Praktische Informatik

Informationen zur Lehrveranstaltung

Tutorium

Montag, 18:15 – 20:00, RR 15

Benotung

- Mindestens 50% aller Hausübungen müssen erledigt werden
- Mindestens eine positive Präsentation
- Schriftliche Prüfung muss positiv sein.

Die Gesamtnote ergibt sich aus (Summe d. drei Noten) / 3, wobei alle positiv sein müssen.

Schriftliche Prüfung

Findet am Freitag, 13. Jänner 2017, 12:00 – 17:00 im RR 15 statt.

Klausuren

- Freitag, 16. Dezember 2016, 15:00 – 17:00, Hörsaal A
- Mittwoch, 18. Jänner 2017, 12:00 – 14:00, Hörsaal A
- Mittwoch, 1. März 2017, 10:00 – 12:00, Hörsaal A

Inhaltsverzeichnis

Unterschiedliche Teilgebiete der Informatik

Die Informatik teilt sich auf in

Angewandte Informatik	Praktische Informatik
Technische Informatik	Theoretische Informatik

Angewandte Informatik

Das ist die **Informatik, wie sie der Anwender kennt**. Der **wirtschaftliche, kommerzielle Nutzen**, die **technische-wissenschaftliche Verwendung** bei z.B. Ampelanlagen, Flugüberwachungssystemen.

Sie teilt sich wiederrum auf in kleinere Gebiete, wie Wirtschaftsinformatik, Computervisualisierung, Künstliche Intelligenz, Juristische Informatik, Chemieinformatik.

Praktische Informatik

Darunter fallen die Grundlagen der Systemsoftware, Programmiersprachen, Algorithmen und Datenstrukturen, Compiler und Interpreter, Betriebssysteme, Netzwerke, Datenbanken, ...

Sozusagen alles, was wir als Informatiker in der Praxis tun werden, die kein Anwender verstehen wird :P

Technische Informatik

Beschäftigt sich mit der Konstruktion von Rechnern, der Hardware und dem Allgemeinen Aufbau eines Computers.

Weiteres betrifft dies auch die Mikroprozessortechnik, Rechnerarchitektur, Rechnerkommunikation und alle Peripheriegeräte.

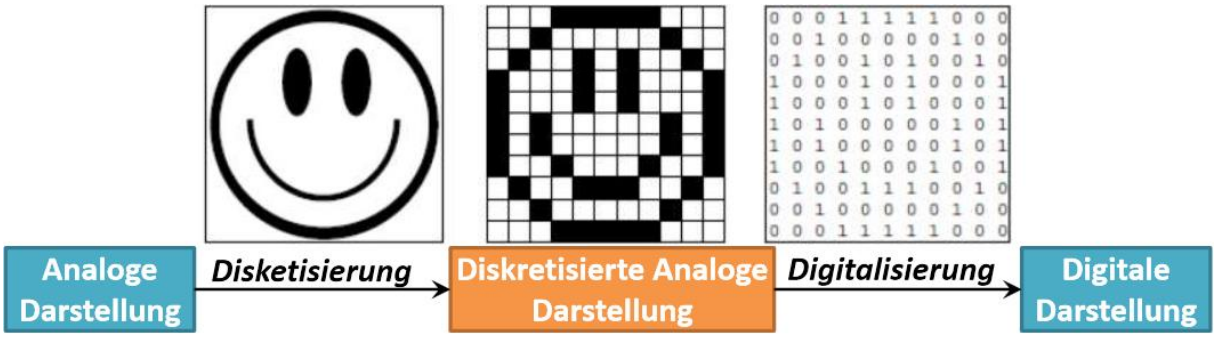
Theoretische Informatik

Alle mathematischen und logischen Grundlagen für die anderen Kerngebiete.

Darunter fallen die Automatentheorie, Formale Sprachen, Komplexitätstheorie und die Berechenbarkeitstheorie.

Unnötiges Wissen was vielleicht geprüft werden könnte

Diskretisierung und Digitalisierung



Diskretisierung

Nennt sich das Festlegen eines Rasters von Messpunkten gleichen Abstands auf einer Achse, über die sich das Signal verändert.

Diskret *= Unstetig, getrennt*

Digitalisierung

Ist die Darstellung der ermittelten Messwerte in einem festen, endlichen Werteraster des Binärsystems. Dabei gilt, desto mehr Bits pro Messwert, desto genauer kann das Ursprungssignal wiedergegeben werden.

Vorteile Digitaler Darstellung

Durch die Hinzunahme von Bits kann man eine Darstellung **beliebig genau** machen.

Faustregel: 10 Bits sind ungefähr 3 Dezimalstellen

Geringe Störempfindlichkeit, da es nur zwei physikalische Größen gibt: Strom oder Kein Strom

Verlustlose Speicherung, und vorallem sehr einfach.

Leichtere Übertragung, durch Elektrik, Elektromagnetisch oder Optisch (Glasfaser)

Programmieren

Bezeichnet die Tätigkeit, Programme zu entwerfen. Programme wiederrum bestehen in ihrer allgemeinsten Form aus Daten (Objekten) und Algorithmen, die Operationen mit den Daten durchführen.

Dazu müssen die Aufgabenstellungen an das Programm spezifiziert werden

Spezifikation

- **Vollständig**: alle relevanten Informationen sind berücksichtigt
- **Detailliert**: alle Hilfsmittel und Aktionen die zur Lösung zugelassen sind, sind aufgelistet und protokolliert.
- **Unzweideutig**: klare Kriterien sind festgelegt, wann eine Lösung akzeptabel ist.

Beispiel:

„Zu zwei Zahlen A und B ist der größte gemeinsame Teiler gesucht“ – das ist eine ungenaue Spezifikation.

Vollständig: Dürfen A und B rationale oder ganze Zahlen sein? Müssen sie positiv sein? Was ist mit 0? Maximum? Wie werden die Zahlen erzeugt?
Detailliert: Dürfen A und B verändert werden?
Unzweideutigkeit: Was bedeutet „ist gesucht?“ Soll Ergebnis ausgegeben, gedruckt, gespeichert werden?

Weil das alles ziemlich viele Fragen werden würden, gibt’s die formale Spezifikation:

Formale Spezifikation

Damit wird mathematisch und logisch die Spezifikation ausgedrückt:

Das heißt, wir definieren eine Vorbedingung für das Beispiel oben:
$A, B \in \mathbf{N}$
$(0 < A < 2^{15}) \wedge (0 < B < 2^{15})$

Und eine Nachbedingung dafür.
(z | A [z ist ein Teiler von A]) $\wedge$ (z | B)
$\forall$ z‘ gilt (z‘ | A) $\wedge$ (z‘ | B) => z‘ $\leq$ z

Zahlensysteme

Positionssysteme

Ist ein Zahlensystem, in dem eine Zahl *N* nach Potenzen der Basis B zerlegt wird.

Die Zahl kann durch folgende Summenformel (der Ziffern) dargestellt werden:

$$N = \sum_{i=0}^{n-1} z_i \cdot B^i$$

Obwohl das Zehnersystem ein Positionssystem ist, kann der Rechner schwer damit umgehen. Das **Binärsystem** lässt sich relativ simpel umsetzen, weil es entsprechend der Basis 2 auch nur 2 Ziffern geben kann – eben Strom oder Kein Strom, 0 oder 1.

Jede Potenz von 2 ist daher auch sehr einfach im Rechner umzusetzen, so auch das **Oktal** und **Hexadezimalsystem**.

