Algorithmen & Datenstrukturen 1

Fabio Oesch, Claude Martin & Jan Fässler

2. Semester (FS 2012)

Inhaltsverzeichnis

1 Information und Daten

1.1 Binäres Zahlensystem

low: 0 bis 0.8 V
high: 2.4 bis 5 V

1.2 Bit und Byte

- eine Bitfolge der Länge n können 2^n Zustände hergestellt werden
- Bits werden mit Indizes von 0 bis n-1 nummeriert.
- LSB: Bit mit kleinstem Index
- MSB: Bit mit höchstem Index

KiByte: 2^{10} Byte MiByte: 2^{20} Byte GiByte: 2^{30} Byte

1.3 Zahlen in Java

1.3.1 Positive Zahlen

$\sum b_i * B^i$ mit $i \in [0, n-1]$ und b_i gleich der Ziffer bei i.

MSB							LSB
1	0	1	0	0	1	1	1
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0

$1 \cdot 2^7 + 0 \cdot 2^6 + 1 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 167$

1.3.2 Ganze Zahlen (2er - Komplement)

$-b_{n-1} * B^{n-1} + \sum b_i * B^i$ mit $i \in [0, n-2]$ und b_i Ziffer bei i.

MSB							LSB
1	0	1	0	0	1	1	1
-2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0

$-1 \cdot 2^7 + 0 \cdot 2^6 + 1 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = -89$

1.3.3 Zahlendarstellung mit beliebiger Basis

- binär (2): Ganzzahlen, die mit **0b** beginnen
- octal (8): Ganzzahlen, die mit **0** beginnen
- dezimal (10): Ganzzahlen, die **nicht mit 0** beginnen
- hexadezimal (16): Ganzzahlen, die mit **0x** beginnen

1.4 Datentypen in JAVA

Name	Byte	Bit	kleinste Zahl	grösste Zahl
byte	1	8	-128	127
short	2	16	-32.768	32.767
char	2	16	\u0000 (0)	\uFFFF (65.535)
int	4	32	-2.147.483.648	2.147.483.647
long	8	64	-9.223.372.036.854.775.808	9.223.372.036.854.775.807
float	4	32	$1,40 * 10^{-45}$	$3,40282346638528860 * 10^{38}$
double	8	64	$4,94065645841246544 * 10^{-324}$	$1,79769313486231570 * 10^{308}$

1.5 Gleitkommazahlen

Exponentialdarstellung: $(-1)^V * (1 + M) * 2^E$
IEEE 754 Float: $(-1)^V * (1 + M) * 2^{E-127}$
IEEE 754 Double: $(-1)^V * (1 + M) * 2^{E-1023}$

V - Vorzeichen (float 1Bit / double 1 Bit)

M - normalisierte Mantisse ($0 \leq M \leq 1$) (float: 23 Bits / double: 52 Bits)

E - Exponent (float: Signed-8-Bit − 127 / double: Signed-11-Bit − 1023)

V	Exponent	Mantrisse

1.5.1 Beispiele (IEEE 754 32bit Float)

2.5 $= 1.25 * 2^1$

-0.75 $= \underbrace{1}_{\ominus} \underbrace{01111110}_{126-127} \underbrace{100000...}_{1+2^{-1}} = -1.5 * 2^{-1}$

0.1 $= \underbrace{0}_{\oplus} \underbrace{01111011}_{123-127} \underbrace{1001\overline{1100}.....}_{1+2^{-1}+2^{-4}+2^{-5}...} = 1.6 * 2^{-4} = 0.100000001490116119384765625$

Umrechnen von -1313.3125 zu IEEE 32-bit float:

1. Ganzzahlteil $1313_{10} = 10100100001_2$.
2. Nachkommateil
   0.3125 $\times 2 = 0.625$ **0**
   0.625 $\times 2 = 1.25$ **1**
   0.25 $\times 2 = 0.5$ **0**
   0.5 $\times 2 = 1.0$ **1**
   (Ist der Nachkommateil länger als 23 Bit, wird das 23. Bit anhand des 24. Bit gerundet.)
3. Ganzzahlteil.Nachkommateil=$1313.3125_{10} = 10100100001.0101_2$
4. Normen

   Komma verschieben bis nur noch eine 1 vor dem Komma steht. Exponent entspricht der Anzahl verschobener Stellen.
   Verschiebung des Kommas nach links: positiver Exponent.
   Verschiebung des Kommas nach rechts: negativer Exponent.

   $10100100001.0101_2 = 1.01001000010101_2 \times 2^{10}$
5. Mantisse ist 01001000010101, Exponent ist $10 + 127 = 137 = 10001001_2$, Vorzeichen ist 1.
6. -1313.3125 ist

1	10001001	01001000010101000000000

1.5.2 Spezialfälle

E=00000000 M=0 $\Rightarrow 0$

E=11111111 M=0 V=+ $\Rightarrow +\infty$

E=11111111 M=0 V=− $\Rightarrow -\infty$

E=11111111 M≠0 $\Rightarrow NaN$

1.5.3 Division mit Null

- float f = ±0.0f/±0.0f = NaN
- float f = 7.6f/0.0f = -7.6f/-0.0f = $+\infty$
- float f = -3.9f/0.0f = 3.9f/-0.0f = $-\infty$

2 Operationen & Ausdrücke

2.1 Auswertungsreihenfolge

- einstellige und mehrstellige Operatoren
  1. Teilausdrücke in Klammern
  2. Ausdrücke mit unären Operatoren (pro Operand von rechts nach links)
  3. Teilausdrücke mit mehrstelligen Operatoren gemäss Prioritätstabelle
- mehrstellige Operatoren gleicher Priorität
  bei gleicher Priorität entscheidet die Assoziativität (von links nach rechts oder von rechts nach links)
- Bewertungsreihenfolge von Operanden
  die Operanden eines Operators werden strikt von links nach rechts ausgewertet

Priorität	**Operatoren**	**Bedeutung**	**Assoziativität**
1	[]	Array-Index	links
	()	Methodenaufruf	links
	.	Komponentenzugriff	links
	++	Postinkrement	links
	−−	Postdekrement	links
2	++	Präinkrement	rechts
	−−	Prädekrement	rechts
	+ −	Vorzeichen (unär)	rechts
	∼	bitweises Komplement	rechts
	!	logischer Negationsoperator	rechts
3	(type)	Typ-Umwandlungt	rechts
	new	Erzeugung	rechts
4	* / %	Multiplikation, Division, Rest	links
5	+ -	Addition, Subratktion	links
	+	Stringverkettung	links
6	<<	Linksshift	links
	>>	Vorzeichenbehafteter Rechtsshift	links
	>>>	Vorzeichenloser Rechtsshift	links
7	< <=	Vergleich kleiner, kleiner gleich	links
	> >=	Vergleich grösser, grösser gleich	links
	instanceof	Typenüberprüfung eines Objektes	links
8	==	Gleichheit	links
	!=	Ungleichheit	links
9	&	bitweises UND	links
10	^	bitweises Exclusiv-ODER	links
11	|	bitweises ODER	links
12	&& &	logisches UND	links
13	|| |	logisches ODER	links
14	? :	Bedingungsoperator	rechts
15	=	Wertzuweisung	rechts
	*= /= %= += -= <<= >>= >>>= &= ^= |=	kombinierter Zuweisungsoperator	rechts

2.2 Bitoperatoren

a	b	AND a & b	OR a | b	Negation ∼a	XOR a ∧ b
0	0	0	0	1	0
0	1	0	1	1	1
1	0	0	1	0	1
1	1	1	1	0	0

2.3 Schiebeoperatoren

- **Rechtsschiebe-Operatoren**
  - vorzeichenbehaftet: $a >> b$, **Bsp:**$-9_{10} = 11110111_2 >> 1 = 11111011_2 = -5_{10}$
  - vorzeichenlos: $a >>> b$, **Bsp:**$-9_{10} = 11110111_2 >>> 1 = 01111011_2 = 123_{10}$
- **Linksschiebe-Operator**
  - kann Vorzeichen verändern: $a << b$
  - $y = 3 << 2 \Rightarrow y = 00001100_2 = 12$ (Für jeden linksschiebeoperator wird $\cdot 2$ gerechnet.)

2.4 Logische Operatoren

logische Negation: !A

logisches UND: A && B / A & B
(Bei zwei & Zeichen wird erst A überprüft und B nur, falls A wahr ist.)

logisches ODER: A || B / A | B

Bedingungsoperator: A ? B : C (if (A) then B else C)

2.5 Beispiele

Listing 1: OneBitCounter

```
class OneBitCounter {
    public static int count (int x) {
        int iEven, iOdd, d = 1;
        iEven = x & 0x55555555; x >>= d; iOdd = x & 0x55555555;
        x = iOdd + iEven; d <<= 1;
        iEven = x & 0x33333333; x >>= d; iOdd = x & 0x33333333;
        x = iOdd + iEven; d <<= 1;
        iEven = x & 0x0F0F0F0F; x >>= d; iOdd = x & 0x0F0F0F0F;
        x = iOdd + iEven; d <<= 1;
        iEven = x & 0x0000FFFF; x >>= d; iOdd = x & 0x0000FFFF;
        x = iOdd + iEven; d <<= 1;
    }
}
```

Listing 2: Addition

```
class Addition {
    public static int add (int a, int b) {
        int c, r, t;

        r = a ^ b;
        c = a & b;
        while (c != 0) {
            c <<= 1;
            t = r;
            r ^= c;
            c &= t;
        }
        return r;
    }
}
```

Listing 3: Multiplication

```
class Multiplication {
    public static long mult (int a, int b) {
        long y = 0;

        while (a != 0) {
            if (a % 2 == 1) y += b;
            b <<= 1;
            a >>= 1;
        }
        return y;
    }
}
```

