Datentypen

Bite

2er Potenzen

2^{10}	2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
1024	512	256	128	64	32	16	8	4	2	1

2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}	2^{-8}
1	0.5	0.25	0.125	0.0625	0.03125	0.015625	0.0078125	0.00390625

Bitshift

\>> Right Shift signed (mit 1en auffüllen)
\>>> Right Shift unsigned (mit 0en auffüllen)
<< Left Shift unsigned (mit 0en auffüllen)

Float

Floating Point nach Dezimalzahl

V	Exponent 8 Bit								Mantisse							
1	1	0	0	0	0	1	0	1	0	0	1	0	0	1	0	...

V = **-1**
Exponent = 128 + 4 + 1 = 133 => 133 - 127 (Bias) = **6**
1.0010010 => Komma um 6 Stellen nach Rechts verschieben
1001001.0 = **73**

Dezimalzahl nach Floating Point

-23.5

1. Dezimal zu Binär: 00010111.1
2. Normalisieren: 00010111.1 => 0001.011110...
3. Exponent: 4 + 127 (Bias) = 131 => 1000 0011

Oktal zu Binär

$2\,2\,7\,6\,1_8 = 10\;010\;111\;110\;001_2$

Zeichen

Hex zu Binär

Hex	Bin	Hex	Bin
0	0000	8	1000
1	0001	9	1001
2	0010	A	1010
3	0011	B	1011

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 → 16 bit (BMP)	10yyyyyy	10xxxxxx	
000uuuzz zzzzyyyy yyxxxxxx	11110uuu → 4 byte → 21 bit (alle)	10zzzzzz	10yyyyyy	10xxxxxx

Suchen

Sequenzielle Suche

```
int search(int[] arr, int x) {
    for(int i = 0; i < arr.length; i++)
        if(arr[i] == x)
            return i;
    return -1;
}
```

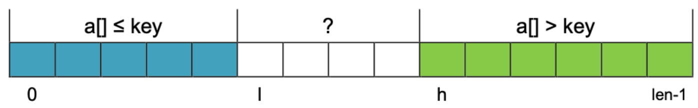
Sequenzielle Suche mit Wächter

```
boolean search(int[] data, int value){
    int last = arr[n - 1];
    arr[n - 1] = x;
    int i = 0;
    while (arr[i] != x)
        i++;
    arr[n - 1] = last;
    return (i < n - 1) || (x == arr[n - 1]);
}
```

Binäre Suche

```
int binSearch(int[] data, int value){
  int l = -1, h = data.length + 1;
  while(l+1 != h){
    int m = (l + h) \ 2;
    if(data[m] < value)
      l = m;
    else if(data[m] > value)
      h = m;
    else
      return m;
  }
  return -1;
}
```

Binäre Suche Speziell




```
int binSearch1(int[] data, int key) {
  int l = 0, h = data.length;
  while (l != h) {
    int m = (l + h) / 2;
    if(data[m]<=key)l=m+1;
    else h=m;
  }
  return l - 1;
}
```

Binäre Suche für Objekte

```
<T extends Comparable<? super T>> int binSearch(T[] data,
T value) {
        int l = -1, h = data.length;
        while (l + 1 != h) {
                int m = (l + h) >>> 1;
                int c = data[m].compareTo(value);
                if (c < 0) l = m;
                else if (c > 0) h = m;
                else return m;
        }
        return NOT_FOUND;
}
```

Max Sub Array

```
int maxSub(int[] data) {
  int max = 0, cur = 0;
  for (int end = 0; end < data.length; end++) {
    cur = cur > 0 ? cur + data[end] : data[end];
      if (cur > max) max = cur;
  }
  return max;
}
```