Zwischen den Systemen, die eine Potenz von 2 sind (also die Fett markierten), kann man ganz einfach konvertieren, indem man jede Zahl/Ziffer in eine Binärzahl umwandelt, und dann entsprechend dem Zielsystem die dafür benötigte Anzahl an Ziffern im Binärsystem zur Ziffer im Zielsystem tauscht:

Oktal: 74 = Binär:	111 | 100
Umgewandelt in Hexadezimal:	0011 | 1100
	= 3C

Hexadezimal

Bildet daher Vierergruppen im Binärsystem.

Oktalsystem

Bildet Dreiergruppen im Binärsystem.

Horner-Schema

Damit lässt sich eine Zahl entsprechend der Basis darstellen:

$N = ((((z_{n-1} * B + z_{n-2}) * B + z_{n-3}) * B + z_{n-4}) * B + \ldots + z_1) * B + z_0$

Umwandlung vom Dezimalsystem in andere Systeme

N wird durch die Basis dividiert. Das Resultat wird weiter dividiert. Der Rest bildet in umgekehrter Reihenfolge (von unten nach oben) die gewünschte Zahl.

Bruchzahlen

$$N = \sum_{i=-m}^{n-1} z_i \cdot B^i$$

Wobei B die Basis, z die Ziffern, n die Anzahl der Stellen vor dem Punkt und m die Anzahl nach dem Punkt repräsentiert.

Umwandlung echter Brüche vom Dezimal in Andere

Mit drei Schritten kann umgewandelt werden:

1. *N* * *B* = Überlauf , Nachkommateil
2. Der Nachkommateil wird wieder mit der Basis multipliziert → Schritt 1
   Während der Überlauf jedoch notiert wird.
3. Die Reihenfolge von Überläufen liefert (von oben nach unten) die gewünschte Zahl.

Umwandlung unechter Brüche

Werden aufgeteilt in einen ganzzahligen Teil und ihren echten Bruchteil aufgeteilt und werden wie gehabt abgewandelt.

Komplementbildung

Einer-Komplement ((B-1)-Komplement)

Ist relativ schwierig zu realisieren, daher auch nicht weiter von uns verfolgt. Zur Erklärung:

Die Zahlendarstellung ist symmetrisch, das heißt die Zahlen werden exakt invertiert:

Auffällig: Es gibt zwei 0en, eine Positive und eine Negative. Weil auch das Schwachsinn ist, ist dieses Komplement nicht verwendbar.

Positive Zahlen		Negative Zahlen	
Dezimal	*Binär*	*Dezimal*	*Binär*
0	**0000**	– 7	1000
1	0001	– 6	1001
2	0010	– 5	1010
3	0011	– 4	1011
4	0100	– 3	1100
5	0101	– 2	1101
6	0110	– 1	1110
7	0111	**– 0**	**1111**

Bildung

Alle Zahlen werden exakt invertiert.

Sollte bei einer Addition ein Überlauf auftreten, wird dieser hinten dazu-addiert.

Zweier-Komplement (B-Komplement)

Alle Zahlen, bei denen das erste Bit gesetzt ist, repräsentieren negative Zahlen.

Die Darstellung ist unsymmetrisch, und als Kreis zu betrachten. Das heißt 0000 ist 0 und 1111 ist -1

Positive Zahlen		Negative Zahlen	
Dezimal	*Binär*	*Dezimal*	*Binär*
0	0000	– 8	1000
1	0001	– 7	1001
2	0010	– 6	1010
3	0011	– 5	1011
4	0100	– 4	1100
5	0101	– 3	1101
6	0110	– 2	1110
7	0111	– 1	1111

Bildung

Jedes einzelne Bit wird invertiert und darauf wird noch einmal 1 addiert.

Addition und Subtraktion

Dank der Verwendung der Komplement-Darstellung, muss ein Rechner nur Addieren können. Bei der Addition werden die Bits einfach addiert, 1+1 = 10. Sollte dabei die höchsten Bits einen Übertrag produzieren, nennt sich dies Überlauf. Dieser wird weggeworfen, wenn es nicht mehr in den Speicher passt.

Multiplikation und Division

Auch die Multiplikation bzw. Division kann einfach durch ein wiederholtes Addieren realisiert werden.

Dabei ist es besonders einfach, wenn der Multiplikator/Divisor ein Vielfaches von 2 ist, denn dann kann die Multiplikation bzw. Division einfach durch Verschiebung nach links bzw. rechts erfolgen.

Operation	Dezimal	Binär
	8	$1000 = 2^3$
·	20	$000 10100 = 2^4 + 2^2$
	160	$10100000 = 2^7 + 2^5$

Operation	Dezimal	Binär
	20	$10100 = 2^4 + 2^2$
/	4	$100 = 2^2$
	5	$00101 = 2^2 + 2^0$

Gleitpunktzahlen

Jede reelle Zahl kann auch in normalisierter Form dargestellt werden:

$$N = z_{n-1}\, z_{n-2} \ldots z_1\, z_0 \,.\, z_{-1} \ldots z_{-m+1}\, z_{-m} \quad = \quad y_0 \,.\, y_1\, y_2 \ldots y_{m+n-1} * 10^{n-1}$$

Dann hat die Zahl zwei Bestandteile:

Mantisse: $y_0 \,.\, y_1\, y_2 \ldots y_{m+n-1}$
Exponent: n-1 (Ganzzahl)

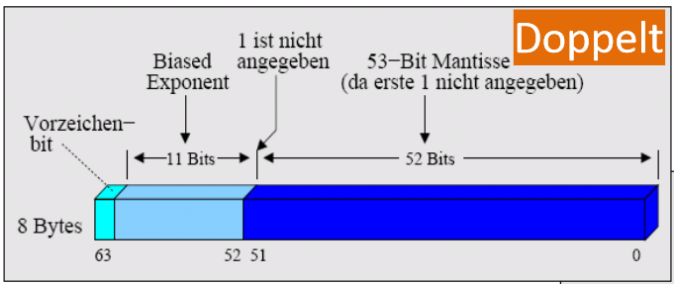
Der Exponent wird dabei so angepasst, dass die erste Nicht-Null Ziffer immer links vom Dezimalpunkt steht, der Rest rechts davon.

Mantisse Exponent und Vorzeichen

Weil im Rechner eh nur Binär vorhanden ist, ist das erste Nicht-Null Zeichen immer eine 1. Daher können wir dieses de-facto weglassen und schaffen damit Platz für das Vorzeichenbit.

Der Exponent hingegen wird mit einem sogenannten Bias addiert und ohne Vorzeichen dargestellt.

Der Bias ist für float 127, für double 1023!



Dezimalzahl	17.625
Binärdarstellung	10001.101
Normalisierte Form	$1.0001101 \cdot 2^4$
Vorzeichen	0
Mantisse	0001101
Exponent	00000100
Bias	01111111
Biased Exponent	10000011

Addition von Gleitpunktzahlen

Zum Addieren müssen wir die Gleitpunktzahlen auf denselben Exponenten bringen, dazu passen wir den kleineren an den größeren an. Danach wird einfach addiert ☺

Binärdarstellung von Reellen Zahlen

Bei Festpunktzahlen steht das Komma an einer definierten Stelle, wobei wir eine eingeschränkte Genauigkeit mit n für ganze Zahlen und m für echte Brüche haben.

Diese Darstellung wird nur in Spezialanwendung verwendet.

Subtraktion von Reellen Zahlen

Die Komplementdarstellung wird gleich wie bei Ganzzahlen gebildet und angewandt. Das heißt Ganzzahl und Nachkommateil für Festpunktzahlen und Mantisse für Gleitpunktzahlen.

