline{a_i} \cdot b_i \cdot \overline{c_i} + a_i \cdot \overline{b_i} \cdot \overline{c_i} + a_i \cdot b_i \cdot c_i$$

ohne Index i geschrieben:

$$s = \overline{a} \cdot \overline{b} \cdot c + \overline{a} \cdot b \cdot \overline{c} + a \cdot \overline{b} \cdot \overline{c} + a \cdot b \cdot c$$

$$s = (\overline{a} \cdot b + a \cdot \overline{b}) \cdot \overline{c} + \overline{a} \cdot \overline{b} \cdot c + a \cdot b \cdot c$$

$$s = (a \oplus b) \cdot \overline{c} + (\overline{a} \cdot \overline{b} + a \cdot b) \cdot c$$

$$s = (a \oplus b) \cdot \overline{c} + (0 + \overline{a} \cdot \overline{b} + a \cdot b + 0) \cdot c$$

$$s = (a \oplus b) \cdot \overline{c} + (\overline{a} \cdot a + \overline{a} \cdot \overline{b} + a \cdot b + b \cdot \overline{b}) \cdot c$$

$$s = (a \oplus b) \cdot \overline{c} + [(\overline{a} + b) \cdot (a + \overline{b})] \cdot c$$

$$s = (a \oplus b) \cdot \overline{c} + [(\overline{a} + \overline{\overline{b}}) \cdot (\overline{\overline{a}} + \overline{b})] \cdot c$$

$$s = (a \oplus b) \cdot \overline{c} + [\overline{(a \cdot \overline{b})} \cdot \overline{(\overline{a} \cdot b)}] \cdot c$$

$$s = (a \oplus b) \cdot \overline{c} + \overline{[a \cdot \overline{b} + \overline{a} \cdot b]} \cdot c$$

$$s = (a \oplus b) \cdot \overline{c} + \overline{(a \oplus b)} \cdot c$$

$$s = (a \oplus b) \oplus c$$

mit Index i erhält man:

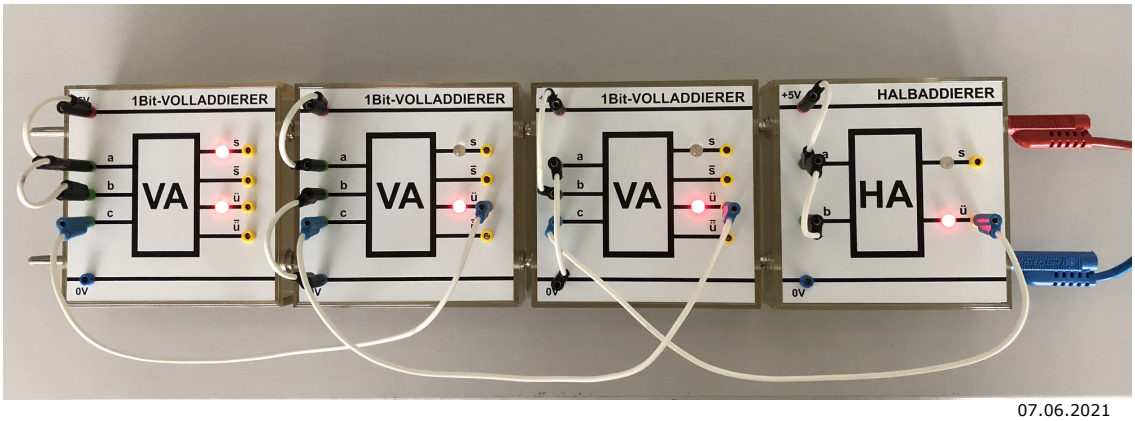
$$s_i = (a_i \oplus b_i) \oplus c_i$$

$$c_{i+1} = \overline{a_i} \cdot b_i \cdot c_i + a_i \cdot \overline{b_i} \cdot c_i + a_i \cdot b_i \cdot \overline{c_i} + a_i \cdot b_i \cdot c_i$$

$$c_{i+1} = (\overline{a_i} \cdot b_i + a_i \cdot \overline{b_i}) \cdot c_i + a_i \cdot b_i \cdot (\overline{c_i} + c_i)$$

$$c_{i+1} = (\overline{a_i} \cdot b_i + a_i \cdot \overline{b_i}) \cdot c_i + a_i \cdot b_i \cdot 1$$

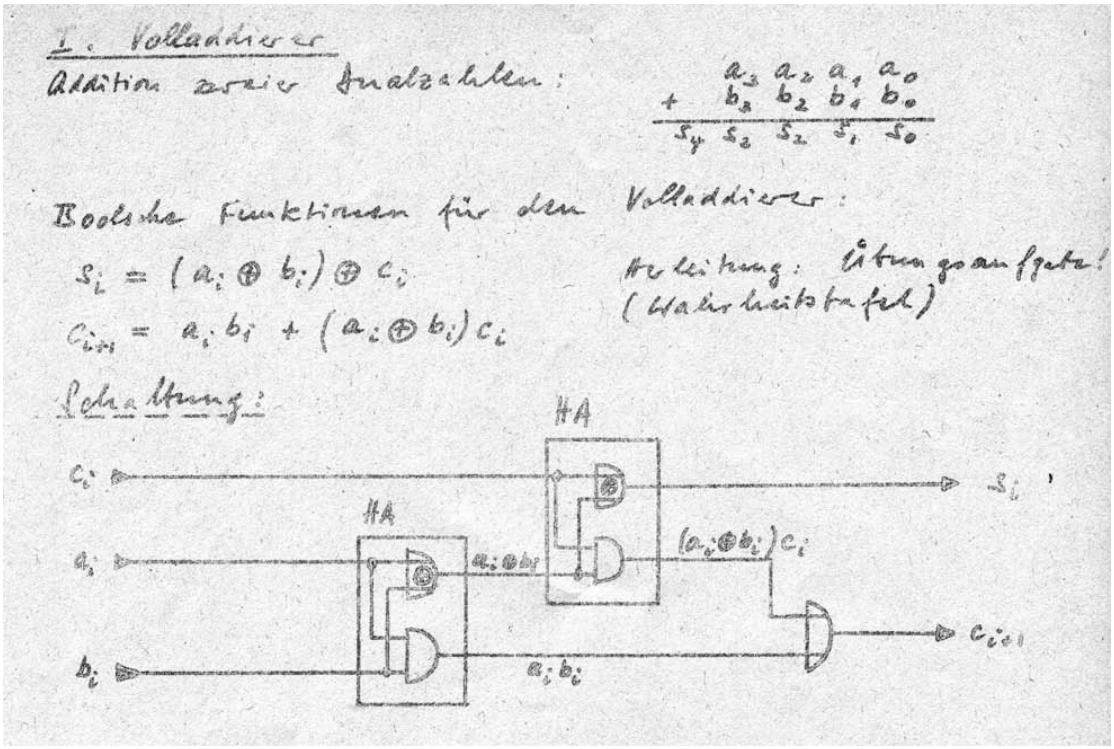
$$c_{i+1} = (a_i \oplus b_i) \cdot c_i + a_i \cdot b_i$$