Asymptotische Komplexität

Komplexitätsklassen

Klasse	Bezeichnung	$\frac{f(2n)}{f(n)} \approx$	Zuwachs
$O(1)$	konstant	1	kein Zuwachs
$O(\log n)$	logarithmisch	$1 + \frac{\log 2}{\log n}$	Zuwachs nimmt mit grösseren *n* ab
$O(n)$	linear	2	Zuwachs mit konstantem Faktor (unabhängig von *n*)
$O(n \log n)$		$2 + \frac{2 \cdot \log 2}{\log n}$	
$O(n^2)$	quadratisch	4	
$O(n^3)$	kubisch	8	
$O(2^n)$	exponentiell	2^n	Zuwachs nimmt mit grösserem *n* zu
$O(3^n)$		3^n	
$O(n!)$	Fakultät	$\prod_{i=n+1}^{2n} i$	

$O(\log n) < O(\sqrt{n}) < O(n)$
Binäre Suche: $O(\log n)$
Sequenzielle Suche: $O(n)$

Sortialgorithmen

Bubble Sort

Sortiert von Hinten nach Vorne. Immer zwei Elemente werden verglichen. Wenn Element i-1 kleiner i dann wird getauscht. Vorne ist immer sortiert. $O(n^2)$

```
void BubbleSort(int[] a) {
  int n = a.length;
  for (int i = 0; i < n - 1; i++) {
    for (int j = n - 1; j > i; j--) {
      if (a[j] < a[j - 1]) {
        int t = a[j - 1]; a[j - 1] = a[j]; a[j] = t;
      }
    }
  }
}
```

Insertion Sort

Vorne nach Hinten. Vorne ist aufsteigend sortiert. In jeder Iteration wird ein Element mehr genommen und geschaut wo es hingehört. $O(n^2)$ Insertion Sort mit binary Search hat $O(n * \log n)$

```
void InsertionSort1(int[] a) {
  int n = a.length;
  for (int i = 1; i < n; i++) {
    for (int j = i; j > 0 && a[j - 1] > a[j]; j--) {
      int t = a[j - 1]; a[j - 1] = a[j]; a[j] = t;
    }
  }
}
```

Selection Sort

Vorne nach Hinten. Vorne ist aufsteigend sortiert. Es wird immer das kleinste Element im Array gesucht. $O(n^2)$

```
void SelectionSort(int[] a) {
  int n = a.length;
  for (int i = 0; i < n - 1; i++) {
    int min = i;
    for (int j = i + 1; j < n; j++) {
      if (a[j] <= a[min]) min = j;
    }
    int t = a[min]; a[min] = a[i]; a[i] = t;
  }
}
```

Selection Sort mit IndexOf

```
void sort(int[] data) {
  int n = data.length;
  for (int i = 0; i < n - 1; i++) {
    int min = indexOfMin(data, i, n);
    int t = data[min];
    data[min] = data[i];
    data[i] = t;
  }
}
```

Halbdynamische Datenstrukturen

Problem bei Arrays. Fixe Grösse und muss bei Initialisierung angegeben werden. Halbdynamische Strukturen passen zur Laufzeit den Speicherbedarf konstant an. Wichtige Fragen: Initiale Kapazität? In welchen Schritten muss erhöht/vermindert werden? Wann soll erhöht/vermindert werden? Zu oft kopieren ist teuer, da zeitintensiv. Nicht unnötig viel Speicher im Voraus reservieren.

Beispiel Code

UTF8Converter

```
int count(byte[] text) {
    int c = 0;
    for (int i = 0; i < text.length; i++) {
        int leading = numLeadingOnes(text[i]);
        if (leading == 3) {
            i = i + 2; c++;
        } else if (leading == 2) {
            i++; c++;
        } else
            c++;
    }
    return c;
}
```

Leading 1en

```
int numLeadingOnes(byte b) {
  if ((b & (byte) 0b1110_0000) == (byte) 0b1110_0000) return 3;
  if ((b & (byte) 0b1100_0000) == (byte) 0b1100_0000) return 2;
  return 1;
}
```

CharToDez

```
int convertToInt(char[] data) {
    int fac = 1, res = 0;
    for (int i = data.length - 1; i >= 0; i--) {
        res = res + fac * ((int) data[i] - 48);
        fac *= 10;
    }
    return res;
}
```

Prime

```
boolean isPrime(int x) {
  int t = (x - 1);
  while (t > 1) {
    if (x % t == 0) return false;
    t--;
  }
  return true;
}
```