3 Konvertierung von Datentypen

3.0.1 explizite Typkonvertierung

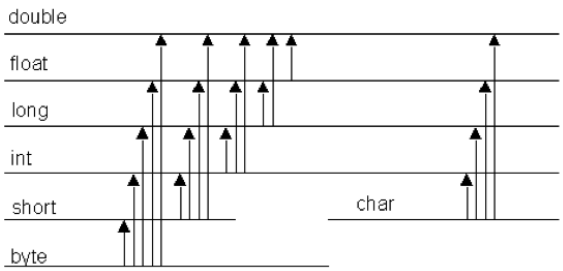
- Konvertierungen sind möglich:
  - zwischen numerischen Datentypen (**erweiternd und einschränkend**)
  - zwischen Referenztypen
- funktionieren gleich wie implizite, allerdings bestehen mehr Möglichkeiten
- cast-Operator: (Typname) Ausdruck

3.0.2 implizite (automatische) Typkonvertierung

- zwischen Operanden von numerischem Typ (**nur erweiternd**)
- zwischen Operanden von Referenztypen
- bei Verknüpfungen von String-Objekten mit Operanden anderer Typen
  Object o = new Object();
  String s = "X"+null+o;
  String s = "X"+"null"+o.toString();
  String s = "Xnulljava.lang.Object@47ac1adf";
- arithmetische Operatoren: Konvertierung in höheren Typ gemäss Hierarchie

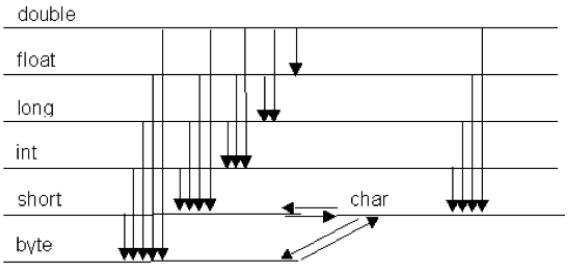
3.0.3 erweiternde Typkonvertierungen

Wert ist immer darstellbar
möglicher Verlust an Genauigkeit (z.B. bei Konvertierung von int nach float)



3.0.4 einschränkende Typkonvertierungen

möglicher Informationsverlust in Grösse, Vorzeichen & Genauigkeit



3.0.5 Integer-Erweiterung

- Datentypen byte, short und char werden in Ausdrücken mit unären und binären Operatoren implizit in int konvertiert
  - Dimensionsausdruck bei der Erzeugung von Arrays
  - Indexausdruck in Arrays
  - Operand der unären Operatoren + und −
  - Operand des Invertierungsoperators für Bits ∼
  - Operanden der Schiebeoperatoren >>, >>> und <<
- byte, short und char werden somit fast ausschliesslich als Datenfelder für Klassen benutzt

3.0.6 Konvertierungsvorschriften

- erweiternde Konvertierung von vorzeichenbehafteten Integer-Typen
  Wert bleibt unverändert
- Konvertierung zwischen char und vorzeichenbehafteten Integer-Typen
  - char ist vorzeichenlos
  - short: Bitmuster bleibt erhalten, da gleiche Breite. Negativ wenn $b_{15} = 1$.
  - char zu int: von links mit Nullen auffüllen. Positiv.
  - char zu byte: Bits 0 bis 7 werden übernommen. Negativ wenn $b_7 = 1$.
- Konvertierung von Integer nach Gleitpunkt
  nächst höherer oder niedriger darstellbarer Wert
- Konvertierung von Gleitpunkt nach Integer
  - Nachkommastellen werden abgeschnitten
  - bei Werten grösser als $2^{31} - 1$ ist das Resultat nicht korrekt ($= 2^{31} - 1$)
- Konvertierung zwischen Gleitpunkt-Typen
  - float nach double: Wert bleibt unverändert
  - double nach float: Wert im zulässigen Wertebereich von float, dann nächst höherer oder niedriger darstellbarer Wert

3.1 Beispiele

Umwandlung	**Resultat**
(int) Double.MAX_VALUE	Integer.MAX_VALUE
(int) (float) Integer.MAX_VALUE	Integer.MAX_VALUE
(int) (float) (Integer.MAX_VALUE - 1)	Integer.MAX_VALUE (Genauigkeit ist nicht gegeben!)
(int) Long.MAX_VALUE	-1
(int) Long.MIN_VALUE	0
(byte)(char) 254	-2
(char)(byte) -2	65534 (falsch)
(char)(byte) -2 & 0xff	254 (korrekt)

4 Zeichen und Strings

4.1 Zeichen in Java

- int: 32-Bit im Format: nur die unteren 21 Bits werden benutzt, die oberen sind alle 0; eine int-Variable kann jeden möglichen Zeichencode (code point) des Unicodes aufnehmen
- char: 16 Bit; eine char-Variable kann nur einen der ersten 2^{16} Zeichencodes des Unicodes aufnehmen; alle anderen Zeichencodes werden durch char-Paare codiert (surrogate character mechanism)
- Interface CharSequence: Sequenz von Zeichencodes im UTF-16 Format; bekannte Implementierung→ String, StringBuffer, StringBuilder, CharBuffer ...

4.2 Standardisierte Zeichencodierungen

4.2.1 ASCII (American Standard Code for Information Interchange)

- nur 128 verschiedene Zeichen (Steuerzeichen, Satzzeichen, Ziffern, Buchstaben usw.)
- pro Zeichen ein eindeutiger 7-Bit-Zeichencode

4.2.2 ISO 8859-1 (Latin-1)

- 256 Zeichen für europäische Sprachen (8-Bit-Zeichencode)
- die ersten 128 Zeichen sind identisch zu ASCII
- die zweiten 128 Zeichen enthalten europäische Sonderzeichen, z.B. Umlaute

4.2.3 Unicode

- über eine Million Zeichen, verschiedene Sprachen (Arabisch, Hebräisch, Chinesisch usw.)
- 21-Bit-Zeichencode: ersten 256 Zeichen sind identisch zu ISO 8859-1

4.3 Universal Character Encoding Standard (Unicode)

Ebene 0: Basic Multilingual Plane (BMP)

- U+0 bis U+FFFF
- enthält die wichtigsten Zeichen von verschiedenen Sprachen

Ebene 1: Supplementary Multilingual Plane (SMP)

- U+1'0000 bis U+1'FFFF
- enthält weniger oft gebrauchte Zeichen (z.B. gotische Zeichen, Musiksymbole)

Ebene 2: Supplementary Ideographic Plane (SIP)

- U+2'0000 bis U+2'FFFF
- enthält sehr seltene CJK Zeichen

Ebene 3 bis 13
reserviert für spätere Ergänzungen

Ebene 14: Supplementary Special-purpose Plane (SSP)

- U+E'0000 bis U+E'FFFF
- enthält zusätzliche Formatierungszeichen

Ebenen 15 und 16: Private Use Planes

- U+F'0000 bis U+F'FFFF und U+10'0000 bis 10'FFFF
- für private, nicht standardisierte Zwecke einsetzbar

4.3.1 UTF-32

- einfachste Umsetzung: die 21 Bits werden in einem (vorzeichnlosen) 32-Bit-Integer abgespeichert; die höherwertigen 11 Bits sind immer null
- Vorteile: einfache Umsetzung; einheitliche Codierungslänge
- Nachteil: hohe Speicherverschwendung
- Bsp:
  A: U+0041 → 0000'0041
  Ω: U+03A9 → 0000'03A9

4.3.2 UTF-16

- Umsetzung: Zeichencodes der BMP werden durch eine einzelne 16-Bit Codierungseinheit gespeichert; Zeichencodes der zusätzlichen Ebenen werden durch zwei 16-Bit Codierungseinheiten gespeichert → Ersatzpaare; guter Kompromiss zwischen Speicherbedarf und einfacher Handhabung
- Ersatzpaare: beide Zeichen eines Ersatzpaares stammen aus der Surrogates Area; das erste Zeichen jeweils aus dem Bereich U+D800 bis U+DBFF; das zweite Zeichen jeweils aus dem Bereich U+DC00 und U+DFFF
- Bsp: U+1'0384 → D800, DF84

4.3.3 UTF-8

- Bekannte Umsetzung: populäre Verwendung in XML; Zeichencodes brauchen zwischen 1 und 4 Bytes
- Vorteile: speichereffizient für häufig verwendete Zeichen; kompatibel mit ASCII
- Nachteile: speicherineffizient für seltene Zeichen, jedoch nicht schlimmer als UTF-32; komplizierte Handhabung, weil unterschiedliche Anzahl von Bytes pro Zeichen beachtet werden muss; nicht kompatibel mit ISO-Latin-1, d.h. Umlaute brauchen bereits 2 Bytes
- Bsp:
  A: U+0041 → 41;
  Ω: U+03A9 → CE, A9;
  Ugaritic Delta: U+10384 → F0, 90, 8E, 84

Zeichencode	Byte 1	Byte 2	Byte 3	Byte 4
0xxxxxxx	0xxxxxxx →1 byte →7 bit (ASCII)			
00000yyy yyxxxxxx	110yyyyy →2 byte →11 bit	10xxxxxx		
zzzzyyyy yyxxxxxx	1110zzzz →3 byte →16bit (BMP)	10yyyyyy	10xxxxxx	
000uuuzz zzzzyyyy yyxxxxxx	11110uuu →4 byte →21bit (alle)	10zzzzzz	10yyyyyy	10xxxxxx

4.3.4 Codierungsschemen

Das bestimmt Codierungsformat und Byte-Reihenfolge:

- bei Dateneinheiten bestehend aus mehreren Bytes muss festgehalten werden, welches Byte zuerst behandelt wird, das
  - most significant byte (MSB) zuerst: big-endian
  - least significant byte (LSB) zuerst: little-endian
- relevant bei UTF-16 und UTF-32
- nicht relevant bei UTF-8 (die Reihenfolge ist klar)
- durch Verwendung des Spezialzeichens Byte Order Mark (BOM) kann die Byte-Reihenfolge automatisch detektiert werden:

4.4 Java

4.4.1 Unicode in JAVA

Datentyp int

- 32-Bit im UTF-32 Format: nur die unteren 21 Bits werden benutzt, die oberen sind alle 0
- eine int-Variable kann jeden möglichen Zeichencode (code point) des Unicodes aufnehmen

Datentyp char

- 16-Bit (Codierungseinheit, code unit)
- eine char-Variable kann nur einen der ersten 2^{16} Zeichencodes des Unicodes (Basic Multilingual Plane, BMP) aufnehmen
- alle anderen Zeichencodes werden durch char-Paare codiert

Interface CharSequence

- Sequenz von Zeichencodes im UTF-16 Format
- bekannte Implementierungen: String, StringBuffer, StringBuilder

4.4.2 Zeichenketten (Strings)

Character-Array
veränderbare Zeichenkette mit expliziter Längenangabe (Teil des Arrays)

Klasse String (UTF-16 Format)
unveränderbare Zeichenkette mit expliziter Längenangabe

Klassen StringBuilder und StringBuffer (UTF-16 Format)
veränderbare Zeichenkette gekapselt als abstrakter Datentyp

4.4.3 String Funktionen

String equals(String vergleichenMit)
vergleicht den Inhalt von zwei Strings

String substring(int anfang, int ende)
Schneidet eine Zeichenkette zwischen Anfang und Ende 1 aus und erzeugt damit ein neues Stringobjekt. Der ursprüngliche String wird dabei nicht verändert.