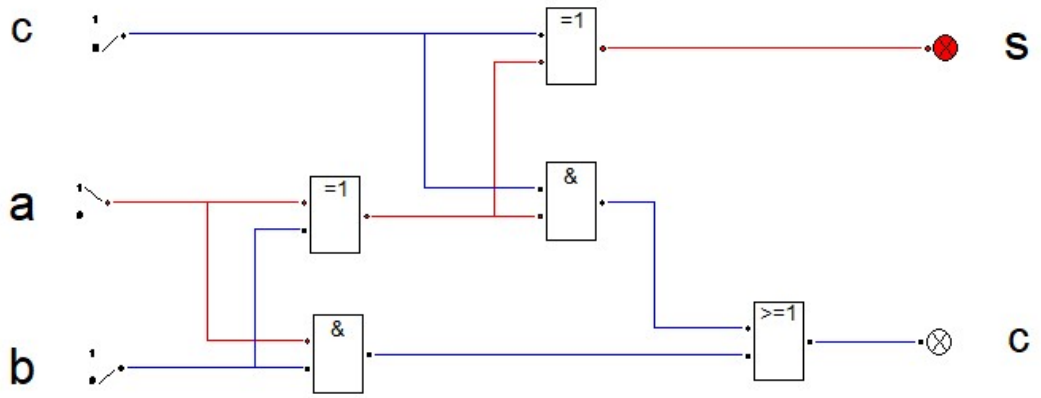


07.06.2021

Halbaddierer (HA) und Volladdierer (VA)

Schaltungen



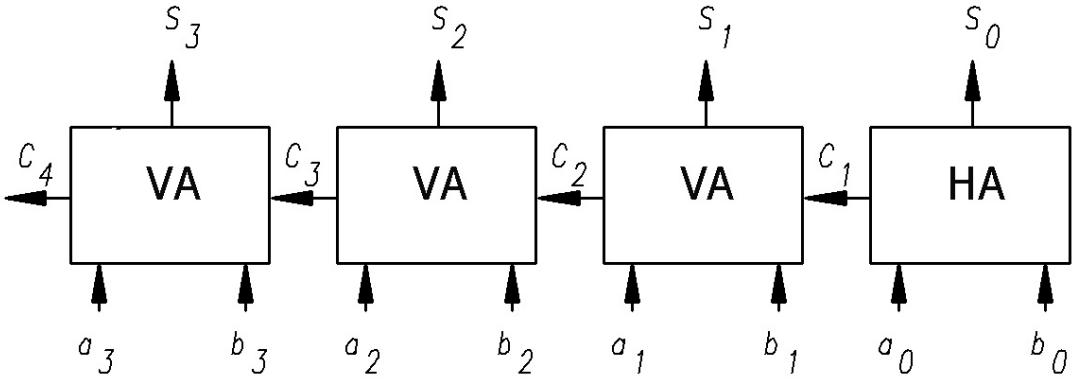


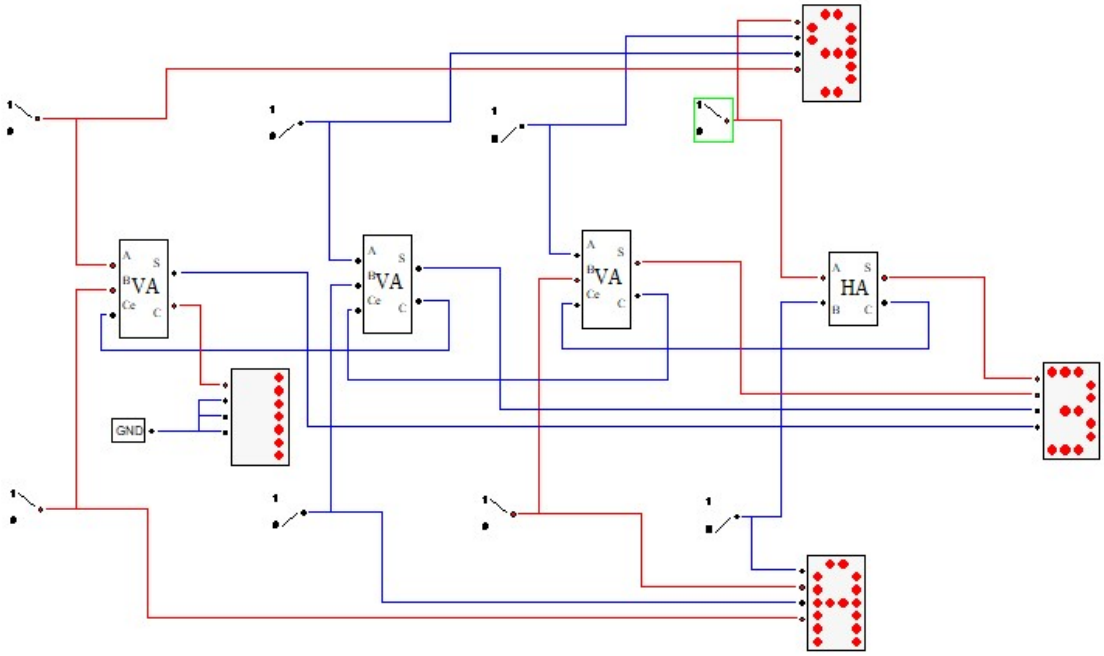
Addier-Schaltungen für Dualzahlen

		a_3	a_2	a_1	a_0
+		b_3	b_2	b_1	b_0
	s_4	s_3	s_2	s_1	s_0

1. Paralleladdierer mit seriellem Übertrag (hier: 4-Bit-Addierer)

Für das Least Significant Bit (LSB) genügt ein Halbaddierer (HA); die höherwertigen Bits erfordern jeweils einen Volladdierer, da hier der Übertrag aus der vorherigen Stelle zu berücksichtigen ist.