IEEE-Format

IEEE-Format	Einfach	Doppelt
Vorzeichen-Bits	1	1
Exponenten-Bits	8	11
Mantissen-Bits	23	52
Bits insgesamt	32	64
Bias	127	1023
Exponentenbereich	[– 126, 127]	[– 1022, 1023]

Sonderfälle

Biased Exponent	Mantisse	Bedeutung
111...111 (= 255 bzw. 2047)	≠ 0	Keine gültige Zahl (*NaN*)
111...111 (= 255 bzw. 2047)	000...000 (= 0)	$\pm\infty$
000...000 (= 0)	000...000 (= 0)	$\pm$ 0

Zeichencodecs

ASCII (American Standard f. Coded Information Interchange)

Festgelegte Abbildungsvorschrift zur binären Kodierung von Zeichen, welche Klein-/Großbuchstaben des lateinischen Alphabets, (arabische) Ziffern und einige Sonderzeichen umfasst. Es enthält 256 Zeichen (weil 8 Bit), wobei das erste nicht genutzt wird – daher sind im Standard-ASCII nur 128 Zeichen dargestellt.

Unicode

Weil 256 Zeichen nicht für alles ausreicht, gibt's den Unicode. Damit lässt sich praktisch jede bekannte Schriftkultur und Zeichensysteme beschreiben. Der Unicode wurde ursprünglich 2 Byte groß definiert, umfasste daher 2^{16} = 65536 Zeichen. In der Version 3.1 wurden jedoch 94000 neue Zeichen aufgenommen, womit die Zeichengrenze erreicht wurde. Jetzt sind es daher 4-Byte.

BCD-Code (Binary Coded Decimals)

Für jede Dezimal**ziffer** werden mindestens vier, manchmal acht Bits verwendet. Dann werden die jeweiligen Ziffern nacheinander durch ihren Binärwert angegeben, was natürlich in einer Speicherplatzverschwendung resultiert, weil nur die Ziffern bis 9 (also bis 1001) repräsentiert werden können. Es lässt sich allerdings sehr einfach verwenden.

Gray-Code

Zwei aufeinanderfolgende Codewörter unterscheiden sich immer nur um genau ein Bit. Wird für Ausgabe von Werten von Analog/Digital-Wandlern zur Vermeidung unsinniger Zwischenwerte.

Dezimal	Grey (Binär)
1	0001
2	0011
3	0010
4	0110
5	0111
6	0101
7	0100
8	1100

Grunddatentypen

Datenstruktur

Man versteht darunter den Datentyp zusammen mit den Operationen, die auf diesem Typ erlaubt sind. Zum Beispiel ist der Module in C nur für short & int.

Datentypen

Daten müssen einer Struktur zugeteilt werden, die eine Bedeutung haben und Operationen ermöglichen. Dazu dienen Datentypen.

- Zeichen (char, strings [char arrays])
- Natürliche Zahlen (unsigned int)
- Ganze Zahlen (int)
- Gleitpunktzahlen (float, double)
- Boolesche Werte (bool)
- Strukturierte Datentypen (Arrays, Strukturen, Zeiger, Funktionen)

Array (Feld)

Ist eine der wichtigsten Datenstrukturen, um mit geordneten Mengen von Variablen gleichen Typs umzugehen.

Dabei sind die einzelnen Elemente über die Indizes ansprechbar. Zweidimensionale Arrays nennt man in der Mathematik Matrizen. Beachte, Mehrdimensionale Arrays explodieren rasch in ihrem Speicherbedarf.

Genauere Informationen sind im Kompendium Programmierung zu finden.

Algorithmus

Wenn ein Problem spezifiziert wurde, muss ein Lösungsweg entworfen werden.

Der Algorithmus dient genau dazu, um eine detaillierte und explizite Vorschrift zur schrittweisen Lösung eines Problems zu bieten. Sie muss präzise formuliert, in endlicher Form dargestellt und effektiv ausführbar sein.

Definition

Ein Algorithmus ist eine in der Beschreibung und Ausführung endliche, deterministische und effektive Vorschrift zur Lösung eines Problems, die effizient sein sollte.

- Endlich: Beendigt sich nach einer endlichen Zeit
- Deterministisch: Eindeutige Bestimmung des nächsten Schrittes
- Effektiv: Eindeutige Ausführbarkeit der Einzelschritte
- Effizient: Geringer Verbrauch an Ressourcen.

Das heißt, wenn wir die Spezifikation (P, Q) haben, wobei P die Vorbedingung und Q die Nachbedingung ist, dann dient der Algorithmus A dazu, die Vorbedingung P in die Nachbedingung Q zu überführen: {P} A = {Q}

Aber wie überprüft man, welcher Algorithmus für dasselbe Problem effizienter sei?

Komplexitätstheorie

Darunter versteht man den groben Bedarf an Ressourcen in Abhängigkeit von Umfang der Eingabedateien (**Zeitaufwand, Speicherbedarf**, usw.)

Laufzeitkomplexität

Man schätzt die maximale Anzahl an „atomaren Schritten“ ab. Zum Beispiel die Anzahl an arithmetischen Operationen, gelesenen Elementen, Zuweisungen, usw. Diese Formel soll allerdings von einer variablen Problemgröße N gefunden werden.

Sei zum Beispiel ein Array mit N Elementen gegeben, müsste man bei der Problemgröße N maximal N/2 Elemente durchlaufen, um festzustellen, ob alle geraden Indizes über 0 liegen.

Komplexitätsklassen

Weil die genaue Laufzeit nahezu unmöglich zu berechnen ist, müssen wir uns auf das Wesentliche durch Klassifikation reduzieren.

Wir verwenden dazu die O-Notation, um den asymptotischen (gegen unendlich gehenden) Wachstum einer Funktion zu beschreiben

- Funktion f ist O(g), falls f asymptotisch nicht schneller wächst als g

Typische Komplexitätsklassen

- $O(N) \rightarrow$ Linear
- $O(N^2) \rightarrow$ Quadratisch
- $O(N^3) \rightarrow$ Kubisch
- $O(1) \rightarrow$ Konstant
- $O(\log N) \rightarrow$ Logarithmisch
- $O(2^n) \rightarrow$ Exponentiell

Speicherplatzbedarf

Ebenso wichtig wie der Zeitaufwand ist der Speicherplatzbedarf für die Charakteristik eines Algorithmus.

Wird beispielsweiße ein Array auf doppelte Elementwerte überprüft, benötigen wir dazu nur einen konstanten Speicherbedarf (weil wir ja nur vergleichen, nix speichern/verändern). Die Laufzeitkomplexität ist jedoch $O(N^2)$, weil wir ja jedes Element mit jedem Element vergleichen müssen.

Programmablaufplan (PAP)

Ein PAP ist ein Ablaufdiagram für Computerprogramme. Sie werden auch Flussdiagram oder Programmstrukturplan bezeichnet und dienen zur graphischen Darstellung der Umsetzung eines Algorithmus.

Dabei gibt es sechs verschiedene Elemente dafür:

Start, Stop, Grenzpunkte:

Verbindung zum nächsten Element:

Operationen

Unterprogram aufrufen

Verzweigung

Ein-/Ausgabe

Suchalgorithmen

Das Problem: Wie finde ich möglichst effizient Elemente in einer Liste?

Ohne Vorbedingungen lässt sich die Suche linear gestalten, also einfach alle Elemente durchlaufen. Ist jedoch eine Vorbedingung gegeben, so muss der Array sortiert werden: Für Binäre Suche oder Interpolationssuche.