String trim()
Erzeugt Kopie und entfernt alle Leerzeichen am Anfang und Ende der Kopie. Die Kopie wird zurückgegeben.

4.4.4 Umwandlungen

Listing 4: UTF 16 zu Latin1

```
static byte[] utfToLatin1(String s) {
  byte[] array = new byte[s.length()];
  int j = 0;
  for (int i = 0; i < s.length(); i++) {
    char c = s.charAt(i);
    if (c >= 256 && (c < 0xdc00 || c >= 0xdfff)) {
      array[j++] = (byte) '?';
    } else if (c < 256) {
      array[j++] = (byte) c;
    } //else: Low-Surrogate
  }
  for (int i = j; i < s.length(); i++) {
    array[j] = (byte) 0;
  }
  return array;
}
```

Listing 5: UTF 32 zu UTF 16

```
static char[] codepoint2chars(int cp) {
  assert Character.isValidCodePoint(cp) : "illegal code point";
  if (cp < 0x10000) {
    if (cp >= 0xD800 && cp <= 0xDFFF)
      throw new IllegalArgumentException("illegal code point");
    return new char[] { (char) (cp & 0xFFFF) };
  } else {
    if (cp > 0x10FFFF)
      throw new IllegalArgumentException("illegal code point");
    cp -= 0x10000;
    char c1 = (char) ((cp >> 10) | 0xD800);
    char c2 = (char) ((cp & 0x3FF) | 0xDC00);
    return new char[] { c1, c2 };
  }
}
```

4.5 ASCII-Zeichentabelle

Dez	Hex	Okt	Zeichen	Dez	Hex	Okt	Zeichen
0	0x00	000	NUL	32	0x20	040	SP
1	0x01	001	SOH	33	0x21	041	!
2	0x02	002	STX	34	0x22	042	“
3	0x03	003	ETX	35	0x23	043	#
4	0x04	004	EOT	36	0x24	044	$
5	0x05	005	ENQ	37	0x25	045	%
6	0x06	006	ACK	38	0x26	046	&
7	0x07	007	BEL	39	0x27	047	’
8	0x08	010	BS	40	0x28	050	(
9	0x09	011	TAB	41	0x29	051	)
10	0x0A	012	LF	42	0x2A	052	*
11	0x0B	013	VT	43	0x2B	053	+
12	0x0C	014	FF	44	0x2C	054	,
13	0x0D	015	CR	45	0x2D	055	−
14	0x0E	016	SO	46	0x2E	056	.
15	0x0F	017	SI	47	0x2F	057	/
16	0x10	020	DLE	48	0x30	060	0
17	0x11	021	DC1	49	0x31	061	1
18	0x12	022	DC2	50	0x32	062	2
19	0x13	023	DC3	51	0x33	063	3
20	0x14	024	DC4	52	0x34	064	4
21	0x15	025	NAK	53	0x35	065	5
22	0x16	026	SYN	54	0x36	066	6
23	0x17	027	ETB	55	0x37	067	7
24	0x18	030	CAN	56	0x38	070	8
25	0x19	031	EM	57	0x39	071	9
26	0x1A	032	SUB	58	0x3A	072	:
27	0x1B	033	ESC	59	0x3B	073	;
28	0x1C	034	FS	60	0x3C	074	<
29	0x1D	035	GS	61	0x3D	075	=
30	0x1E	036	RS	62	0x3E	076	>
31	0x1F	037	US	63	0x3F	077	?
Dez	Hex	Okt	Zeichen	Dez	Hex	Okt	Zeichen
64	0x40	100	@	96	0x60	140	‘
65	0x41	101	A	97	0x61	141	a
66	0x42	102	B	98	0x62	142	b
67	0x43	103	C	99	0x63	143	c
68	0x44	104	D	100	0x64	144	d
69	0x45	105	E	101	0x65	145	e
70	0x46	106	F	102	0x66	146	f
71	0x47	107	G	103	0x67	147	g
72	0x48	110	H	104	0x68	150	h
73	0x49	111	I	105	0x69	151	i
74	0x4A	112	J	106	0x6A	152	j
75	0x4B	113	K	107	0x6B	153	k
76	0x4C	114	L	108	0x6C	154	l
77	0x4D	115	M	109	0x6D	155	m
78	0x4E	116	N	110	0x6E	156	n
79	0x4F	117	O	111	0x6F	157	o
80	0x50	120	P	112	0x70	160	p
81	0x51	121	Q	113	0x71	161	q
82	0x52	122	R	114	0x72	162	r
83	0x53	123	S	115	0x73	163	s
84	0x54	124	T	116	0x74	164	t
85	0x55	125	U	117	0x75	165	u
86	0x56	126	V	118	0x76	166	v
87	0x57	127	W	119	0x77	167	w
88	0x58	130	X	120	0x78	170	x
89	0x59	131	Y	121	0x79	171	y
90	0x5A	132	Z	122	0x7A	172	z
91	0x5B	133	[	123	0x7B	173	{
92	0x5C	134	\	124	0x7C	174	|
93	0x5D	135	]	125	0x7D	175	}
94	0x5E	136	^	126	0x7E	176	∼
95	0x5F	137	_	127	0x7F	177	DEL

5 Suchen

5.1 Zahlensuche

Generell gilt hier: Ordnung reduziert den Suchaufwand!

5.1.1 Lineare Suche

Listing 6: Lineare Suche

```
int i=0;
while(i<array.length && array[i]!=x) i++;
boolean gefunden = (i < array.length);
```

5.1.2 Lineare Suche mit Wächter (Sentinel)

Listing 7: Lineare Suche mit Waechter

```
boolean gefunden = false;
int last = (array.length-1);
if (array[last]==x) {
  gefunden=true;
} else {
  int tmp = array[last];
  array[last]=x;
  int i=0;
  while(array[i] != x) i++;
  gefunden = (i < last);
  array[last] = tmp;
}
```

5.1.3 Binäre Suche

Listing 8: Binaere Suche

```
boolean binarySearch(double[] array, double x) {
  int first=0, last=array.length-1, m;
  while(first <= last) {
    m = first + (last - first) / 2;  //schneller (m=(first+last)>>>1)
    if(array[m] == x) return true;
    else if (array[m] < x) first=m+1;
    else last=m-1;
  }
}
```

5.1.4 Analyse

Typ	Laufzeit	n	256	2^{20}
Lineare Suche	lineare Laufzeit	n	256	2^{20}
Binäre Suche	logarithmisch	$[log_2(n)] + 1$	9	21

5.2 Textsuche

5.2.1 Naive Textsuche

text:	a	e	e	i	e	i	n	(n Zeichen)
pattern:	e	i	n					(m Zeichen)
	↯	e	i	n				
		✓	↯					
			e	i	n			
			✓	✓	↯			
				e	i	n		
				↯	e	i	n	
					✓	✓	✓	

Auswertung: $T(n,m) = m(n-m+1) = m*n-m^2+m$

5.2.2 Knuth-Morris-Pratt (KMP)

Als erstes wird beim KMP das zu suchende Pattern untersucht. Das Pattern wird mit sich selber verglichen. Das Ziel ist zu wissen, bei welchem Buchstaben des Patterns man bei einem Missmatch weitermachen soll.
Es wird für jedes Teilstück des Patterns das Endstück maximaler Länge gesucht, welches einem Anfangsstück entspricht. Dann wird abgespeichert, wo mit dem Vergleichen fortgefahren werden muss, wenn an dieser Stelle ein Fehler auftritt.

0	1	2	3	4	5	6	7
h	a	u	s	h	a	l	t
-1	0	0	0	0	1	2	0

0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3
P	A	R	T	I	C	I	P	A	T	E		I	N		P	A	R	A	C	H	U	T	E
-1	0	0	0	0	0	0	0	1	2	0	0	0	0	0	0	1	2	3	0	0	0	0	0

Die Tabelle Zeigt die Verschiebeinformationen für das Beispiel. Wenn beim Vergleich der Position 5 mit dem Text ein Fehler auftritt, dann muss bei Position 1 (also 1 wieder verglichen werden). Nur für den Anfangsbuchstaben ist der Wert -1. Wenn dieser genommen wird, heisst das, dass das Pattern wieder von vorne verglichen werden muss.
Diese Verschiebeinformationen werden zu Beginn der Suche im Text berechnet. Damit muss bei der Suche nie ein Teilstück zweimal durchsucht werden.