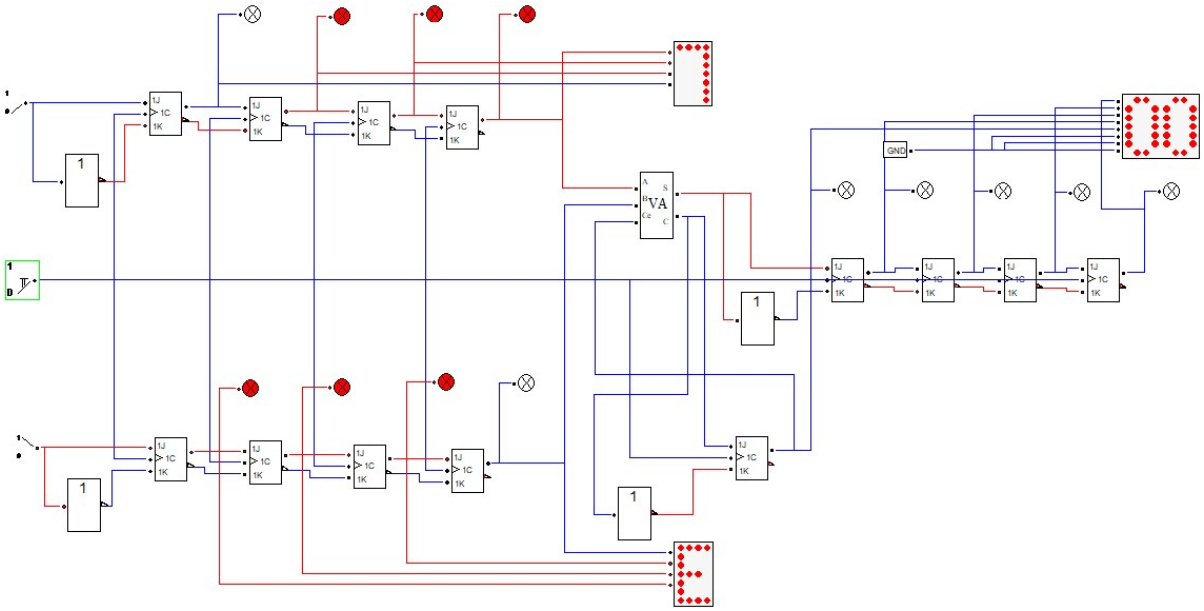



```
Dezimal:  09   Hexadezimal:  09    Dual: 0000 1001
        + 10                 + 0A        + 0000 1010
          19                   13          0001 0011
```

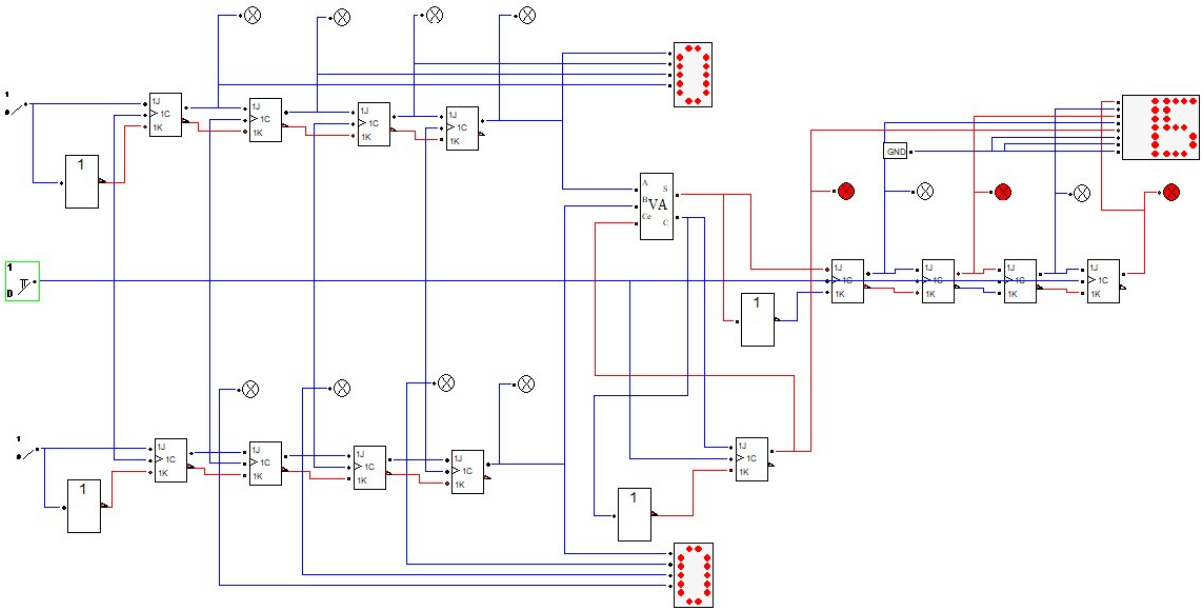
2. Serieller 1-Bit-Addierer für 4-stellige Dualzahlen

Die Operanden werden jeweils in einem 4-Bit-Schieberegister abgelegt, nach 4 Taktimpulsen finden wir das Ergebnis (hier: die Summe) in einem weiteren 4-Bit-Schieberegister.

Da der Übertrag aus der vorherigen Stelle für die Addition in der aktuellen Stelle zu berücksichtigen ist, wird er in einem Flip-Flop zwischengespeichert. Dieses Flip-Flop liefert auch das Most Significant Bit (MSB) des Ergebnisses.