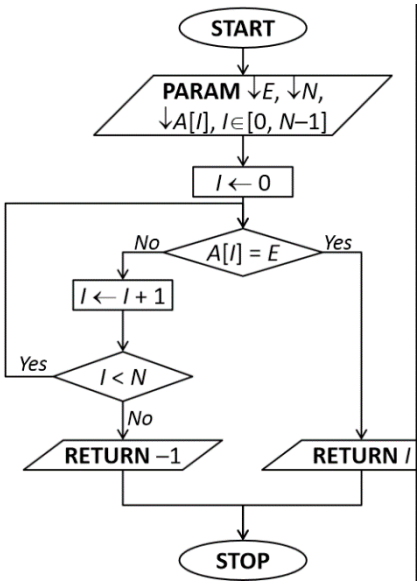
Aufgabenstellung

Eine Funktion bekommt drei Eingabeparameter: eine Arraygröße N, ein Array A mit Elementen, und ein Element E. Es soll einen Algorithmus implementiert werden, der überprüft ob das Element E im Array A vorhanden ist. Falls das Element gefunden wird, soll die Funktion den Index I des Elements E in Array zurückgeben (A[I] = E) und –1 sonst.

Lösungen

Lineare Suche

```
def linearSearch(e, n, a):
    for i in range(0, n):
        if(a[i] == e):
            return i
    return -1
```



Komplexität

Im besten Fall ist das erste Element das gesuchte Element. Dann ist die die Komplexität O(1), wovon aber nicht ausgegangen wird. Wir gehen immer vom schlimmsten aus, daher O(N)

Binäre Suche („Teile und Herrsche“)

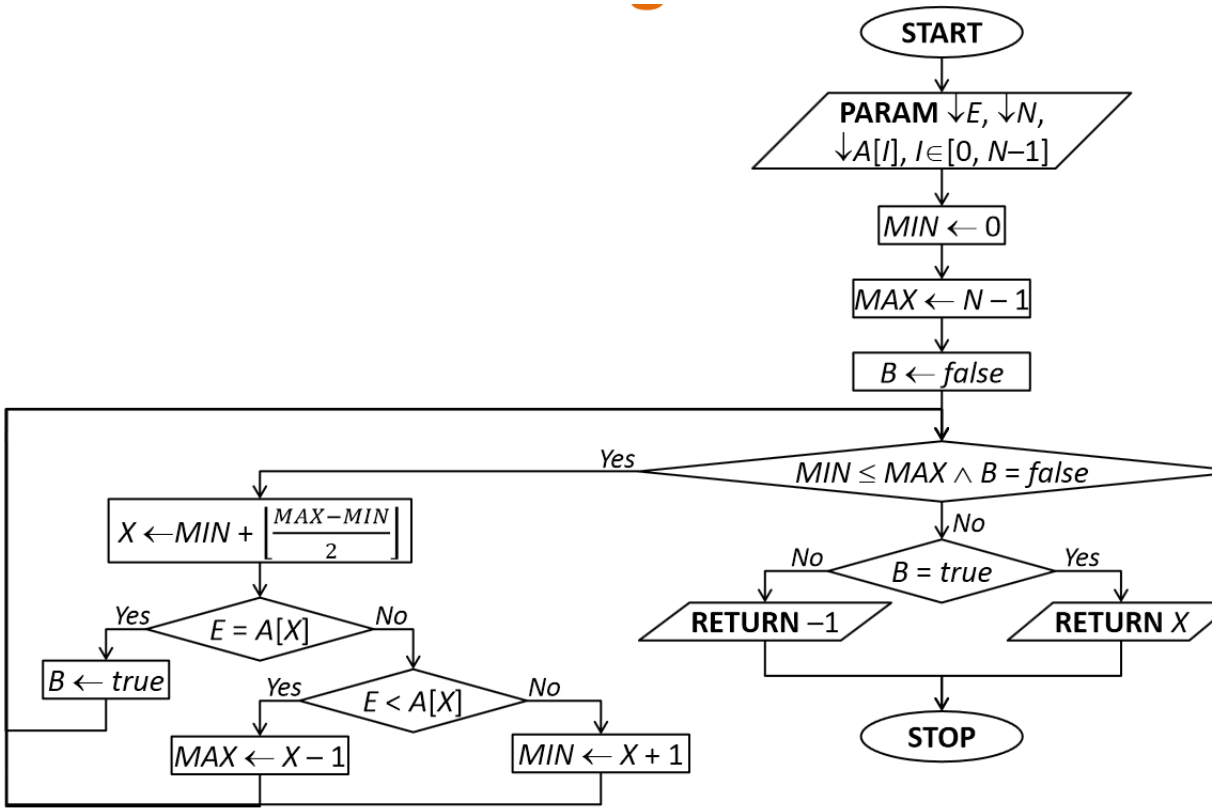
Die Vorbedingung dafür ist, dass A sortiert ist:
$A[i] < A[i+1], \forall\ i \in [0, n-2]$

Nun wird das Array schrittweise „halbiert“, das heißt wir beginnen mit dem Intervall [0, n-1] bei n/2. Wenn dieses Element (also das mittlere Element des Intervalls) gleich dem gesuchten Element ist, haben wir es gefunden! Yay!

Wenn es größer als das Gesuchte ist, dann suchen wir in der unteren Hälfte weiter und halbieren wieder. Ist es kleiner als das Gesuchte, suchen wir in der oberen Hälfte weiter.
Sobald das Intervall leer ist, ist das Element nicht vorhanden -> Fertig.

Damit wird mit jedem (und **mit einem einzigen**) Vergleich das übriggebliebene Suchintervall halbiert.

```
def binarySearch(e, n, a):
    min = 0
    max = n - 1
    while(min <= max):
        i = int(min + (max - min) / 2)
        if(a[i] == e):
            return i
        elif(a[i] < e):
            min = i + 1
        else:
            max = i - 1
    return -1
```



Komplexität

Auch hier ist im besten Fall das Element das Erste, in der Mitte. Im schlimmsten Fall jedoch ist unser gesuchtes Element an der Stelle $(N/2^p)$ ➔ $2^p = N$ ➔ $p = \log(N)$ und damit $O(\log_2 N)$

Interpolationssuche

Das Array wird nicht in der Mitte, sondern in Abhängigkeit vom Schlüssel geteilt. Dabei ist die Position = (#Elemente / #verschiedener Elemente) * gesuchtem Wert

Zum Beispiel hat das deutsche Alphabet 30 versch. Buchstaben. Wir suchen nach einem Namen im Telefonbuch mit 1000 Seiten. Dann wäre:

- Pos(A) = [(1000/30) * (A – A)] = 0 — Weil A = 1
- Pos(B) = [(1000/30) * (B – A)] = [(1000/300) * 1] = 33 — Weil B = 2
- Pos(B) = [(1000/30) * (P – A)] = [(1000/300) * 18] = 600 — Weil P = 19

Auch hier ist eine gleichmäßige Verteilung des Arrays nötig.