Implementierung:

Listing 9: Knuth-Morris-Pratt

```
public class KPM {
    String m_text;
    String m_pattern;
    int[] m_next;

    public KPM(String text, String pattern) {
        m_text = text;
        m_pattern = pattern;
        m_next = new int[pattern.length()];
    }

    public void initnext() {
        int i = 0; // wird das Pattern einmal durchlaufen
        int j = -1; // wird zum Vergleich mit dem Anfagsst ck verwendet

        m_next[i] = j;
        while (i < m_pattern.length() - 1) {
            if (j < 0 || m_pattern.charAt(i) == m_pattern.charAt(j)) {
                i++;
                j++;
                m_next[i] = j;

            } else {
                j = m_next[j];
            }
        }
    }

    public void search_kmp() {
        int t = 0;
        int p = 0;

        while (t < m_text.length()) {
            // p weiss, mit was im Pattern verglichen werden muss
            // t geht durch den Text
            if ((p < 0) || m_text.charAt(t) == m_pattern.charAt(p)) {
                t++;
                p++;
            } else {
                p = m_next[p];
            }
            if (p == m_pattern.length()) {
                System.out.println("Gefunden, Uebereinstimmung startet bei "
                        + (t - p + 1) + ". Zeichen");
                p = 0;
            }
        }
    }

    public static void main(String[] args) {
         KPM kpm = new KPM("baabcabac", "bac");
         kpm.initnext();
         kpm.search_kmp();
    }
}
```

6 Sortieren

Ein Array a ist sortiert, wenn gilt:
$\forall i \in [0, a.lenght - 2] : a[i]\ relop\ a[i+1]$
relop: Relation, Binäres Prädikat (typisch: $\leq, \geq, <, >$)

6.1 Überprüfen, ob das Array sortiert ist

Listing 10: Ist Array sortiert?

```
i=0;
while(i<a.lenght-1 && a[i] relop a[i+1]) i++;
boolean sortiert = (i==a.length-1);
```

Aufwand: linear in der Länge des Arrays

6.2 Sortieren durch direktes Auswählen (selection sort)

Das Array wird durch gegangen und nach dem grössten Element durchsucht. Einmal gefunden, wird es hinten hingesetzt. Danach wird das zweitgrösste Element gesucht und vor das grösste Element gestellt. Wenn man dies weiterführt wächst der sortierte Teil des Arrays kontinuierlich, während der unsortierte Teil kleiner wird.

Listing 11: Selection Sort

```
int k, max; //k = index of max ; max = value of max
for (int last = a.length - 1; last > 0; last--) {
  k = 0;
  max = a[k];
  for (int j = 1; j <= last; j++) {
    if (a[j] > max) {
      max = a[k = j];
    }
  }
  if(k!=last){
    a[k] = a[last];
    a[last] = max;
  }
}
```

Worst-Case Aufwand: $T(n) = 1+2+3+...+(n-3)+(n-2)+(n-1) = \sum_{k=1}^{n-1} k = \frac{n(n-1)}{2} = \frac{n^2-n}{2} \approx \underline{\underline{\frac{n^2}{2}}}$

6.3 Sortierten durch direktes einfügen (insertion sort)

Das Array wird durch gegangen und jedes Element an der richtigen Stelle, der bereits durch gegangenen Elemente, eingesetzt.

Listing 12: Insertion Sort

```
for(int first=1; first<a.length;first++) {
  tmp=a[first];
  k=first-1;
  while(k>=0 && a[k]>tmp) {
    a[k+1]=a[k];
    k--;
  }
  a[k+1] = tmp;
}
```

Worst-Case Aufwand: $T(n) = 1+2+3+...+(n-3)+(n-2)+(n-1) = \sum_{k=1}^{n-1} k = \frac{n(n-1)}{2} = \frac{n^2-n}{2} \approx \underline{\underline{\frac{n^2}{2}}}$

Durchschnitt: $T(n) = 0.5+1+1.5+...+\frac{(n-3)}{2}+\frac{(n-2)}{2}+\frac{(n-1)}{2}+\frac{n}{2} = \sum_{i=1}^{n-1} \frac{i}{2} = \frac{1}{2} * \sum_{i=1}^{n-1} i = \frac{1}{2} * (\frac{n^2-n}{2}) \approx \underline{\underline{\frac{n^2}{4}}}$

6.4 Enumeration Sort

Durchlaufe das gegebene Array a und vermerke jedes Auftreten eines Wertes u in einem Hilfsarray t an der Position p, wobei p sehr einfach aus u berechnet werden kann; d.h. t[p] muss zu Beginn 0 sein und wird mit jedem Auftreten von u um eins erhöt.
Durchlaufe t und schreibe jeweils fortlaufend t[p] mal den wert u in das Array a.

Listing 13: Enumeration Sort

```
public static void sort(byte[] a) {
        int[] t = new int[256];
        int pos = 0;
        for (byte b: a) {
                t[b+128] ++;
        }
        for (int i=0; i< t.length; i++) {
                for (int k = 0; k<t[i]; k++) {
                        a[pos++] = (byte) (i-128);
                }
        }
}
```

6.5 Quicksort (randomisierter Algorightmus)

0. falls A.length <= 1: Array A ist schon sortiert
1. wähle zufällig ein Pivotelement p aus A und teile A wie folgt auf:

   A_1: enthält nur Elemente aus A \ p, die $<= p$ sind

   A_2: enthält nur Elementa us A \ p, die $>= p$ sind
2. quicksort(A_1)
   quicksort(A_2)
3. $A = [A_1, p, A_2]$

6.5.1 Implementierung

Listing 14: Quicksort

```
void quicksort (int[] a) {
  if (a.length > 1) sort(a,0,a.length-1);
}
void sort(int[] a, int l, int r) {
  int i=l, j=r;
  int p = Math.random(a, l, r),
  do {
    while (a[i] < p) i++;
    while (a[j] > p) j--;
    if (i <= j) {
      int tmp = a[i];
      a[i] = a[j];
      a[j] = tmp;
      i++;
      j--;
    }
  } while (i < j);
  if (l < j) sort(a, l, j);
  if (i < r) sort(a, i, r);
}
```

6.5.2 Aufwand Quicksort

Worstcase: Es wird immer das Element ganz links (bzw. ganz rechts) genommen

$$
\begin{aligned}
T(1) &= 0 \\
T(n) &= c_1 + c_2 * n + T(1) + T(n-1) = c_1 + c_2 * n + 0 + T(n-1) \\
&= c_1 + c_2 * n + (c_1 + c_2 * (n-1) + T(n-2)) \\
&= 2 * c_1 + c_2 * ((n-2) + (n-1) + n) + T(n-3) \\
&= 2 * c_1 + c_2 * ((n-1) + n) + (c_1 + c_2 * (n-2) + T(n-3)) \\
&= 3 * c_1 + c_2 * ((n-2) + (n-1) + n) + T(n-3) \\
&= i * c_1 + c_2 * \sum_{k=0}^{i-1} (n-k) + T(n-i) \\
&= i * c_1 + c_2 * (n * \sum_{k=0}^{i-1} 1 - \sum_{k=0}^{i-1} k) + T(n-1) \\
&\Rightarrow \text{ sei nun } i = n-1 \\
&= c_1 * (n-1) + c_2 * (n * (n-1) - \frac{(n-2) * (n-1)}{2}) + T(1) \\
&= c_1 * (n-1) + c_2 * (n^2 - n - \frac{n^2}{2} + \frac{3 * n}{2} - 1) \\
&= c_1 * (n-1) + c_{15} : 362 * (\frac{n^2}{2} + \frac{n}{2} - 1) \\
T(n) &\in O(n^2)
\end{aligned}
$$

Bestcase: Das Array nimmt immer das Element, dass in der Mitte des Arrays ist

$$
\begin{aligned}
T(1) &= 0 \\
T(n) &= c_1 + c_2 * n + 2 * T(\frac{n}{2}) \\
&= 2 * T(\frac{n}{2}) + c_2 * n + c_1 \\
&= 2 * (2 * T(\frac{n}{4}) + c_2 * \frac{n}{2} + c_1) + c_2 * n + c_1 \\
&= 4 * T(\frac{n}{4}) + c_4) + c_2 * n + 2 * c_1 + c_2 * n + c_1 \\
&= 4(2 * T(\frac{n}{8}) + c_2 * \frac{n}{4} + c_1) + 2 * c_2 * n + 3 * c_1 \\
&= 8 * T(\frac{n}{8}) + 3 * c_2 * n + 7 * c_1 \\
&\Rightarrow 2^i = n \text{ und } i = ld(n) \text{ werden ersetzt} \\
&= 2^i * T(\frac{n}{2^i}) + i * c_2 * n + \sum_{k=0}^{i-1} 2^k * c_1 \\
&= n * T(\frac{n}{n}) + ld(n) * c_2 * n + c_1 * \sum_{k=0}^{ld(n)-1} 2^k \\
&= c_2 * n * ld(n) + c_1 * (2^{ld(n)} - 1) \\
&= c_2 * n * ld(n) + c_1 * (n-1) \\
&= n * (c_2 * ld(n) + c_1) - c_1 \\
T(n) &\in O(n \cdot log\ n)
\end{aligned}
$$

6.6 Mergesort

Listing 15: Mergesort

```
public class Mergesort {
    private int[] numbers;
    private int[] helper;
    private int number;

    public void sort(int[] values) {
        this.numbers = values;
        number = values.length;
        this.helper = new int[number];
        mergesort(0, number - 1);
    }
    private void mergesort(int low, int high) {
        // Check if low is smaller then high, if not then the array is sorted
        if (low < high) {
            // Get the index of the element which is in the middle
            int middle = (low + high) / 2;
            // Sort the left side of the array
            mergesort(low, middle);
            // Sort the right side of the array
            mergesort(middle + 1, high);
            // Combine them both
            merge(low, middle, high);
        }
    }
    private void merge(int low, int middle, int high) {
        // Copy both parts into the helper array
        for (int i = low; i <= high; i++) {
            helper[i] = numbers[i];
        }
        int i = low;
        int j = middle + 1;
        int k = low;
        // Copy the smallest values from either the left or the right side back
        // to the original array
        while (i <= middle && j <= high) {
            if (helper[i] <= helper[j]) {
                numbers[k] = helper[i];
                i++;
            } else {
                numbers[k] = helper[j];
                j++;
            }
            k++;
        }
        // Copy the rest of the left side of the array into the target array
        while (i <= middle) {
            numbers[k] = helper[i];
            k++;
            i++;
        }
    }
}
```