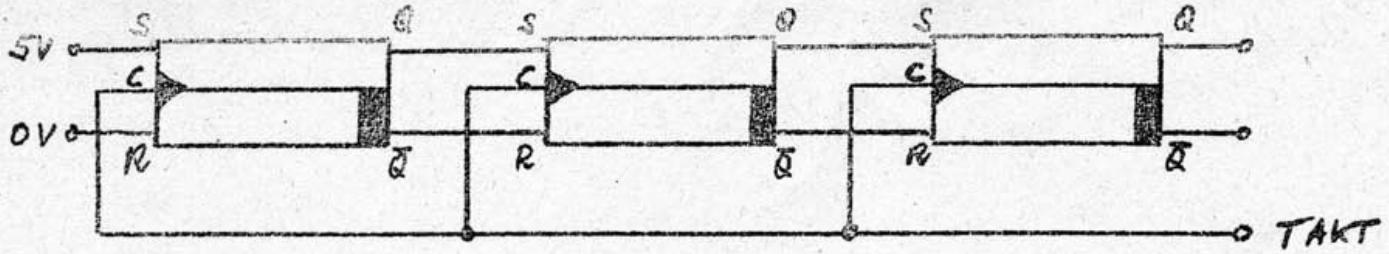


Nach 4 Taktimpulsen (hier: Triggerung der Flip-Flops auf der steigenden Taktflanke):



Dezimal:	Hexadezimal:	Dual:
07	07	0000 0111
+ 14	+ 0E	+ 0000 1110
21	15	0001 0101

SCHIEBE REGISTER



Beispiel:

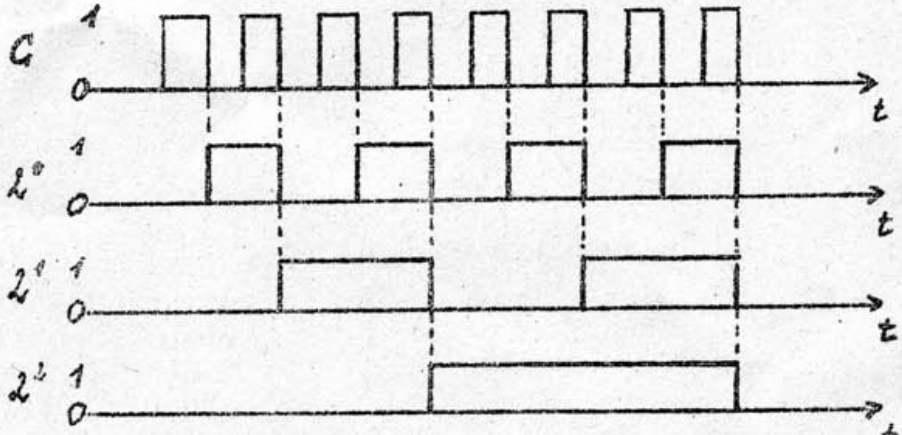
vor der ersten fallenden Taktflanke	1	0	1
nach der ersten fallenden Taktflanke	0	1	0
nach der zweiten fallenden Taktflanke	0	0	1
nach der dritten fallenden Taktflanke	0	0	0

die zu Beginn im Register stehende Information (hier: 3-bit-Information) ist nach dem 3. Takt völlig aus dem Schieberegister "hinausgeschoben" worden.

DUALZÄHLER (BINÄRZÄHLER)

Z	2^2	2^1	2^0
0	0	0	0
1	0	0	1
2	0	1	0
3	0	1	1
4	1	0	0
5	1	0	1
6	1	1	0
7	1	1	1

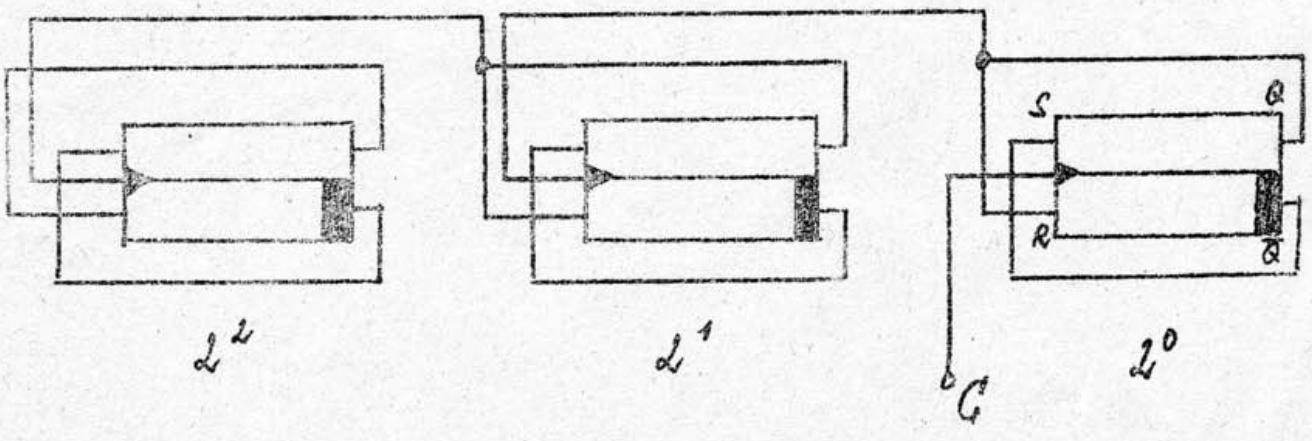
Zustandstabelle



Zeitlicher Verlauf der Ausgangszustände

Für einen Dualzähler mit Zählbereich $0 \ldots 2^n - 1$ benötigt man n FLIP-FLOPs (2^n Zustände).

a) Vorwärtszähler:



Die wesentlichen Komponenten einer **CENTRAL PROCESSING UNIT** (**CPU**) bestehen aus der **CONTROL UNIT** (**CU**) und der **ARITHMETIC LOGIC UNIT** (**ALU**).

Die **ALU** berechnet arithmetische und logische Funktionen, die **CU** decodiert die im Arbeitsspeicher abgelegten Befehle und führt sie aus.

In der Minimalkonfiguration beherrscht die **ALU** die arithmetische Funktion **„Addition"** sowie die logischen Operationen **„Negation"** (NOT) und **„Konjunktion"** (AND). Zu Lasten der Rechenzeit lassen sich die übrigen arithmetischen und logischen Funktionen auf die genannten, minimal verfügbaren Operationen zurückführen.

1. Subtraktion

Die duale Subtraktion

	a_3	a_2	a_1	a_0
−	b_3	b_2	b_1	b_0
	d_3	d_2	d_1	d_0

läßt sich auf eine duale Addition nach folgendem Verfahren zurückführen:

- Bilde das Einerkomplement des Subtrahenden $b_3\ b_2\ b_1\ b_0$, indem man alle Ziffern negiert (invertiert; aus 0 wird 1 und aus 1 wird 0).
- Addiere das Einerkomplement und die Zahl 1 zum Minuenden.
- Das Ergebnis ist die gesuchte Differenz; dabei bleibt der Überlauf unberücksichtigt.

a) Verdeutliche das genannte Verfahren anhand einiger selbst gewählter Beispiele (ein Beweis des Verfahrens ist nicht erforderlich.).

b) Ergänze die Schaltung „4-bit-Paralleladdierer.dsim" so, daß man nach entsprechender Umschaltung wahlweise eine duale Addition oder eine duale Subtraktion durchführen kann.
Hinweise:
   - Ersetze den HA für das least significant bit (LSB) durch einen VA, um erforderlichenfalls eine „1" als Summand einspeisen zu können (wie?).
   - Die Invertierung der Ziffern des Subtrahenden gelingt z. B. durch den geeigneten Einsatz von XOR-Gattern.