NoOfUTF32

```
public int nofUTF32Bytes (byte[] utf8) {
  int count = 0;
  for (byte b : utf8) {
    if ((b & 0b11000000) != 128) count++;
  }
  return count * 4;
}
```

NoOfAscii

```
int nofAsciiChars(byte[] utf8) {
    int count = 0;
    for (byte b : utf8)
        if ((b >> 7) == 0b00000000)
            count++;
    return count;
}
```

Remove Duplicates

```
int removeDupl(int[] data, int size) {
    int i = 0, j = 1;
    while (j < size) {
      if (data[i] != data[j]) data[++i] = data[j++];
      else j++;
    }
    return size > 0 ? i + 1 : 0;
}
```

Suchen in Texten

Naive Textsuche

$\mathcal{O}(n * m)$

Anzahl Vergleiche: Textlänge - Musterlänge + 1 (+ Treffer)

Boyer-Moore

1. Suche jeweils das hinterste Zeichen im Text, welches nicht mit dem Muster übereinstimmt
2. Falls es kein solches gibt (j $<$ 0), wurde das Muster gefunden
3. Ansonsten (j $\geq$ 0) wird die Position von i anhand des Shift-Arrays verschoben

Vorkommens-Heuristik

- Für alle Zeichen, die im Muster nicht enthalten sind -> Länge des Musters
- Zeichen nur an letzter Position im Muster -> Länge des Musters
- Sonst Muster-Länge - 1 - lastIndex(Muster, Zeichen)

Asymptotische Komplexität

n = Text-Länge, m = Muster-Länge

Worst-Case: $\mathcal{O}(n * m)$

Erwarteter Average-Case: $\mathcal{O}(n/m)$

Best-Case: $\mathcal{O}(m)$ (Muster an erster Stelle)

	0	1	2	3	4	5
Pattern	l	e	e	r	e	r

	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
Shift:	6	6	6	6	1	6	6	6	6	6	6	5	6	6	6	6	6	2	6	6	6	6	6	6	6	6

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36
M	a	n	c	h		k	ü	h	l	e	r		L	e	h	r	e	r		w	i	r	d		i	m	m	e	r		l	e	e	r	e	r
l	e	e	r	e	r																															
						l	e	e	r	e	r																									
								l	e	e	r	e	r																							
														l	e	e	r	e	r																	
																				l	e	e	r	e	r											
																										l	e	e	r	e	r					
																															l	e	e	r	e	r

```
public int firstMatch(String text, String pattern) {

    int[] shift = allShifts(pattern);

    int l = pattern.length(), i = 0, j = l - 1; //
    ↪ Warum?
    while (i + l <= text.length() && j >= 0) {

        j = l - 1; // Warum?

        while (j >= 0 && pattern.charAt(j) ==
        ↪ text.charAt(i + j)) {
            j--;
        }

        if (j >= 0) {
            i = i + shift[text.charAt(i + l - 1)];
        }
    }
    return (i + l <= text.length()) ? i : -1;
}
```

Rekursion

DrawCricles

```
private void drawCircles(Graphics g, int left, int
↪ top, int size) {
    if (size >= 8) {
        g.drawOval(left, top, size, size);
        int s = size / 2;
        drawCircles(g, left, top + s/2, s); //  links
        drawCircles(g, left + s, top + s/2, s); //
        ↪ rechts
    }
}
```

DrawSquares

```
private void drawSquares(Graphics g, int left, int
↪ top, int size) {
    left = left + margin;
    top = top + margin;
    size = size - 2 * margin;
    if (size >= 4) {
        g.drawRect(left, top, size, size);
        int s = size / 2;
        drawSquares(g, left, top, s); // oben links
        drawSquares(g, left + s, top, s); // oben
        ↪ rechts
        drawSquares(g, left + s, top + s, s); // unten
        ↪ rechts
    }
}
```