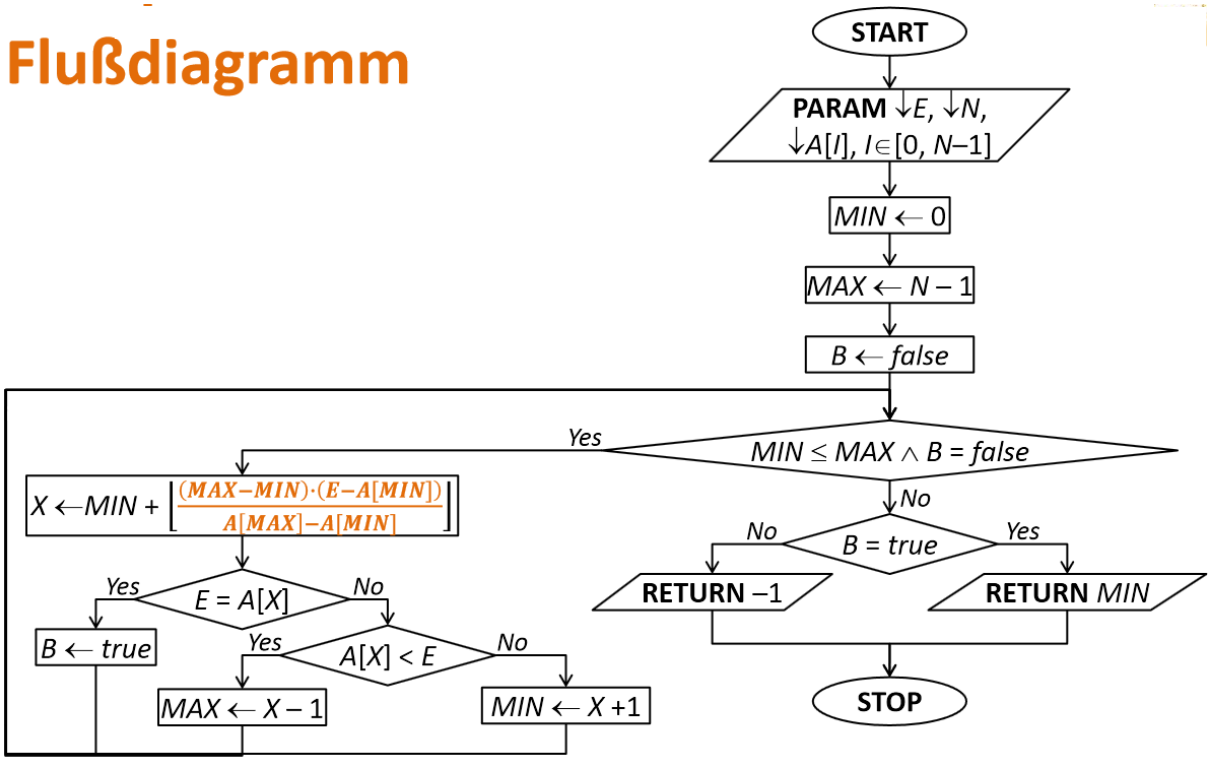
Komplexität

Auch hier im besten Fall das Erste O(1).

Im schlimmsten Fall jedoch O(N), also sehr performant!

```
def interpolationSearch(e, n, a):
    min = 0
    max = n – 1
    while(min <= max):
        if(a[min] == a[max] == e):
            return min
        if(e < a[min] or e > a[max]):
            return -1
        i = int(min + (max – min) * (e – a[min]) /
                                                (a[max] – a[min]))
        if(a[i] == e):
            return i
        if(a[i] < e):
            min = i + 1
        else:
            max = i – 1
    return –1
```

Flußdiagramm



Sortieralgorithmen

Das erste Computerprogramm war ein Sortierprogramm, um nichtnumerische Fähigkeit des Computers zu demonstrieren. Einigen Gerüchten zufolge gäbe es kein Programmierproblem, bei dem nicht früher oder später Sortierung nötig wäre.

Klassifizierung der Sortieralgorithmen

Speicherstelle der Daten

- Interne Sortieralgorithmen, wenn sich die Daten im Arbeitsspeicher befinden (z.B. Arrays)
- Externe, wenn sich die Daten auf einem Peripheriegerät befinden (z.B. Festplatte)

Speicherplatz
Wird zusätzlicher Speicher benötigt? Wird kein zusätzlicher benötigt? Wird doppelter benötigt?

Stabilität
Sortieralgorithmen gelten als stabil, wenn er bei Elementen, die nach dem Sortierkriterium gleich sind, die zuvor vorliegende Reihenfolge der Elemente relativ zueinander beibehält.
Also wenn sie gleich sind, sollen zwei gleiche Elemente in gleicher Reihenfolge wie davor hintereinanderstehen.

Komplexität
Zwischen O(n * log n) und O(n^2).

Bubblesort
Ist ein vergleichsbasierter Sortieralgorithmus, bei dem der Array mehrmals durchlaufen wird und stets die Elemente A[i] und A[i+1] vertauscht – solange bis keine Vertauschungen mehr notwendig sind. Dadurch wird er maximal N mal durchgeführt, für eine Arraylänge N.

Es ist ein stabiler Sortieralgorithmus, mit minimalen Speicherverbrauch. Die Komplexität ist O(N^2).

Durchlauf	A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]
	19	96	30	11	73	2	99	72	69	1
1	19	**30**	**11**	**73**	**2**	**96**	**72**	**69**	**1**	**99**
2	19	**11**	**30**	**2**	**73**	**72**	**69**	**1**	**96**	99
3	**11**	**19**	**2**	**30**	**72**	**69**	**1**	**73**	96	99
4	11	**2**	**19**	30	**69**	**1**	**72**	73	96	99
5	**2**	**11**	19	30	**1**	**69**	72	73	96	99
6	2	11	19	**1**	**30**	69	72	73	96	99
7	2	11	**1**	**19**	30	69	72	73	96	99
8	2	**1**	**11**	19	30	69	72	73	96	99
9	**1**	**2**	11	19	30	69	72	73	96	99

Schlecht

```
def bubbleSort(a, n):
    b = False
    while(not b):
        b = True
        for i in range(0, n-1):
            if(a[i] > a[i+1]):
                t = a[i]
                a[i] = a[i+1]
                a[i+1] = t
                b = False
```

Shakersort

Ist eine Erweiterung des Bubblesort, weil sich dort die großen Elemente ja nur nach hinten bewegen, aber keine kleinen Elemente nach vorne kommen.

Daher wird im Shakersort abwechselnd von links nach rechts durchlaufen, damit die kleineren Elemente auch nach vorne kommen. Auch dieser ist ein stabiler Algorithmus, er braucht aber weniger Schritte als der Bubblesort

Durchlauf	A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]
	19	96	30	11	73	2	99	72	69	1
1 →	19	30	11	73	2	96	72	69	1	99
2 ←	1	19	30	11	73	2	96	72	69	99
3 →	1	19	11	30	2	73	72	69	96	99
4 ←	1	2	19	11	30	69	73	72	96	99
5 →	1	2	11	19	30	69	72	73	96	99
6 ←	1	2	11	19	30	69	72	73	96	99

```
def shakerSort(a, n):
    b = False
    while(not b):
        b = True
        for i in range(0, n-1):
            if(a[i] > a[i+1]):
                t = a[i]
                a[i] = a[i+1]
                a[i+1] = t
                b = False
        for i in range(n-1, 0):
            if(a[i] < a[i-1]):
                t = a[i]
                a[i] = a[i-1]
                a[i-1] = t
                b = False
```