2. Weitere Rechenoperationen

Gegeben sind die (im einfachsten Fall positiven ganzzahligen) Operanden a und b. Um zu verdeutlichen, wie man die „höheren" Rechenoperationen mittels geeigneter Iteration auf die Grundoperationen „Addition" und „Subtraktion" zurückführen kann, schreibe und teste ein Python-Programm, welches die Operationen „Multiplikation" (a*b), „Division" (a/b, ganzzahlige Division) und „Potenzierung" (a**b) realisiert.

3. Logische Operationen

Zeige examplarisch, daß sich die logischen Verknüpfungen

a) $a + b$
b) $a \oplus b$
c) $a \cdot (b+\overline{c})$

auf die Operationen NOT und AND zurückführen lassen.

Algorithmus “Grundrechenarten” mit GUI

In der ALU einer CPU sind in der Regel nur die Rechenoperationen “Addition” und “Subtraktion” hardwaremäßig implementiert; die “höheren” Rechenoperationen “Multiplikation”, “Potenzierung” und “Division” werden als iterativ formulierte Maschinenprogramme, also als Software, realisiert. Wir simulieren diese Vorgänge mit dem in Python codierten Algorithmus **Grundrechenarten_GUI.py**, in welchem für die höheren Rechenoperationen iterativ geschriebene Funktionen definiert werden.
Wir führen

- die Multiplikation auf wiederholte Addition,
- die Potenzierung auf wiederholte Multiplikation und
- die (ganzzahlige) Division auf wiederholte Subtraktion

zurück.

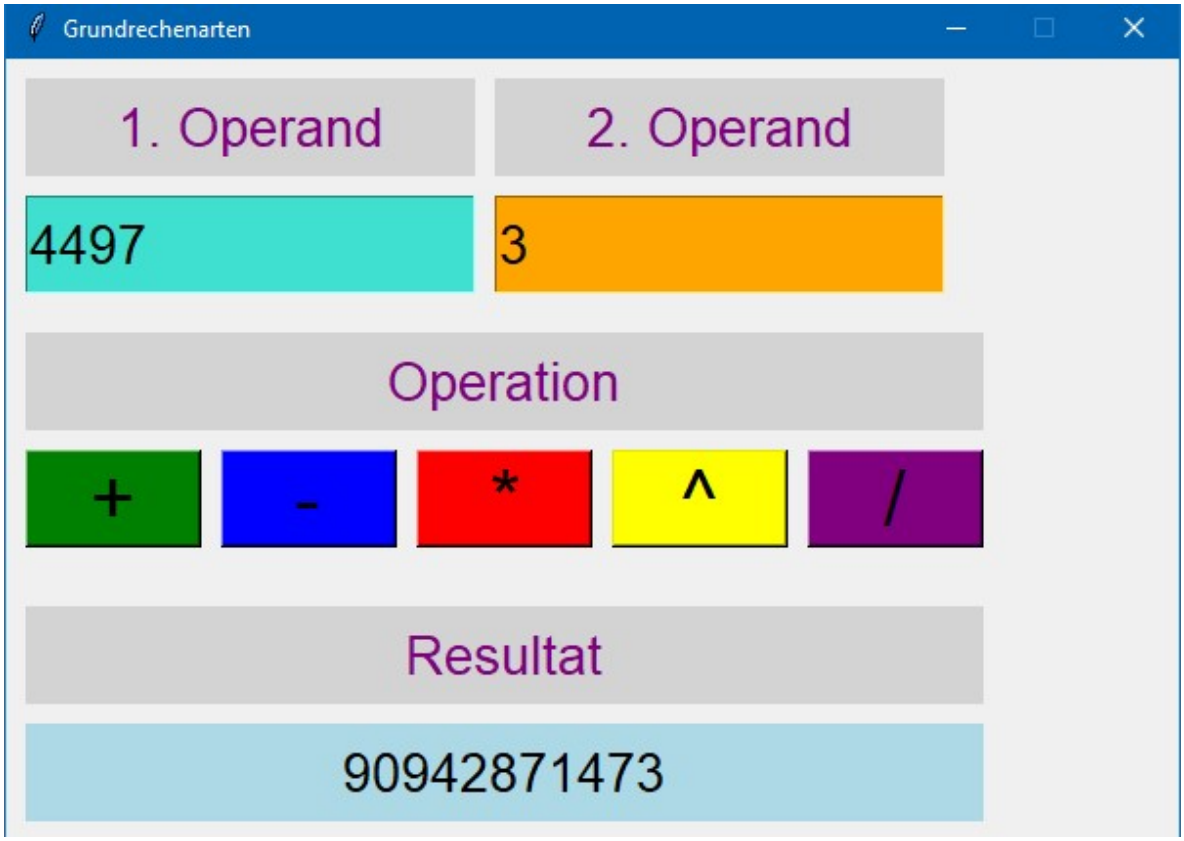
Programmtext in Python:

```
from tkinter import *

# Erstellung eines Fensters
fenster = Tk()
fenster.title('Grundrechenarten')
fenster.geometry('600x400')
fenster.resizable(0,0)


# Funktionen für die Grundrechenarten

def summe(a,b):
      return a + b

def differenz(a,b):
      return a - b

def produkt(a,b):
      ergebnis = 0
      i = 0
      while i <= b - 1:
          ergebnis = summe(ergebnis,a)
          i +=1
      return ergebnis

def potenz(a,b):
      if b == 0: return 1
      else:
        ergebnis = a
        i = 0
        while i <= b - 2:
          ergebnis = produkt(ergebnis,a)
          i = i + 1
        return ergebnis

def quotient(a,b):
      rest = a
      ergebnis = 0
      while rest >= b:
          rest = differenz(rest,b)
          ergebnis += 1
      return ergebnis
```

```
# auszuführende Kommandos (nach Klick auf den entsprechenden Button)

def addiere():
# Die für die Operanden jeweils eingegebenen Zeichenkette (string)
# wird in den Typ "integer" konvertiert und den Variablen
# x und y zugewiesen.
       x = int(entry1.get())
       y = int(entry2.get())
# Zur Berechnung des Egebnisses wird die Funktion "summe" aufgerufen.
# Das Ergebnis wird der Variablen result zugewiesen.
       result = summe(x,y)
# result (vom Typ integer) wird in in eine Zeichenkette (string)
# konvertiert und als text in label5 ausgegeben.
       label5.config(text=str(result))

def subtrahiere():
       x = int(entry1.get())
       y = int(entry2.get())
       result = differenz(x,y)
       label5.config(text=str(result))

def multipliziere():
       x = int(entry1.get())
       y = int(entry2.get())
       result = produkt(x,y)
       label5.config(text=str(result))

def potenziere():
       x = int(entry1.get())
       y = int(entry2.get())
       result = potenz(x,y)
       label5.config(text=str(result))

def dividiere():
       x = int(entry1.get())
       y = int(entry2.get())
       result = quotient(x,y)
       label5.config(text=str(result))


# Eingabe der Operanden

label1 = Label(master=fenster,
              bg='lightgrey',fg='purple',
              text='1. Operand',font=("Arial", 20))
label1.place(x=10, y=10, width=230, height=50)

label2 = Label(master=fenster,bg='lightgrey',fg='purple',
              text='2. Operand',font=("Arial", 20))
label2.place(x=250, y=10, width=230, height=50)


entry1 = Entry(master=fenster, bg='turquoise', font=("Arial", 20))
entry1.place(x=10, y=70, width=230, height=50)

entry2 = Entry(master=fenster, bg='orange', font=("Arial", 20))
entry2.place(x=250, y=70, width=230, height=50)
```

```
# Auswahl der Rechenoperation

label3 = Label(master=fenster,
               bg='lightgrey',
               fg='purple',
               text='Operation',
               font=("Arial", 20))
label3.place(x=10, y=140, width=490, height=50)


button = Button(master=fenster, bg='green', text='+',
                font=("Arial", 30), command = addiere)
button.place(x=10, y=200, width=90, height=50)

button = Button(master=fenster, bg='blue', text='-',
                font=("Arial", 30), command = subtrahiere)
button.place(x=110, y=200, width=90, height=50)

button = Button(master=fenster, bg='red', text='*',
                font=("Arial", 30), command = multipliziere)
button.place(x=210, y=200, width=90, height=50)

button = Button(master=fenster, bg='yellow', text='^',
                font=("Arial", 30), command = potenziere)
button.place(x=310, y=200, width=90, height=50)

button = Button(master=fenster, bg='purple', text='/',
                font=("Arial", 30), command = dividiere)
button.place(x=410, y=200, width=90, height=50)


# labels zur Ausgabe des Resultats

label4 = Label(master=fenster,
               bg='lightgrey',
               fg='purple',
               text='Resultat',
               font=("Arial", 20))
label4.place(x=10, y=280, width=490, height=50)

label5 = Label(master=fenster,
               bg='lightblue',
               fg='black',
               text='',
               font=("Arial", 20))
label5.place(x=10, y=340, width=490, height=50)
```