DrawLines

```
void drawFigure(Graphics g, int x, int y, int len, int
↪ level) {

    if (level >= 0) {
        len = len / 3;
        drawFigure(g, x, y, len, level - 1);
        drawFigure(g, x + len, y + len, len, level -
        ↪ 1);
        drawFigure(g, x + 2 * (len), y, len, level -
        ↪ 1);
    } else {
        g.drawLine(x, y, x + len, y);
    }
}
```

DrawStars

```
private void drawFigure(int lineSize, int level) {

    if(level > 0) {
        drawFigure(lineSize/3, level - 1);
        turnLeft(60);
        drawFigure(lineSize/3, level - 1);
        turnRight(120);
        drawFigure(lineSize/3, level - 1);
        turnLeft(60);
        drawFigure(lineSize/3, level - 1);
    } else {
        draw(lineSize/3);
        turnLeft(60);
        draw(lineSize/3);
        turnRight(120);
        draw(lineSize/3);
        turnleft(60);
        draw(lineSize/3);
    }
}
```

DrawTriangles

```
private void drawRecTriangles(Point p1, Point p2,
↪ Point p3, int depth) {

    if(depth > 0) {
        drawTriangle(p1, p2, p3);
        drawRecTriangles(p1, midPoint(p2, p3),
        ↪ midPoint(p3, p1), depth -1);
        drawRecTriangles(midPoint(p1, p2),
        ↪ midPoint(p2, p3), p3, depth -1);
        drawRecTriangles(midPoint(p1, p2), p2,
        ↪ midPoint(p3, p1), depth -1);
    }
}
```

Fibonacci

```
public static long fiboRec(int n) {
    if (n > 1) {
        return fiboRec(n - 1) + fiboRec(n - 2);
    }
    return n;
}

public static long fiboIter(int n) {
    if (n < 1) return n;
    else {
        int i = 1;
        long f = 1, f1 = 0;
        while (i < n) { // Invariante: f = f(i) AND f1
        ↪ = f(i-1)
            f = f + f1;
            f1 = f - f1;
            i = i + 1;
        }
        return f;
    }
}
```

Merge-Sort

$T(n) = 2 * T(n/2) + n$

Worst-, Best- und Average-Case:

$\mathcal{O}(n * log(n))$, selten Worst-Case $\mathcal{O}(n^2)$

Zusätzlicher Speicher von $\mathcal{O}(n)$

```
private void sort(int[] a, int beg, int end) {

    if (end - beg > 1) {
        int m = (beg + end) >>> 1;
        sort(a, beg, m);
        sort(a, m, end);
        merge(a, beg, m, end);
    }
}

private void merge(int[] a, int beg, int m, int end) {
    int[] b = new int[end - beg];

    int i = 0, j = beg, k = m;
    while (j < m && k < end) {
        if (a[j] <= a[k]) b[i++] = a[j++];
        else b[i++] = a[k++];
    }
    while (j < m) {
        b[i++] = a[j++];
    }
    while (i > 0) {
```

```
            --i;
            a[beg + i] = b[i];
        }
    }
```

Merge Inplace

```
    private void sort(int[] a, int beg, int m, int end) {

            int j = beg, k = m;
            while(j != k && k != end) {
                    if(a[j] <= a[k]){
                            j++;
                    } else {
                            int temp = a[k];
                            for(int i = k; i > j;
                            ↪  i--){
                                    a[i-1] = a[i];
                            }
                            a[j] = temp;
                            k++;
                            j++;
                    }
            }

    }
```

Merge Halber-Speicher

```
    private void merge(int[] a, int beg, int m, int end) {
            int[] b = new int[m-beg];
            for(int i = beg; i < m; i++) {b[i-beg] =
            ↪  a[i];}
            int i = beg, j = 0, k = m;
            while (0 < (m - beg) || k < end) {
                    if(b[j] < a[k]){
                            a[i] = b[j]; i++; j++;
                    } else {
                            a[i] = a[k]; i++; j++;
                    }

            }
            while(j < (m-beg)){
                    a[i] = b[j]; i++; j++;
            }
    }
```