Insertionsort

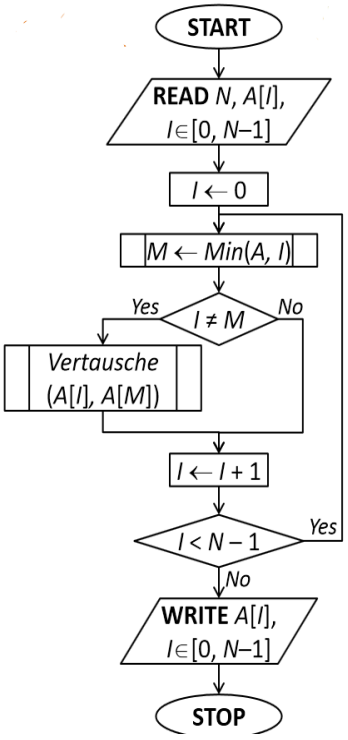
Man zieht (ähnlich wie bei Skat, falls des wer kennt...) eine Karte und fügt sie an die richtige Position in den bereits gezogenen Karten ein. Dabei sind die gezogenen Karten aufsteigend sortiert. Ist ein Stabiler Algorithmus, mit der Komplexität $O(N^2)$.

i	A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]
	19	96	30	11	73	2	99	72	69	73
0	19	96	30	11	73	2	99	72	69	73
1	19	96	30	11	73	2	99	72	69	73
2	19	30	96	11	73	2	99	72	69	73
3	11	19	30	96	73	2	99	72	69	73
4	11	19	30	73	96	2	99	72	69	73
5	2	11	19	30	73	96	99	72	69	73
6	2	11	19	30	73	96	99	72	69	73
7	2	11	19	30	72	73	96	99	69	73
8	2	11	19	30	69	72	73	96	99	73
9	2	11	19	30	69	72	73	73	96	99

```
def insertionSort(a, n):
    for i in range(0, n-1):
        j = i + 1
        while(a[j] < a[j-1] and j > 0):
            swap(a, j, j-1)
            j = j - 1
```

Selectionsort

I	*A*[0]	*A*[1]	*A*[2]	*A*[3]	*A*[4]	*A*[5]	*A*[6]	*A*[7]	*A*[8]	*A*[9]
	19	96	30	11	73	2	99	72	69	73
0	19	96	30	11	73	**2**	99	72	69	73
1	**2**	96	30	**11**	73	19	99	72	69	73
2	**2**	**11**	30	96	73	**19**	99	72	69	73
3	**2**	**11**	**19**	96	73	**30**	99	72	69	73
4	**2**	**11**	**19**	**30**	73	96	99	72	**69**	73
5	**2**	**11**	**19**	**30**	**69**	96	99	**72**	73	73
6	**2**	**11**	**19**	**30**	**69**	**72**	99	96	**73**	73
7	**2**	**11**	**19**	**30**	**69**	**72**	**73**	96	99	**73**
8	**2**	**11**	**19**	**30**	**69**	**72**	**73**	**73**	99	**96**
9	**2**	**11**	**19**	**30**	**69**	**72**	**73**	**73**	**96**	99




```
def selectionSort(a, n):
    for i in range (0, n-1):
        m = minIndex(a, n, i)
        if(i != m):
            swap(a, i, m)
```

Jop, auf der Vorlesungsfolie war nix dazu und i war zu faul nachzuschauen – dann weat er schon nicht so wichtig sein haha

Quicksort („Teile und Herrsche“)

Ist einer der am häufigsten verwendeten Sortieralgorithmen, basierend auf dem Prinzip der Rekursion (Seite 28). Der Vorteil ist ein geringer Speicherbedarf, weil die Daten direkt (nur mit Hilfsstack) sortiert werden und im durchschnittlichen Fall eine Komplexität von O(n * log n) aufweist. Im schlimmsten Fall jedoch $O(n^2)$.

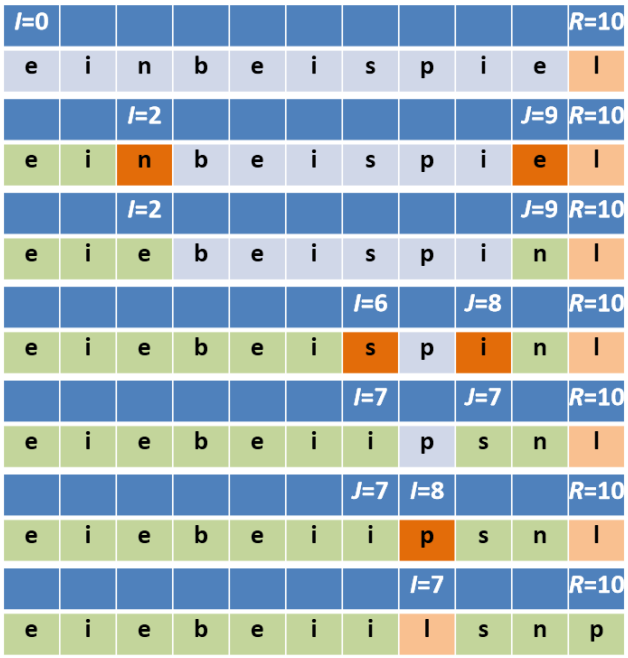
Teile... den Array A[0:N] in 2 Teilarrays A[0:P-1] und A[P+1:N], sodass alle Elemente in A[0:P] kleiner als alle in A[P+1:N] sind, wobei P = ***Partition***(A, 0, N) ist.

... und Herrsche, indem die Teilarrays durch 2 rekursive Aufrufe des Quicksort sortiert werden. ***Quicksort***(A, 0, P-1) und ***Quicksort***(A, P+1, N)

Der erste Aufruf von Quicksort ist dann ***Quicksort***(A, 0, N), womit sich die Rekursion darin erklärt.

```
def quicksort(a, l, r):
    if (l < r ):
        p = partition(a, l, r);
        quicksort(a, l, p - 1);
        quicksort(a, p + 1, r);
```

Partition Trace:




```
def partition(a, l, r):
  i = l
  j = r - 1
  while i < j:
    while i < r and a[i] <= a[r]:
      i = i + 1
    while j > l and a[j] >= a[r]:
      j = j - 1
    if i < j:
      swap(a, i, j)
  swap(a, i, r)
  return i
```

Diverse Algorithmen, die praktisch sein könnten

Euklidischer Algorithmus

Zu zwei positiven natürlichen Dezimalzahlen A und B, soll der größte gemeinsame Teiler berechnet und ausgegeben werden. Die Inhalte v. A & B dürfen sich ändern.

1. Lies die Ganzzahlen ein
2. Solange der Wert von A größer als 0 ist, wiederhole Schritte 2.a und 2.b
    a. Wenn Wert von A kleiner als B ist, dann vertausche A und B
    b. Weise A die Differenz von A minus B zu
3. Gib den Wert von B als größten gemeinsamen Teiler aus

Heron-Verfahren

Dient zum Ziehen einer Quadratwurzel, wobei eine positive Zahl N eingelesen wird und die approximierte Wurzel eine Genauigkeit kleiner als ein bestimmter relativer Fehler sein.

1. Man fängt mit einer beliebigen Zahl X = (N / C) an, wobei C eine Konstante sei.

2. Falls N = X^2, dann ist X die genaue Wurze; Ende
3. Falls | ((N – X^2) / N) | ≤ ε dann ist X die approximierte Wurzel; Ende
4. Sonst:
    a. Berechne eine Hilfsgröße H = MAX((N/X), N)
    b. Definiere einen Näherungswert X = ((X+H) / 2)
    c. Springe zu Schritt 2 zurück.