Nach Ausführen des Programms erhalten wie folgende graphische Benutzeroberfläche (GUI); die jeweils dazugehörenden Programmtextauszüge sind in nahezu derselben Farbe (hellgrau, türkis, orange, grün, blau, rot, gelb, purpur) gehalten wie in der GUI:




```
label1 = Label(master=fenster,
               bg='lightgrey',fg='purple',
               text='1. Operand',font=("Arial", 20))
label1.place(x=10, y=10, width=230, height=50)

label2 = Label(master=fenster,bg='lightgrey',fg='purple',
               text='2. Operand',font=("Arial", 20))
label2.place(x=250, y=10, width=230, height=50)


entry1 = Entry(master=fenster, bg='turquoise', font=("Arial", 20))
entry1.place(x=10, y=70, width=230, height=50)


entry2 = Entry(master=fenster, bg='orange', font=("Arial", 20))
entry2.place(x=250, y=70, width=230, height=50)


label3 = Label(master=fenster,
               bg='lightgrey',
               fg='purple',
               text='Operation',
               font=("Arial", 20))
label3.place(x=10, y=140, width=490, height=50)
```

```
button = Button(master=fenster, bg='green', text='+',
                font=("Arial", 30), command = addiere)
button.place(x=10, y=200, width=90, height=50)


button = Button(master=fenster, bg='blue', text='-',
                font=("Arial", 30), command = subtrahiere)
button.place(x=110, y=200, width=90, height=50)


button = Button(master=fenster, bg='red', text='*',
                font=("Arial", 30), command = multipliziere)
button.place(x=210, y=200, width=90, height=50)


button = Button(master=fenster, bg='yellow', text='^',
                font=("Arial", 30), command = potenziere)
button.place(x=310, y=200, width=90, height=50)


button = Button(master=fenster, bg='purple', text='/',
                font=("Arial", 30), command = dividiere)
button.place(x=410, y=200, width=90, height=50)



label4 = Label(master=fenster,
               bg='lightgrey',
               fg='purple',
               text='Resultat',
               font=("Arial", 20))
label4.place(x=10, y=280, width=490, height=50)


label5 = Label(master=fenster,
               bg='lightblue',
               fg='black',
               text='',
               font=("Arial", 20))
label5.place(x=10, y=340, width=490, height=50)
```

Binäre Suche

Informatik 12 Januar 2022

Gegeben: Ein sortiertes Array **a** mit den **n** Komponenten **a[0], ... , a[n-1]**
Aufgabe: Entscheide, ob ein für die Variable **value** eingegebener Wert als Wert einer Komponente des Arrays **a** vorkommt.

Beispiel

value = 13
n = len(a) = 10

Wir übergeben **value** und die Liste **a[0], . . . , a[9]** der Booleschen Funktion **binarysearch**, welche **a[0], . . . , a[9]** als lokale Liste **array[0], . . . , array[9]** fortführt.