6.7 Laufzeiten

Sortierverfahren	Best-Case	Average-Case	Worst-Case	Zusätzlicher Speicherplatz
Bubblesort	$O(n \cdot \log(n))$	$O(n \cdot \log(n))$	$O(n^2)$	
Insertionsort	$O(n)$	$O(n^2)$	$O(n^2)$	
Mergesort	$O(n \cdot \log(n))$	$O(n \cdot \log(n))$	$O(n \cdot \log(n))$	bei Array $O(n)$ bis $O(n \cdot \log(n))$
Quicksort	$O(n \cdot \log(n))$	$O(n \cdot \log(n))$	$O(n^2)$	$O(n \cdot \log(n))$ für Stack
Selectionsort	$O(n^2)$	$O(n^2)$	$O(n^2)$	

7 Halbdynamische Datenstrukturen

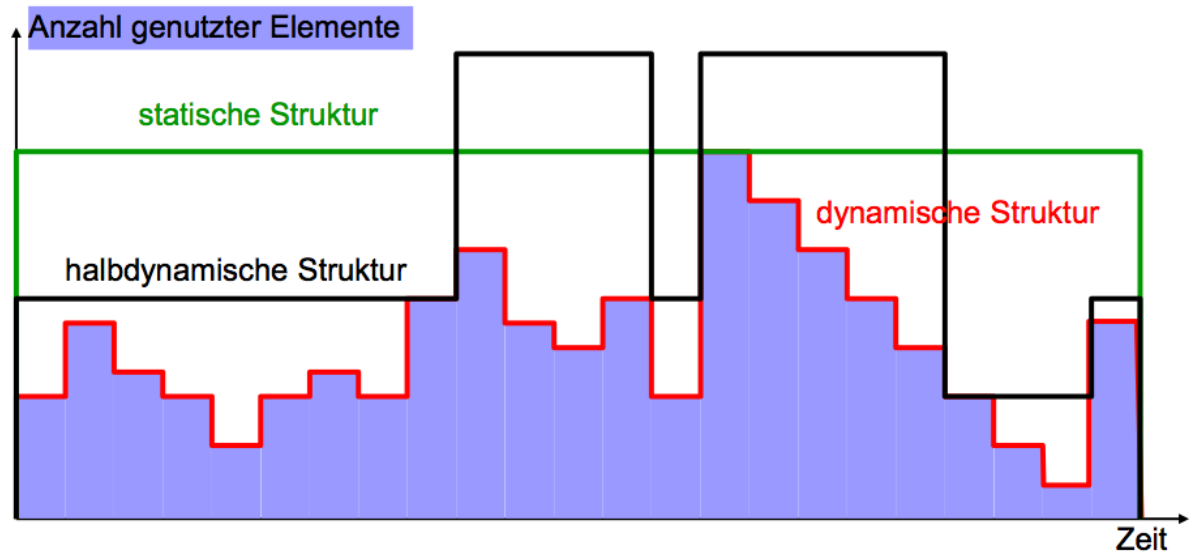
7.1 Halbdynamische Datenstruktur

statische Datestruktur
- fixer Speicherbedarf, unabhängig von der Anzahl genutzter Elemente im Array
- direkter Zugriff auf die Elemente des Arrays

halbdynamische Datenstruktur
- Speicherbedarf passt sich schrittweise der genutzten Anzahl Elemente an und bleibt zwischendrin konstant
- direkter Zugriff auf die Elemente
- Anpassung des Speicherbedarfs ist zeitaufwändig

dynamische Datenstrukturen (Liste, Stack, Baum usw.)
- Speicherbedarf hängt direkt von der genutzten Anzahl Elemente ab
- üblich unterstützte Operationen: Suchen, Einfügen, Entfernen
- indirekter Zugriff auf die Elemente

7.2 Datennutzung über die Zeit



7.3 Datenstruktur: ArrayList

Implementiert das Interface list

- ArrayList verwendet intern ein Array von Objekten
- die Länge des Arrays entspricht der Kapazität
- die verwendete Anzahl Elemente ist in size gespeichert
- Eine ArrayList startet mit der Grösse 10 wenn nicht's anderes angegeben wurde
- Die Kapazität wird erhöht, wenn die ArrayList voll ist und ein neues Element hinzugefügt wird.
- Die neue Kapazität wir bei der Erhöhung wie folgt berechnet:
  JRE1.6: $cap_{new} = (cap_{old} * 3)/2 + 1$
  JRE1.7: $cap_{new} = cap_{old} + (cap_{old} >> 1)$
- void add(int index, Object element)
- void clear()

- boolean contains(Object o)
- Object get(int index)
- int indexOf(Object o)
- boolean isEmpty()
- ListIterator listIterator(int index)
- Object remove(int index)
- boolean remove(Object o)
- Object set(int index, Object element)
- int size()
- ...

7.4 Generics Wildcards

7.4.1 Unbounded Wildcard <?>

Einsatz
- Deklaration einer Referenzvariable, die auf Objekte beliebiger aktuell parametrisierter Klassen eines generischen Typs zeigen kann
- Arrays von generischen Klassen

Interpretation
- bei G<T> steht der formale Typ-Parameter T stellvertretend für genau einene Referenztyp
- bei G<?> steht das Fragezeichen für alle möglichen Referenztypen

Listing 16: Unbound Wildcard

```
Punkt<?>[] array = new Punkt<?>[5];

array[0] = new Punkt<Integer>(1,2);
array[1] = new Punkt<Double>(1.5, 2.5);
array[2] = new Punkt<String>("hoi", "du");

Object o = array[0].getX();

// Number n = array[0].getX();
// Integer i = array[0].getX();
// array[0].getX(5);
```

7.4.2 Upper Bound Wildcard <? extends T>

Einsatz Deklaration einer Referenzvariable, die auf Objekte eines generischen Typs zeigen kann, wobei die Objekte mit T oder einer Subklasse von T aktuell parametrisiert sein müssen

Interpretation bezeichnet die oberste Klasse in der Klassenhierarchie, welche als aktuellen Typ-Parameter für den formalen Typ-Parameter T eingesetzt werden darf

Listing 17: Upper Bound Wildcard

```
Punkt<? extends Number> ref;
ref = new Punkt<Integer>(1,2);
// ref = new Punkt<String>("hoi", "du");

Object o = ref.getX();
Number n = ref.getX();
// Integer i = ref.getX();
// ref.setX(5);
```

7.4.3 Low Bound Wildcard <? super T>

Einsatz Deklaration einer Referenzvariable, die auf Objekte eines generischen Typs zeigen kann, wobei die Objekte mti T oder einer Superklasse von T aktuell parametrisiert sein müssen

Interpretation bei G<? super C> steht das Fragezeichen für alle möglichen Superklassen von C (inkl. C selber)

Listing 18: Lower Bound Wildcard

```
Punkt<? super Integer> ref;
ref = new Punkt<Integer>(1,2);
// ref = new Punkt<Double>(1.5, 2.5);

Object o = ref.getX();
// Number n = ref.getX();
// Integer i = ref.getX();
ref.setX(5);
```

7.4.4 Interner Aufbau (bis Java 1.4.2)

- alle Klassen aus dem Java Collections Framework verwalten intern Referenzen vom Typ Object ⇒ maximale Flexibilität
- alle Methodenschnittstellen verwenden den Typ Object
- da alle Referenztypen zur Klasse Object zuweisungskompatibel sind, bieten die Collections keine Typsicherheit

7.4.5 Interner Aufbau (seit Java 5.0)

- alle Collections verwalten intern Referenzen vom Typ <E>, wobei E ein beliebiger Platzhalter (Typ-Parameter) für einen konkreten Referenztyp ist
- Collections werden generisch (maximale Flexibilität)
- Collections können für alle möglichen Referenztypen verwendet werden, auch für Object (= früherer interner Aufbau)

8 Programmverifikation

8.1 Korrektheit eines Algorithmus

- Ein Algorithmus heisst korrekt, wenn er seiner Spezifikation genügt
- überprüfung:
  - Verifikation: Mittels logischer Herleitung
  - Testen: Fehlerfreiheit kann nicht nachtgewiesen werden

8.2 Aussagenlogik

8.2.1 Syntax

- atomare Formeln (Atome): A, B, C, ...
- seien F und G zwei beliebige Formeln
- Konjuktion: $(F \wedge G)$ ist eine Formel
- Disjunktion: $(F \vee G)$ ist eine Formel
- Negation: $negF$ ist eine Formel
- Implikation: $(F \rightarrow G)$ ist eine Kurzschreibweise für $(\neg F \vee G)$
- Bikonditional: $(F \leftrightarrow G)$ ist eine Kurzschreibweise für $(F \rightarrow G) \wedge (G \rightarrow F)$

8.2.2 Semantik

- Belegung $\mathfrak{B}$: eine Teilmenge der atomaren Formeln wird mit einem Wahrheitswert aus false, true bzw. 0,1 belegt
- die Bedeutungen der Konjunktion, Disjunktion und Negation sind analog zur Bool'schen Algebra definiert
- passende Belegung $\mathfrak{B}$ auf Formel F angewendet: $\mathfrak{B}$(F)

8.2.3 Modell

- eine Belegung heisst zu einer Formel F **passend**, wenn alle in F vorkommenden Atome belegt sind
- eine Belegung $\mathfrak{B}$ ist ein **Modell** für eine Formel F, wenn sie passend ist und wenn der Wahrheitswert von $\mathfrak{B}$(F) = true ist: $\mathfrak{B} \models F$
- Belegung ist $\mathfrak{B}$ **kein Modell** für Formel F, wenn sie zwar passend ist, aber wenn der Wahrheitswert von $\mathfrak{B}$(F) = false ist: $\mathfrak{B} \not\models F$

8.2.4 Erfüllbarkeit

- Eine Formel F heisst **erfüllbar**, wenn für sie ein Modell existiert, andernfalls **unerfüllbar**
- Eine Formel F heisst **gültig** (oder Tautologie), wenn alle passenden Belegungen Modelle sind: $\models F$

8.2.5 Äquivalenz

Zwei Formeln F und G heissen (semantisch) äquivalent, falls für alle passenden Belegungen $\mathfrak{B}$ gilt: $\mathfrak{B}$(F) = $\mathfrak{B}$(G). Dafür schreiben wir $\models (F \leftrightarrow G)$ oder kurz $F \equiv G$.