Polyas Sieb

Das Polyas Sieb generiert Kubikzahlen in fünf Schritten:

1. Generiere natürliche Zahlen von 1 bis N
2. Streiche jede dritte Zahl heraus
3. Generiere die Summenfolge jeweils zweier übriggebliebener Zahlen
4. Streiche jede zweite Zahl heraus
5. Das Resultat sind die Kubikzahlen

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	2		4	5		7	8		10	11		13	14	
1	3		7	12		19	27		37	48		61	75	
1			7			19			37			61		
1			8			27			64			125		

```
def polyaSieb(n):
  init(n)
  for i in range(1, n+1):
    put(i)

  streichen(3)
  summenfolge()
  streichen(2)
  summenfolge()

  while(not isEmpty()):
    print(get())
```

```
def summenfolge():
  s = 0
  while(True):
    z = get()
    s = s + z
    put(s)
    z = front()
    if(z == 1):
      return
```

```
def streichen(n):
  i = 1
  while(True):
    z = get()
    if(i < n):
      put(z)
      i = i + 1
    else: i = 1
    z = front()
    if(z == 1 or isEmpty()):
      return
```

Softwarekrise

Mitte der 1960er Jahre überstiegen die Kosten für die Software erstmals jene der Hardware. Daher kam es zu den ersten großen gescheiterten Software-Projekten. Das Resultat vieler Software-Projekte war Spaghetticode. Damit ist Quellcode, der komplexe, verworrene und schlecht nachvollziehbare Kontrollstrukturen aufweist. Daher ist die Strukturierte Programmierung sein sehr wichtiger Teil!

Was ist strukturierte Programmierung

Sie verlangt im Grunde die Beschränkung auf drei Kontrollstrukturen:

- **Sequenz**: Hintereinander auszuführende Programmanweisungen
- **Verzweigung**: Auswahl
- **Schleifen**: Wiederholung

Die großen Hochsprachen sind allesamt strukturiert.

Prozedurale Programmierung:
Wird von oben nach unten gemäß den Kontrollstrukturen abgearbeitet. Dabei wird baumartig in überschaubare Teile zerlegt, die aufgerufen werden können und Variablen werden Datentypen zugewiesen

---

Aus der Prozeduralen Programmierung geben sich einige Vorteile. Codeeinsparung, Wiederverwendbarkeit und Prozedur als Abstraktionseinheit sind nur einige davon.

Sequenz

Ist zum Beispiel die Zuweisung. Dabei wird in einer Sequenz einer Variable ein Wert zugewiesen.

Schleifen

Es gibt verschiedene Typen:

Kopfgesteuerte (vorprüfende) Schleifen

z.B.: While

Bevor der Schleifenkörper durchlaufen wird, wird geprüft ob er durchlaufen werden soll.

Wird verwendet, wenn nicht vorhersehbar ist, wie oft die Schleife durchlaufen werden soll und ob sie überhaupt durchlaufen werden soll.

Fußgesteuerte (nachprüfende) Schleifen

z.B.: Do-While

Prüft erst nach Ablauf des Schleifenkörpers.

Ähnlich der Kopfgesteuerten, wenn nicht im Vorhinein bekannt, wie oft durchlaufen werden soll – wird aber definitiv einmal durchlaufen!

Zählschleife

z.B.: For

Kopfgesteuerte Schleife, mit Zahlenintervallen.

Wird verwendet, wenn ein ganzer Bereich – beginnend mit einem bestimmten Startwert – bis zu einem vorgegebenen Endwert mit fester Schrittweite durchlaufen werden soll.

Mengenschleifen

z.B.: Foreach

Sonderform der Zählschleife, wobei eine Menge durchlaufen wird.

Programmiersprachen

Weil Maschinen mit ihrer jeweiligen Maschinensprache nur eine begrenzte, kleine Menge an einfachen Maschinebefehlen können und diese schwer vom Menschen zu lesen sind, musste man höhere Programmiersprachen entwickeln, die bestimmte Paradigmen abstrahieren konnten. Diese werden dann in Maschinensprache übersetzt.

Höhere Sprachen

Fortran war erste, ca 1954 – COBOL war erste kommerzielle.
Mit PASCAL kam 1968 eine zur Ausbildung entwickelte, mit ADA konnte man sicherheitskritische Anwendungen programmieren und 1969 kam dann C, der Grundstein vieler weiterer Sprachen.
Mit C++ wurde 1980 erstmals Objektorientiert programmiert.

Generationen

	Sprachenart	Beispiel	Merkmale (Abstraktion von)
1G	Maschinen	Maschinen-Code	Binäre Befehle
2G	Assembler	Assembler-Code	Symbolische Befehle
3G	prozedural	FORTRAN, COBOL	Hardware-unabhängig
3G+	funktional objektorientiert	PASCAL, C, C++, Java, C#	Strukturiert (Daten=abstrakte Datentypen) Objektorientiert
4G	deklarativ	LISP,PROLOG,SQL	Transaktions-orientiert
5G	???		

Alles über der 3ten Generation ist eine höhere Programmiersprache.

Assemblersprache

Weil die Maschinensprache binär ist um vom Prozessor direkt ausgeführt werden zu können, und jede Rechnerarchitektur eine eigene Maschinensprache besitzt, entwarf man eine Assemblersprache, welche für spezifische Prozessorarchitektur den Maschinencode in einer lesbaren Form repräsentiert.

Daher hat jede Computerarchitektur auch eine eigene Assemblersprache, kann aber im Gegensatz zu höheren Sprachen enorme Geschwindigkeitsgewinne verbuchen, weil sie die nächste Sprache zum Maschinencode ist.

Kompilieren mit GCC

Um Assemblercode zu erhalten, kann man mit dem GCC-Compiler einfach

```
gcc -S file.c
```

schreiben und erhält eine file.s

Diese kann wiederrum mit dem Assemblieren zu einer Executable (Maschinencode) gewandelt werden:

```
as file.s -o file.o
```

Kompilieren / Interpretieren

Beim Kompilieren wird aus dem Quellcode in den Maschinencode übersetzt (über Assembler). Dann hat man zwei Dateien, wobei die Executable nur auf dem Zielrechner mit der Architektur, auf die Kompiliert wurde, ausgeführt werden kann.

Dabei können aus mehreren Quellcode-Dateien durch den Compiler mehrere Objekt-Dateien entstehen, die dann vom Linker zum Programm gelinkt werden.

Beim Interpretieren wird jedoch der „Quellcode“ jedes Mal neu interpretiert. Es wird also kein Maschinencode erzeugt, sondern in einer Umgebung zur Laufzeit direkt übersetzt. Z.B: LISP oder JavaScript.

Loader (Locater)

Damit kompilierte Programme dann auch als Befehlssatz im Computer geladen werden kann, muss erst die Adresse dieses Codes „lociert“ bzw „relociert“ werden.

Stack & Queue

Stack (Stapel)

Wird von den meisten Mikroprozessoren direkt in der Hardware unterstützt, dient zum Aufruf von Unterprogrammen (das heißt, ein aufgerufenen Programm liegt ganz oben am Stack, die aufrufenden darunter). Der Stack arbeitet folglich mit dem Last-In-First-Out (LIFO) Prinzip.

Es gibt drei wichtige Operationen:

- **Push**: Legt ein Element an oberster Stelle auf den Stack
- **Pop**: Entfernt das oberste Element aus dem Stack
- **Top**: Liefert das oberste Stack-Element, entfernt es aber nicht.

Weiteres sollten Implementierungen vorhanden sein, um zu prüfen, ob ein Stack leer / voll sei.