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
3	4	5	5	7	8	11	13	19	21

array[0]	array [1]	array [2]	array [3]	array [4]	array [5]	array [6]	array [7]	array [8]	array [9]
3	4	5	5	7	8	11	13	19	21

1. Schritt:
Wir bestimmen den mittleren Index des Arrays array: len(array)//2 = 5

2. Schritt:
midvalue = array[len(array)//2] = array[10//2] = array[5] = 8
Wir vergleichen value mit midvalue:

Falls value == midvalue: **binarysearch** gibt den Wert **`True`** zurück; gefunden!
Falls value < midvalue: suche in der Liste a[0], . . . , a[4] links von a[5]
Falls value > midvalue: suche in der Liste a[6], . . . , a[9] rechts von a[5]

hier: wegen 13 > 8 suchen wir in der Liste a[6], . . . , a[9]

Suche **value** in der Liste **a[6], . . . , a[9]**

a[6]	a[7]	a[8]	a[9]
11	13	19	21

Diese Liste **a[6], . . , a[9]** und **value** übergeben wir der Booleschen Funktion **binarysearch**, welche **a[6], . . , a[9]** als lokale Liste **array[0], . . . , array[3]** fortführt.

array[0]	array[1]	array[2]	array[3]
11	13	19	21

1. Schritt:
Wir bestimmen den mittleren Index des Arrays array: len(array)//2 = 4//2 = 2

2. Schritt:
midvalue = array[len(array)//2] = array[4//2] = array[2] = 19
Wir vergleichen value mit midvalue:

Falls value == midvalue: **binarysearch** gibt den Wert **`True`** zurück; gefunden!
Falls value < midvalue: suche in der Liste array[0], . . . , array[1] links von array[2]
Falls value > midvalue: suche in der Liste array[3] rechts von array[2]

hier: wegen 13 < 19 suchen wir in der Liste array[0], . . . , array[1]

Suche **value** in der Liste **array[0], . . . , array[1]**

array[0]	array[1]
11	13

Diese Liste **array[0], . . . , array[1]** und **value** übergeben wir der Booleschen Funktion **binarysearch**, welche **array[0], . . . , array[1]** als lokale Liste **array[0], . . . , array[1]** fortführt.

1. Schritt:
Wir bestimmen den mittleren Index des Arrays array: len(array)//2 = 2//2 = 1

2. Schritt:
midvalue = array[len(array)//2] = array[2//2] = array[1] = 13
Wir vergleichen value mit midvalue:
Falls value == midvalue: **binarysearch** gibt den Wert **`True`** zurück; gefunden!
Falls value < midvalue: suche in der Liste array[0] links von array[1]
Falls value > midvalue: suche in der leeren Liste [] rechts von array[1], dann: **binarysearch** gibt den Wert **`False`** zurück; nicht gefunden!

hier: wegen 13 = value = midvalue = 13: **binarysearch** gibt den Wert **`True`** zurück; gefunden!

Der Booleschen Funktion **binarysearch** werden das aus **n** Komponenten bestehende sortierte Feld **a** (in Python: Liste) und der zu suchende Wert **value** übergeben; **binarysearch** liefert den Wert **True**, falls eine Komponente von **a** mit **value** übereinstimmt, andernfalls den Wert **False**.
Die Variable **z** ermittelt die Anzahl der Aufrufe von binarysearch.

Quelltext in Python:

```
. . . . . .

z = 0
. . . . .

def binarysearch(array,value):
     global z
     z += 1
     print(array)
     if array == [] or (len(array) == 1 and array[0] !=  value):
         return False
     else:
         midvalue = array[len(array)//2]
         if midvalue == value:
             return True
         elif value < midvalue:
             return binarysearch(array[:len(array)//2],value)
         else:
             return binarysearch(array[len(array)//2 + 1:],value)
```

Aufruf der Funktion **binarysearch**:

```
binarysearch(a,value)
```

Komplexität des Algorithmus **binarysearch**:

Die Komplexität und damit der Rechenaufwand **A(n)** wird wesentlich bestimmt durch die Anzahl **z** der Aufrufe der rekursiv definierten Funktion **binarysearch**; o. B. d. A. sei **n** eine Potenz von 2, d. h. $\mathbf{n = 2^k}$ mit k = 0, 1, 2, 3, ...
Beachte: die maximale Anzahl von Aufrufen (worst case) tritt ein, falls die Suche ergebnislos ist.

k = 0 ⇔ n = 1

```
# Aufrufe binarysearch = 1
```

k = 3 ⇔ n = 8

```
gesuchte Zahl: 79
[14, 50, 52, 70, 74, 80, 89, 97]
[80, 89, 97]
[80]
79 wurde nicht gefunden
# Aufrufe binarysearch = 3
```

k = 4 ⇔ n = 16

```
gesuchte Zahl: 80
[13, 33, 42, 42, 44, 44, 45, 45, 47, 52, 57, 59, 62, 72, 92, 94]
[52, 57, 59, 62, 72, 92, 94]
[72, 92, 94]
[72]
80 wurde nicht gefunden
# Aufrufe binarysearch = 4
```

Eine Verdopplung von n impliziert höchstens einen weiteren Aufruf von **binarysearch**!

Offensichtlich gilt:

$$z = k$$

Wegen $n = 2^k \Leftrightarrow k = \log_2(n)$ folgt:

$$z = \log_2(n)$$

Somit hat der Algorithmus **binarysearch** logarithmische Komplexität:

$$A(n) \sim \log_2(n)$$

Bemerkung:
Falls im ungünstigsten Fall binarysearch noch die leere Liste [ ] übergeben wird, gilt: z = k + 1

Modifikation des Algorithmus binarysearch:
Die rekursive Funktion **binarysearch** liefert den booleschen Wert **False**, falls **value** nicht gefunden wird, andernfalls den Index **index** der betreffenden Komponente.
Außer **a** und **value** sind die Indices **begin** und **end** an die Funktion **binarysearch** zu übergeben, so daß **binarysearch** die Teilliste **a[begin] , ... , a[end]** durchsucht.
Beachte: **index** wird innerhalb des Funktionsrumpfs als globale Variable definiert.

```
z = 0
. . . . .
def binarysearch(array, value, begin, end):
     global index
     global z
     z += 1
     print(array[begin:end+1])
     if begin > end: return False
     middle = (begin + end) // 2
     print('mittleres Element:  a[',middle,'] = ',array[middle])
     if array[middle] == value:
        index = middle
     elif array[middle] < value:
        return binarysearch(array, value, middle + 1, end)
     else:
        return binarysearch(array, value, begin, middle - 1)
```