AND		OR	
$(F \wedge F)$	$\equiv F$	$(F \vee F)$	$\equiv F$
$(F \wedge G)$	$\equiv (G \wedge F)$	$(F \vee G)$	$\equiv (G \vee F)$
$((F \wedge G) \wedge H)$	$\equiv (F \wedge (G \wedge H))$	$((F \vee G) \vee H)$	$\equiv (F \vee (G \vee H))$
$(F \wedge (F \vee G))$	$\equiv F$	$(F \vee (F \wedge G))$	$\equiv F$
$(F \wedge (G \vee H))$	$\equiv ((F \wedge G) \vee (F \wedge H))$	$(F \vee (G \wedge H))$	$\equiv ((F \vee G) \wedge (F \vee H))$
$\neg(F \wedge G)$	$\equiv (\neg F \vee \neg G)$	$\neg(F \vee G)$	$\equiv (\neg F \wedge \neg G)$
$(F \wedge G)$	$\equiv G$, falls $\models F$	$(F \vee G)$	$\equiv G$, falls $\models F$
$(F \wedge G)$	$\equiv$ F, falls F unerfüllbar	$(F \vee G)$	$\equiv$ F, falls F unerfüllbar

8.3 Prädikatenlogik

8.3.1 Syntax

- Erweiterung der Aussagenlogik um Quantoren, Funktionen, Prädikate, Relationen und Variablen
  - nullstellige Prädikate entsprechen den Atomen der Aussagelogik
  - Relationen können als Prädikate aufgefasst werden
  - nullstellige Funktionen sind Konstanten
- Formeln
  - logische Verknüpfung von Prädikaten:
    die Argumente der Prädikate sind Terme, gebildet aus Funktionen, Variablen und Konstanten
  - sei x eine Variable und F eine Formel
    - $\forall$ x F ist eine Formel mit gebundenem x
    - $\exists$ x F ist eine Formel mit gebundenem x
  - eine Formel heisst geschlossen oder Aussage, wenn alle ihre Variablen durch Quantoren gebunden sind
- Prädikatenlogik mit Identität
  wenn F und G zwei Formeln sind, so ist auch F = G eine Formel

8.3.2 Semantik

- Struktur $S = (U_s, I_s)$
  U_s ist eine nicht-leere Grundmenge und I_s eine Abbildung
  - der Prädikate über U_s
  - der Funktionen auf U_s
  - der Variablen auf Elemente von U_s
- Semantik bezüglich passender Struktur S

S(Term) jeder Term wird gemäss I_s auf ein Element aus U_s abgebildet

S(Prädikat) jedes Prädikat bestimmt einen Wahrheitswert in Abhängigkeit seiner Argumente (Terme)

S(Formel) die aussagenlogische Verknüpfung der Wahrheitswerte der Prädikate

S($\forall$x F) F ist eine Formel
- ist true, falls für alle d $\in U_s$ gilt: $S_{[x/d]}(F) = true$
- ist false, sonst

S($\exists$x F) F ist eine Formel
- ist true, falls für alle d $\in U_s$ gilt: $S_{[x/d]}(F) = true$
- ist false, sonst

8.3.3 Äquivalenz

Definition

Zwei prädikatenlogische Formeln F und G sind äquivalent, falls für alle passenden Strukturen S gilt: S(F) = S(G). Dafür schreiben wir F ≡ G.

Äquivalenzregeln (zusätzlich zur Aussagenlogik)

$$\begin{aligned} \neg\forall x F &\equiv \exists x \neg F \\ \forall x \forall y F &\equiv \forall y \forall x F \\ (\forall x F \wedge \forall x G) &\equiv \forall x (F \wedge G) \\ \neg\exists x F &\equiv \forall x \neg F \\ \exists x \exists y F &\equiv \exists y \exists x F \\ (\exists x F \vee \exists x G) &\equiv \exists x (F \vee G) \end{aligned}$$

falls x in G nicht frei vorkommt, gilt:

$$\begin{aligned} (\forall x F \wedge G) &\equiv \forall x (F \wedge G) \\ (\forall x F \vee G) &\equiv \forall x (F \vee G) \\ (\exists x F \wedge G) &\equiv \exists x (F \wedge G) \\ (\exists x F \vee G) &\equiv \exists x (F \vee G) \end{aligned}$$

8.3.4 Beispiele

Alle Studierenden von Professor p mögen Logik.

L(x) : "x mag Logik"
S(x,y) : "x studiert bei y"
$\forall x : S(x, p) \rightarrow L(x)$
$\forall x : \neg S(x, p) \vee L(x)$

Ein Professor ist glücklich, wenn alle seine Studierenden Logik mögen

G(x) : "x ist glücklich"
$\exists y : (\forall x : S(x, y) \rightarrow L(x)) \rightarrow G(y)$

Ein Professor ist unglücklich, wenn er keine Studierende hat.

$\exists y : (\forall x : \neg S(x, y)) \rightarrow \neg G(y))$

8.4 Verifikation

8.4.1 Hoare-Tribble

Ein Hoare-Triple ($\{R\}P\{S\}$) ist gültig, wenn bei erfüllter Vorbedingung R und nach Ausführung von P die Spezifikation (Nachbedingung) S erfüllt ist. Das Programm P ist korrekt, wenn das Hoare-Triple gültig ist.

Variablen

Input: $a, b, c, ..., z$
Output: $a', b', c', ..., z'$

Vorbedingung (Precondition)

Damit das Programm P die Spezifikation (Postcondition) S erfüllen kann, muss die Vorbedingung R erfüllt sein. Je schwächer die Vorbedingung R, desto besser: **weakest precondition (WP)**.

Beispiele

Spezifikation: $S = (x' = 10)$
Programm: $P = (x' := x + 2)$
mögliche Vorbedingung $R = (a = 0 \vee b = 2 \vee x = 8 \vee z = 0)$
einfachste Vorbedingung (weakest Precondition); $WP = (x = 8)$

Nachbedingungen (Postcondition)

Die Nachbedingungen entsprechen der **Spezifikation** des Programms und sind je präziser, umso besser.

Implikation: $[\ \underbrace{S}_{speziellesSpez.} \rightarrow \underbrace{T}_{allg.Spez.}\ ] \equiv \forall a, b, c, ..., z, a', b', c', ..., z' : \neg S \vee T$

- $a, b, c, ..., z$ sind alle Variablen, die in den Spezifikationen S und T vorkommen
- wenn ein Programm P die speziellere Spezifikation S erfüllt, so auch die allgemeinere T
- es gibt ein Programm Q, welches die Spezifikation T erfüllt und mindestens so einfach ist wie das Programm P, welches die Spezifikation S erfüllt

Besipiele:

$S = (x' = 10), T = (x' > 0)$

$P_1 = (x' := 6), P_2 = (x' := 10), P_3 = (x' := x + 1)$

- $\underbrace{\{true\}}_{algemeinsteVorbedingung} \quad P_1\{S\} = false$
- $\{true\}P_1\{T\} = true$
- $\{true\}P_2\{S\} = true$
- $\{true\}P_2\{T\} = true$
- $\{true\}P_3\{S\} = false$
- $\{x = 9\}P_3\{S\} = true$

8.4.2 Ablauf der Verifikation

Gegeben

Vorbedingung R, Spezifikation S, Programm P

Ablauf

- aus S und P die schwächste Vorbedingung $WP = wp(P, S)$ berechnen
- falls $[R \rightarrow WP]$ gilt, dann erfüllt das Programm P die Spezifikation S bei eingehaltener Vorbedingung R
- R kann somit durch WP ersetzt werden

Berechnung der WP

- ausgehend von der Spezifikation S
- Anweisung für Anweisung rückwärts rechnen
- Berechnungsregeln für Zuweisung, Sequenz, Selektion, Iteration

8.5 Weakest Precondition: Rechenregeln

8.5.1 Zuweisung

Gegeben

Zuweisung $P = (x' := y)$

Nachbedingung S

Regel

$WP = wp(P, S) = wp((x' := y), S) = S|_{alle\ \ x'\ \ durch\ \ y\ \ ersetzt}$

Beispiele

$S = (a' = z), P = (a' := b + c), WP = (b + c = z)$

$S = (n' = n), P = (n' := n - 1), WP = (n = n - 1) = false$

8.5.2 Weakest Precondition herausfinden (Beispiel)

$S = (r' = x \cdot y)$, $P = (\text{if } x = 2 \cdot k \text{ then } r' := 2 \cdot k \cdot y \text{ else } s' := 2 \cdot k \cdot y; r' := s' + y)$, $B = (x = 2 \cdot k)$

$$
\begin{aligned}
S1 &= wp((r' := 2ky), S) = (2ky = xy) = (2k = x) = B \\
S2 &= wp((s' := 2ky; r' := s' + y), S) = wp(s' := 2ky, s' + y = xy) = (2ky + y = xy) = (2k + 1 = x) \\
R &= wp(P, S) = (B \to S1 \wedge \neg B \to S2) \\
&= (\neg B \vee S1) \wedge (B \vee S2) = \neg B \wedge B \vee \neg B \wedge S2 \vee S1 \wedge B \vee S1 \wedge S2 \\
&= \neg B \wedge S2 \vee S1 \wedge B \vee S1 \wedge S2 \\
&= B \wedge S1 \vee \neg B \wedge S2 \\
&= (x = 2k \wedge x = 2k) \vee (x \neq 2k \wedge 2k + 1 = x) \\
&= (x = 2k) \vee (2k + 1 \neq 2k \wedge 2k + 1 = x) \\
&= (x = 2k) \vee (true \wedge 2k + 1 = x) \\
&= (x = 2k) \vee (x = 2k + 1)
\end{aligned}
$$