Quick-Sort

$T(n) = 2 * T(n/2) + a * n + b$

Best-Case: $\mathcal{O}(n * log(n))$

Average-Case: $\mathcal{O}(log(n))$

Worst-Case: $\mathcal{O}(n^2)$

Zusätzlicher Speicher von $\mathcal{O}(log(n))$

```
        static void sort(int[] a, int beg, int end) {

        if (beg < end) {
            int x = a[(beg + end) / 2];
            int i = beg;
            int j = end;

            while (i <= j) {
                while (a[i] < x) i++;
                while (a[j] > x) j--;
                if (i <= j) {
                    int t = a[i];
                    a[i] = a[j];
                    a[j] = t;
                    i++;
                    j--;

                }
            }
            sort(a, beg, j);
            sort(a, i, end);
        }
    }

    private void sort(SortData a, int beg, int end) {

        if (beg < end) {
            int x = (beg + end) / 2;
            int i = beg, j = end;
            while (i <= j) {
                while (a.less(i, x)) i++;
                while (a.less(x, j)) j--;
                if (i <= j) {
                    a.swap(i, j);
                    if (i == x) {
                        x = j;
                    } else if (j == x) {
                        x = i;
                    }

                    ++i;
                    --j;
                }
            }
            sort(a, beg, j);
            sort(a, i, end);
        }
    }


        void sort(int[]a, int beg, int end) {
        if(beg >= end) return;
        if(a[beg] > a[end]) swap(a, beg, end);
        int p1 = a[beg], p2 = a[end];

        int i = beg + 1, k = i, j=end;
        while(k != j ){
            if(a[k] <= p1){
                swap(a,i,k); i++; k++;
            }else if(a[k] <= p2){
                k++;
            }else{
                j--; swap(a,k,j);
            }
        }
        sort(a, beg, i-1);
        sort(a, i, j-1);
        sort(a, j, end);
    }
```

Summenformeln

Gauss

$$\sum_{k=1}^{n} k = \frac{n(n+1)}{2} \tag{1}$$

Gerade Zahlen

$$\sum_{k=1}^{n} 2k = n * (n+1) \tag{2}$$

Ungerade Zahlen

$$\sum_{k=1}^{n} 2 * (k-1) = n^2 \tag{3}$$

Quadrat Zahlen

$$\sum_{k=1}^{n} k^2 = \frac{n * (n+1) * (2n+1)}{6} \tag{4}$$

Kubische Zahlen

$$\sum_{k=1}^{n} k^3 = (\frac{n * (n+1))}{2})^2 \tag{5}$$

Sortier-Algorithmus: (vereinfacht):

```
int i = 0;           1.
while (i != n - 1) {
2.  min(a, i);       // Rearranges Elements a[i] to [n-1] so that (∀ l: i+1 ≤ l < n : a[i] ≤ a[l])
    i = i + 1;
}  3.
```

Um zu zeigen, dass es sich um eine gültige Invariante handelt / Programm korrekt funktioniert, muss gezeigt werden, dass:

1. Gilt *Inv*, bevor die Schleife das erste Mal startet? (zu beweisen: wp(`i = 0`, Inv) ≡ true)
2. Gilt jeweils am Anfang eines Schleifendurchlaufs die Vorbedingung dafür, dass am Ende der Schleife *Inv* (wieder) gilt? (zu beweisen: (i ≠ n-1) ∧ Inv ⇒ wp(`min(a, i); i = i + 1`, Inv))
3. Gilt die Nachbedingung (∀ k : 0 < k < n : a[k-1] ≤ a[k]), wenn die Schleife endgültig beendet ist? (zu beweisen: ¬(i ≠ n-1) ∧ Inv ⇒ (∀ k : 0 < k < n : a[k-1] ≤ a[k]))

wp(`while` (E_b) S, R) ≡ Inv ∧ (E_b ∧ Inv ⇒ wp(S, Inv)) ∧ (¬E_b ∧ Inv ⇒ R) 4

Inv. gilt vorher — Inv. bleibt gültig, wenn Schleife läuft — Aus Invariante und Schleifenende folgt Nachbed.

Algorithmen u 18