Aufruf der Funktion **binarysearch** zur Suche von **value** in der sortierten Liste **a[0], ... , a[n-1]:**

```
binarysearch(a, value, 0, len(a)-1)
```

```
gesuchte Zahl: 521
[120, 162, 163, 181, 205, 392, 444, 521, 528, 557, 643, 663, 689, 810, 847, 899, 913, 992]
mittleres Element:  a[ 8 ] =  528
[120, 162, 163, 181, 205, 392, 444, 521]
mittleres Element:  a[ 3 ] =  181
[205, 392, 444, 521]
mittleres Element:  a[ 5 ] =  392
[444, 521]
mittleres Element:  a[ 6 ] =  444
[521]
mittleres Element:  a[ 7 ] =  521
521 wurde gefunden an der Stelle 7
a[ 7 ] =  521
# Aufrufe binarysearch = 5

gesuchte Zahl: 241
[173, 183, 187, 243, 265, 307, 345, 376, 589, 622, 868, 976]
mittleres Element:  a[ 5 ] =  307
[173, 183, 187, 243, 265]
mittleres Element:  a[ 2 ] =  187
[243, 265]
mittleres Element:  a[ 3 ] =  243
[]
241 wurde nicht gefunden
# Aufrufe binarysearch = 4
```

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 $\mathbf{(1 + x)^n > 1 + n \cdot x}$

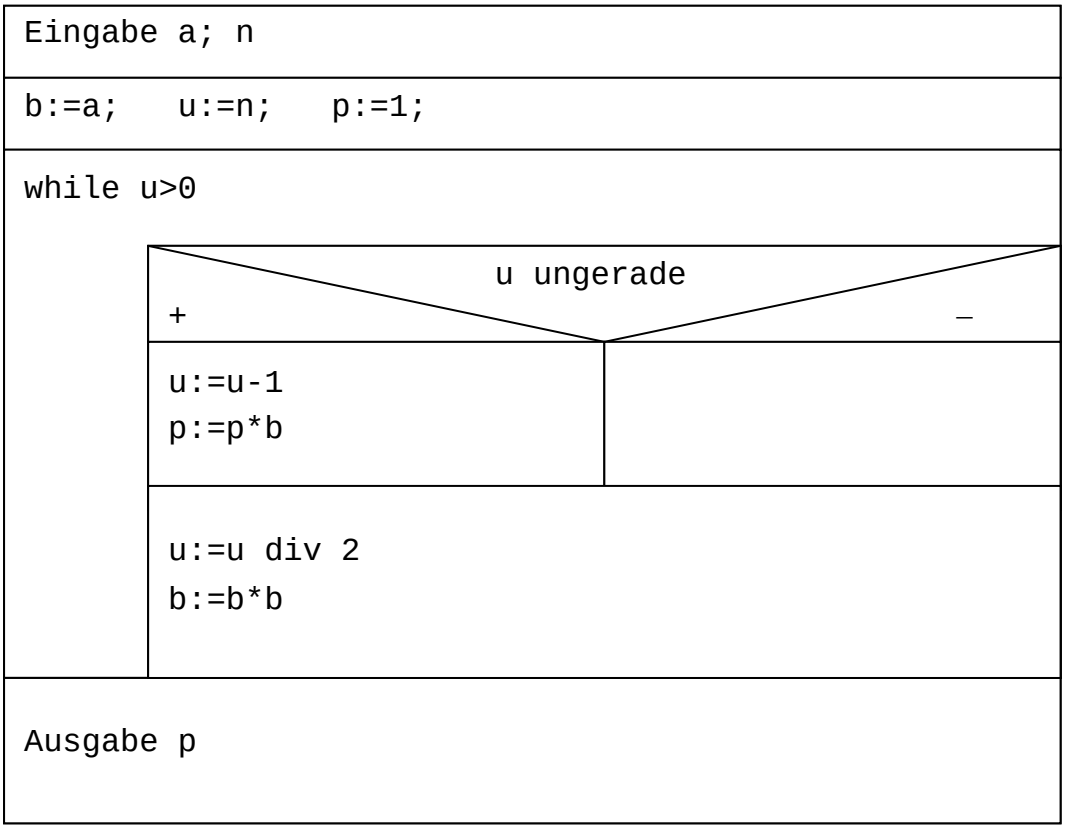
ist wahr für alle natürlichen Zahlen n mit $n \geq 2$ und für reelle Zahlen x mit $x \neq 0$ und $1+x>0$.

(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.

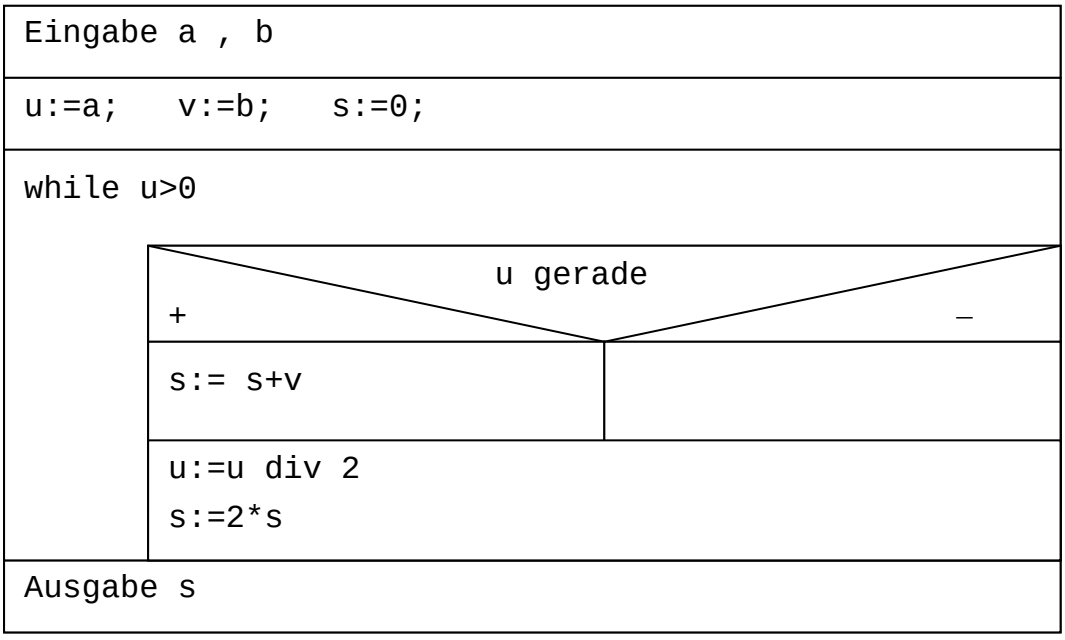
3. Den Potenzierungsalgorithmus „elmo“ kann man modifizieren, indem man die while-Schleife durch folgende Befehlssequenz ersetzt:

```
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.