8.5.3 Sequenz

Gegeben

Sequenz P = $(Anweisung_1; Anweisung_2; ...; Anweisung_n)$
Nachbedingung S

Regel

$WP = wp(P, S) = wp((Anweisung_1; Anweisung_2; ...; Anweisung_{n-1}), wp(Anweisung_n, S))$

Beispiel

$P = (t' := x; x' := y; y' := t')$
$S = (x' = b \wedge y' = a)$

$$
\begin{aligned}
WP &= wp((t' := x; x' := y; y' := t'), (x' = b \wedge y' = a)) \\
&= wp((t' := x; x' := y), wp((y' := t'), (x' = b \wedge y' = a))) \\
&= wp((t' := x; x' := y), (x' = b \wedge t' = a)) \\
&= wp((t' := x), wp((x' := y), (x' = b \wedge t' = a))) \\
&= wp((t' := x), (y = b \wedge t' = a)) \\
&= (y = b \wedge x = a)
\end{aligned}
$$

8.5.4 Selektion

Gegeben

logischer Ausdruck b ; Selektion P = (if b then P_1 else P_2) ; Nachbedingung S

Regel

$$
\begin{aligned}
WP &= wp(P, S) = wp((\text{ if b then } P_1 \text{ else } P_2), S) \\
&= (\neg b \vee wp(P_1, S)) \wedge (b \vee wp(P_2, S)) \\
&= b \wedge wp(P_1, S) \vee \neg b \wedge wp(P_2, S)
\end{aligned}
$$

Beispiel

$P = (if\ x < 0\ then\ x' := -x\ else\ x' := x - 1)$
$S = (x' > 0)$
$WP = wp((if\ x < 0\ then\ x' := -x\ else\ x' := x - 1), x' > 0)$
$= (x \geq 0 \vee -x > 0) \wedge (x < 0 \vee x - 1 > 0)$
$= (x \geq 0 \vee x < 0) \wedge (x < 0 \vee x > 1)$
$= true \wedge (x \notin [0, 1])$
$= x \notin [0, 1]$
$= (x < 0 \wedge -x > 0) \vee (x \geq 0 \wedge x - 1 > 0)$
$= (x < 0) \vee (x \geq 0 \wedge x > 1)$
$= (x < 0) \vee (x > 1)$
$= x \notin [0, 1]$

8.5.5 Iteration

Iterationen
for-Schleife: Anzahl Durchläufe bekannt → als Sequenz betrachten
while-Schleife: Spezialfall der Rekursion → Invariante bestimmen

Invariante (INV)
eine logische Bedingung ausgedrückt in den Variabeln aus P, welche vor P und nach jedem durchlauf von P_1 gültig ist
wenn $\{INV \wedge B\}P_1\{INV\}$ gültig ist, so ist auch $\{INV\}$ while B do $P_1\{INV \wedge \neg B\}$ gültig

Regel
WP = wp(P,S) = INV

8.6 Invarianten-Anwendung

Listing 19: Multiply

```
long multiply(int x, int y) {
  // 4. schwaechste Vorbedingung WP
  long a = x, b = y, r = 0;
  // 3. INV muss gelten
  while (a != 0) {
    // 2. INV & (a!=0) muss gelten
    if ((a & 1) == 1) r += b;
    b = b << 1;
    a = a >> 1;
    // 1. INV muss gelten
  } // 5. INV & (a = 0) muss gelten
  return r;
} //Nachbedingung S muss gelten
```

8.6.1 Ist INV wirklich eine korrekte Invariante?

Gegeben
Invariante INV = $(a \geq 0 \wedge a \cdot b + r = x \cdot y)$

Beweisverfahren
wenn an Position 1 die Invariante INV gilt, dann muss an der Position 2 auch die Invariante gelten
zu zeigen: $\{(INV \wedge B) \rightarrow wp(P1, INV)\}$

Beispiel
R1 = wp(a := a >> 1, INV) = $(\frac{a}{2} \geq 0 \wedge \frac{a}{2} \cdot b + r = x \cdot y)$
R2 = wp(b := b << 1, R1) = $(\frac{a}{2} \geq 0 \wedge \frac{a}{2} \cdot b \cdot 2 + r = x \cdot y)$
R3 = wp((if (a&1) = 1 then r := r + b else NOP), R2) = $\frac{a}{2} \geq 0 \wedge a \cdot b + r = x \cdot y$

8.6.2 Was ist die schwächste Vorbedingung WP?

Gegeben
Invariante INV = $(a \geq 0 \wedge a \cdot b + r = x \cdot y)$

Beweisverfahren
Wenn an Position 2 die Invariante INV gilt, dann gilt sie auch an der Position 3.
Somit ist INV an der Position 3 die Nachbedingung der Instruktionen vor der while-Schleife und wir können mit dem üblichen Verfahren WP berechnen.

Beispiel
R1 = wp(r := 0, INV) = $(a \geq 0 \wedge a \cdot b = x \cdot y)$
R2 = wp(b := y, R1) = $(a \geq 0 \wedge a \cdot y = x \cdot y)$
WP = wp(a := x, R3) = $(x \geq 0 \wedge x \cdot y = x \cdot y) = (x \geq 0)$

8.6.3 Welches ist die Nachbedingung S?

Gegeben

Invariante INV = $(a \geq 0 \wedge a \cdot b + r = x \cdot y)$
Schleifenbedingung B = $(a \neq 0)$

Verfahren

Wenn an Position 1 die Invariante INV gilt, dann gilt an der Position 5 die Invariante, nicht aber B, also S = $(INV \wedge \neg B)$

Beispiel

$S = (INV \wedge \neg B) = (a \geq 0 \wedge a \cdot b + r = x \cdot y \wedge a = 0)$
$S = (r = x \cdot y \wedge a = 0)$
Der Rückgabewert r ist also gleich dem Produkt aus x und y.

9 Komplexität

9.1 Zeit- und Speicherbedarf

Modell: Random-Access-Machine (RAM)
Eine möglichst einfache CPU mit einem unendlich grossen Speicher bestehend aus Zellen für beliebig grosse Zahlen.

Speicherbedarf eines Algorithmus
Die minimale Anzahl Speicherzellen im RAM-Modell, die vorhanden sein müssen, damit der Algorithmus korrekt ausgeführt werden kann.

Zeitbedarf eines Algorithmus
Laufzeit in Echtzeit: gemessene Ausführungszeit eines Programms mit vordefinierten Testdaten auf einem präzise beschriebenen Computersystem.
Laufzeit in RAM-Instruktionen: die minimale Anzahl RAM-Befehle, die ausgeführt werden müssen, um den Algorithmus abzuarbeiten.

Definitionen
Laufzeit: T(n)
Speicherbedarf: M(n)

9.2 Worst-Case vs. Average-Case

Best-Case (meistens uninteressant)
die Zahlen sind bereits sortiert

Worst-Case (sinnvoll und praktikabel)
die Zahlen sind in umgekehrter Reihenfolge

Average-Case (oft zu schwierig)
Um eine Aussage der Art ¨Im Durchschnitt dauert es¨ machen zu können, muss Klarheit bestehen, worüber der Durchschnitt gebildet werden darf. Fairerweise müssen alle möglichen Probleminstanzen der Länge n mit ihrer Wahrscheinlichkeitsverteilung in Betracht gezogen werden.

9.3 Gross-O und Gross-Omega

9.3.1 Gross-O

- obere Schranke einer Worst-Case Abschätzung
- sinnvoll für konkrete Algorithmen
- Beispiel:
  n Zahlen können mit Heap-Sort in Zeit T(n) $\in \mathcal{O}$(n log n) bei einem Speicherbedarf M(n) $\in \mathcal{O}$(n) sortiert werden

9.3.2 Gross-Omega

- untere Schranke einer Worst-Case Abschätzung
- sinnvoll für Problemklassen
- Beispiel:
  um n beliebige Zahlen in beliebiger Reihenfolge auf einer RAM zu sortieren, braucht es mindestens Ω(n log n) Rechenschritte
  Formal: $\Omega(g(n)) = \{h(n) | \exists c > 0 : \exists \text{ unendlich viele } n : h(n) \geq c \cdot g(n)\}$
- Gross-Theta: Sind $\mathcal{O}$ und Ω gleich wird geschrieben: $T(f) \in \Theta(g)$
  Kann gelesen werden als: „f wächst genauso schnell wie g“

9.4 Reihen

9.4.1 Wichtige Reihen

Name	Reihe	Wert
geometrische Partialsumme	$\sum_{i=0}^{n} q^i$	$\frac{1-q^{n+1}}{1-q}$
arithmetische Reihe	$\sum_{k=1}^{n} k$	$\frac{n(n+1)}{2}$
	$\sum_{k=1}^{n} k^2$	$\frac{n(n+1)(2n+1)}{6}$
Exponentialfunktion	$\sum_{n=0}^{\infty} \frac{x^n}{n!}$	e^x

9.4.2 Rechenregeln

- $\sum_{k=0}^{n} k = 0 + 1 + 2 + ... + n = \sum_{k=1}^{n} k = \frac{n(n+1)}{2}$
- $\sum_{i=0}^{\infty}(a_i + b_i) = \sum_{i=0}^{\infty} a_i + \sum_{i=0}^{\infty} b_i$
- $\sum_{i=0}^{\infty}(a_i - b_i) = \sum_{i=0}^{\infty} a_i - \sum_{i=0}^{\infty} b_i$
- $\sum_{k=1}^{n} n = \sum_{k=0}^{n-1} n = n \cdot n = n^2$
- $\sum_{k=0}^{n-1} k = \frac{(n-1)(n-1+1)}{2} = \frac{n(n-1)}{2}$
- $\sum_{k=0}^{n-1} n - k = \sum_{k=1}^{n} n - \sum_{k=0}^{n-1} k = n^2 - \frac{n(n-1)}{2}$
- $\sum_{i=0}^{\infty} A \cdot a_i = A \cdot \sum_{i=0}^{\infty} a_i$
- $\sum_{i,j=0}^{\infty}(a_i \cdot b_j) = (\sum_{i=0}^{\infty} a_i) \cdot (\sum_{i=0}^{\infty} b_i)$

10 Rekursion

10.1 Rekursionsschema

Rekursive Methoden müssen eine Fallunterscheidung enthalten

- zuerst einfachen Fall ohne Rekursion abhandeln
- dann der aufwendigere Fall mit dem rekursiven Aufruf
- Sicherstellen, dass der aufwändigere Fall durch Rekursion irgendwann im einfachen Fall endet.

Listing 20: Rekursionsschema

```
void rek(...) {
    if (einfacher Fall) {
        sofort erledigen
    } else {
        rekursiver Aufruf
    }
}
```

Listing 21: Rekursionsschema Beispiel - Fakultät

```
int rek(int x) {
    if (x <= 1) {
        return x;
    } else {
        return x * rek(x-1);
    }
}
```

10.2 Backtracking

- eigenet sich für Probleme, bei denen alle Kandidaten einer Lösungsmenge (der ganze Suchraum) inspiziert werden müssen, um zu eintscheiden, ob ein Kandidat eine Lösung darstellt
- erschöpfende Suche ist meistens mit exponentiellem Aufwand verbunden → nur verwenden, wenn nichts Besseres vorhanden
- typische Beispiele
  - Labyrinthsuche
  - Erfüllbarkeit von logischen Aussagen in konjunktiver Normalform
  - n-Damen-Problem
  - Springerweg, Springerkreis
  - ...

10.2.1 3 Voraussetzungen für Backtracking

1. Lösung des Problems läst sich als Vektor V (unbestimmter, aber endlicher Länge) darstellen
2. Alle Lösungskandidaten V_i bilden zusammen einen endlich grossen Lösungsraum
3. Es existiert ein effizienter Test zur Erkennung von nicht-erweiterbaren Teillösungen → $(v_1, v_2, \ldots, v_k)$ lässt sich nicht zu $(v_1, v_2, \ldots, v_k, v_{k+1})$ erweitern

10.2.2 8-Damen-Problem

- Ziel: 8 Damen auf einem Schachbrett platzieren, so dass keine zwei Damen sich gegenseitig bedrohen
- Verallgmeinerung: n Damen auf einem Schachbrett der Grösse $n \times n$ platzieren, so dass ...
- Zustandsraum
  - alle Möglichkeiten, um 4 Damen auf ein leeres Schachbrett der Grösse 4×4 zu platzieren
  - Anzahl Zustände: $16 \cdot 15 \cdot 14 \cdot \cdots = 43680 > 13^4 = (4^2 - 4 + 1)^4$
  - bei n Damen gibt es also mehr als $(n^2 - n + 1)^n \in O(n^{2n})$ Zustände → exponentiell grosser Suchraum
- Ansatz: erschöpfende Suche mittels Backtracking
  - durch Vorwissen kann der Suchraum reduziert werden
  - Einsatz von Heuristiken (vielversprechende Kandidaten zuerst) können die Laufzeit stark reduzieren

10.3 Code

Listing 22: 8-Damen-Problem

```
public class Queens {

   /*************************************************************************
    * Return true if queen placement q[n] does not conflict with
    * other queens q[0] through q[n-1]
    *************************************************************************/
    public static boolean isConsistent(int[] q, int n) {
        for (int i = 0; i < n; i++) {
            if (q[i] == q[n])             return false;   // same column
            if ((q[i] - q[n]) == (n - i)) return false;   // same major diagonal
            if ((q[n] - q[i]) == (n - i)) return false;   // same minor diagonal
        }
        return true;
    }

   /*************************************************************************
    * Print out N-by-N placement of queens from permutation q in ASCII.
    *************************************************************************/
    public static void printQueens(int[] q) {
        int N = q.length;
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++) {
                if (q[i] == j) System.out.print("Q ");
                else           System.out.print("* ");
            }
            System.out.println();
        }
        System.out.println();
    }


   /*************************************************************************
    *  Try all permutations using backtracking
    *************************************************************************/
    public static void enumerate(int N) {
        int[] a = new int[N];
        enumerate(a, 0);
    }

    public static void enumerate(int[] q, int n) {
        int N = q.length;
        if (n == N) printQueens(q);
        else {
            for (int i = 0; i < N; i++) {
                q[n] = i;
                if (isConsistent(q, n)) enumerate(q, n+1);
            }
        }
    }


    public static void main(String[] args) {
        int N = Integer.parseInt(args[0]);
        enumerate(N);
    }

}
```

11 Divide-and-Conquer

Divide	1.)	Gesamtproblem in 2 oder 3 gleichgrosse Teilprobleme zerlegen
Conquer	2.)	jedes Teilproblem separat lösen (gleicher Algorithmus wie fürs Gesamtproblem)
Merge	3.)	aus Teillösungen die Gesamtlösung erzeugen.

11.1 Code

Listing 23: max Teilsumme

```
int maxTSumme(int[] a, int left, int right) {
  if (left > right) return 0;
  if (left == right) return (a[left] >= 0 ? A[left] : 0);
  int m = (left + right) >>> 1;
  int v1 = maxTSumme(a, left, m)
  int v2 = maxTSumme(a, m+1, right);
  int sum=0; rmax=0; lmax=0;
  for (int i=m+1; i <= right; i++) {
    sum += a[i];
    if (sum > rmax) rmax =  sum;
  }
  sum = 0;
  for (int i=m; i>=left; i--) {
    sum += a[i];
    if (sum > lmax) lmax = sum;
  }
  int v3 = lmax + rmax;
  return max(v1, v2, v3);
}
```

11.2 Analyse

11.2.1 Schritte

- Zeile 2-4: 10 Schritte
- Zeile 5: $T(\frac{n}{2})$ Schritte
- Zeile 6: $T(\frac{n}{2})$ Schritte
- Zeile 8 - 16: $6 + 6 \cdot n$ Schritte
- Zeile 17 - 18: 5 Schritte

n: länge von a

11.2.2 Aufwandsformel

$T(n) = 2 \cdot T(\frac{n}{2}) + 6n + 21$
$T(1) = 7$

11.3 Berechnung

11.3.1 Teleskopieren

$T(n) = 2 \cdot T(\frac{n}{2}) + 6 \cdot n + 21$
$T(n) = 2 \cdot (2 \cdot T(\frac{n}{4}) + \frac{6 \cdot n}{2} + 21) + 6 \cdot n + 21$
$T(n) = 4 \cdot T(\frac{n}{4}) + 12 \cdot n + 3 \cdot 21$
$T(n) = 4 \cdot (2 \cdot T(\frac{n}{8}) + \frac{6 \cdot n}{4} + 21) + 12 \cdot n + 3 \cdot 21$
$T(n) = 8 \cdot T(\frac{n}{8}) + 18 \cdot n + 7 \cdot 21$
...

11.3.2 Verallgemeinerung

$T(n) = 2^i \cdot T(\frac{n}{2^i}) + 6 \cdot i \cdot n + (2^i - 1) \cdot 21$

$\boxed{\frac{n}{2^i} = 1 \Rightarrow n = 2^i \Rightarrow i = log_2(n) = ld(n)}$

$T(n) = 2^{ld(n)} \cdot T(1) + 6 \cdot ld(n) \cdot n + (2^{ld(n)} - 1) \cdot 21$
$T(n) = n \cdot 7 + 6 \cdot ld(n) \cdot n + (n - 1) \cdot 21$
$T(n) = 6 \cdot n \cdot ld(n) + 28 \cdot n - 21 \Rightarrow O(n \cdot \log_2(n))$

11.4 Beispiel Teleskopieren

$$
\begin{aligned}
T(n) &= 2T(\frac{n}{2}) + \sqrt{n} \\
&= 2(2T(\frac{n}{4}) + \sqrt{\frac{n}{2}}) + \sqrt{n} \\
&= 4T(\frac{n}{4} + 2\sqrt{\frac{n}{2}} + \sqrt{n} \\
&= 4(2T(\frac{n}{8}) + \sqrt{\frac{n}{4}}) + 2\sqrt{\frac{n}{2}} + \sqrt{n} \\
&= 8T(\frac{n}{8}) + 4\sqrt{\frac{n}{4}} + 2\sqrt{\frac{n}{2}} + \sqrt{n} \\
&\dots \\
&= 2^i T(\frac{n}{2^i}) + \sum_{k=0}^{i-1} 2^k \sqrt{\frac{n}{2^k}} = 2^i T(\frac{n}{2^i}) + \sum_{k=0}^{i-1} \sqrt{2^k}\sqrt{n} \\
&= 2^i T(\frac{n}{2^i}) + \sqrt{n} \sum_{k=0}^{i-1} 2^{\frac{k}{2}} \\
&\text{mit } T(1) = 1 \text{ und } n = 2^i \\
&= n \cdot T(1) + \sqrt{n} \sum_{k=0}^{i-1} 2^{\frac{k}{2}} = n + \sqrt{n} \sum_{k=0}^{i-1} \sqrt{2}^k \\
&= n + \sqrt{n} \frac{\sqrt{2}^i - 1}{\sqrt{2} - 1} = n + \sqrt{n} \frac{\sqrt{n} - 1}{\sqrt{2} - 1} \\
&= n + \frac{n - \sqrt{n}}{\sqrt{2} - 1} \\
T(n) &\in O(n)
\end{aligned}
$$