Polnische Notation

Die umgekehrte Polnische Notation basiert auf dem Prinzip des Stacks. Sie definiert eine Postfixschreibweise für mathematische Schreibweise. Postfix bedeutet, der Operator steht nach den Operanden:

Infix Ausdruck	Postfix Ausdruck
((4 + 5) * (3 – 1)) / 9	4 5 + 3 1 – * 9 /

Der Vorteil darin besteht, dass es keine Präzedenzen gibt.

Die Polnische Notation funktioniert nach dem Prinzip das Stacks, das heißt, wird ein Operator gelesen, werden beide obersten Stackelemente entfernt, mit dem Operator verknüpft und das Ergebnis wird wieder als oberstes Element auf dem Stack abgelegt.

```
def calcPostfix(p, n):
  init(n);
  for i in range(0, n):
    if(type(p[i]) is int):
      push(p[i])
    elif(p[i] in [ "+", "-", "*", "/“ ]):
      x = popTop()
      y = popTop()
      push(calc(y, x, p[i]))
  return popTop()
```

Transformation von Infix zu Postfix

```
def infix2Postfix(a, n):
    init(n);
    p = [None] * n;
    j = 0
    for i in range(0, n):
        if(type(a[i]) is int):
            p[j] = a[i]
            j = j + 1
        elif(a[i] in [ "+", "-", "*", "/" ]):
            push(a[i])
        elif(a[i] == ")"):
            p[j] = popTop()
            j = j + 1
    while(not isEmpty()):
        p[j] = popTop()
        j = j + 1
    return p, j
```

Queue (Warteschlange)

Ist das Komplementärstück zum Stack, arbeitet nach dem FIFO-Verfahren, also First in First Out. Findet einen sehr breite Anwendungszweck – Zur Interprozesskommunikation, in Routern, Datenübergabe zwischen asynchronen Prozessen, Kommunikation mit langsamen Geräten.

Hier gibt es lediglich zwei wichtige Operationen:

- **Put**: Fügt ein Element am Ende der Queue an.
- **Get**: Holt das Element am Anfang der Queue und löscht es.

Auch hier sollten Methoden zum Überprüfen auf Voll/Leer existieren. Die Queue ist leer, wenn der Anfang das Ende ist. **Front** bietet sich an, um das nächste Element zu „peeken“.

Weil man hier dann theoretisch eine zu hohe Komplexität benötigt (weil man irgendwie festhalten muss, wo Anfang, Ende ist) hat man sich den Ringpuffer, bzw. das zirkulare Array ausgedacht, mit dem eine Komplexität von O(1) erreichbar ist.

Rekursion

Als Rekursion bezeichnet man die Methodik, eine Funktion **durch sich selbst zu definieren**. Als Beispiel dient die Fakturierung:

$fact: \mathbb{N} \to \mathbb{N}$

$$fact(n) = \begin{cases} 1 & n = 1 \quad (Rekursionsanfang) \\ n \cdot fact(n-1) & n > 1 \quad (Rekursionsschritt) \end{cases}$$

Es sei auch erlaubt, mehrere Funktionen o. Datenstrukturen durch wechselseitige Verwendung voneinander zu definieren. Dann spricht man von einer **wechselseitigen Rekursion**.

Rekursive Datenstrukturen:

Formen der Rekursion

Lineare Rekursion	Verschachtelte Rekursion
Kaskadenförmige Rekursion	Wechselseitige Rekursion

Lineare Rekursion

Iterative Implementierung der Fakultätsberechnung

Iterativ bedeutet, man iteriert über alle Elemente, Indizes, etc. in einem bestimmten Intervall von Anfang bis Ende.

```
n = int(input("N = "));
p = 1;
for i in range(2, n+1):
    p = p * i;
print(p);
```

Linearrekursive Implementierung der Fakultät

Eine Funktion ist linearrekursiv, wenn in jedem Fall der rekursiven Definition **höchsten ein rekursiver Aufruf** vorkommt.

```
def fact_LR(n):
  if(n == 1 or n == 0):
    return 1
  else:
    return n * fact_LR(n-1)
```

Jede iterative Schleife lässt sich durch eine lineare Rekursion ersetzen!

```
while(Bedingung):
  Tut-Was();
```

→

```
def Schleife()
  if(Bedingung):
    Tut-Was();
    Schleife();
```

Dabei ist die linearrekursive Version meistens die bessere, wobei dies stark vom gewünschten Resultat abhängig ist:

- Iterative Version

```
def dec2Bin_I(n):
  while(n > 0):
    r = n % 2
    print(r)
    n = n // 2
```

- Linearrekursive Version

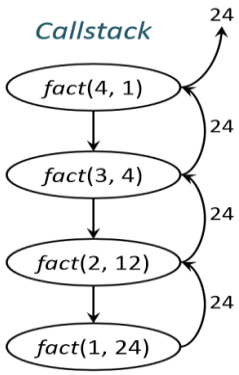
```
def dec2Bin_R(n):
  if(n < 2):
    print(n)
  else:
    dec2Bin_I(n//2)
    print(n % 2)
```

Hier ist die rekursive Version besser, weil sie die Quotienten auf dem Callstack ablegt und die Reste beim Zurückgeben in umgekehrter Reihenfolge ausgibt.

Endrekursive Implementierung der Fakultät

Eine Funktion ist endrekursiv, wenn der rekursive Funktionsaufruf die **letzte Aktion zur Berechnung** von f ist.

```
def fact_ER(n, f):
  if n == 1 or n == 0:
    return f;
  else:
    return fact_ER(n-1, f*n);
```



Verschachtelte Rekursion

Ist eine Rekursion, bei welcher rekursive Aufrufe in Parameterausdrücken rekursiver Aufrufe vorkommen. Als Beispiel dient die Ackermannfunktion:

$$Ack(n,m) = \begin{cases} m+1 & n = 0 \\ Ack(n-1,1) & m = 0 \\ Ack(n-1,\ Ack(n,m-1)) & m \neq 0 \wedge n \neq 0 \end{cases}$$

Die Ackermann-Funktion gilt als außerordentlich schwer zu durchschauen. Sie wächst extrem schnell und damit können in der theoretischen Informatik Grenzen aufgezeigt werden.

Kaskadenförmige Rekursion

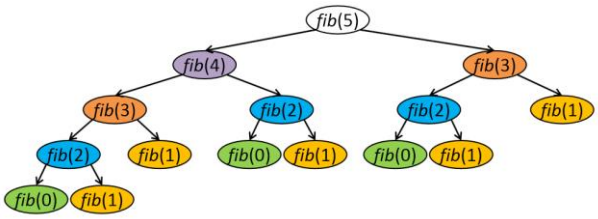
Die Form der Rekursion enthält mehrere rekursive Aufrufe, die nebeneinanderstehen und einen Baum bilden.

Sie gilt als äußerst elegant, kann aber einen höheren Berechnungsaufwand nach sich ziehen und ist daher mit Vorsicht zu genießen. Gerne wird sie als Ausgangspunkt für die Ableitung einer anderen effizienteren Formulierung gebraucht, selbst aber eher selten implementiert.

Als Beispiel dient die Fibonacci-Folge:

$$fib(N) = \begin{cases} 0 & N = 0 \\ 1 & N = 1 \\ fib(N-1) + fib(N-2) & N > 2 \end{cases}$$

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...



Wechselseitige Rekursion

Sie liegt vor, wenn in der **Definition einer Funktion f ein Aufruf von g** vorkommt **und** in der **Definition von g ein Aufruf von f**.

Beispiel: Even/Odd

```
def even(n):
  if n == 0:
    return True
  else:
    return odd(n-1)

def odd(n):
  if n == 0:
    return False
  else:
    return even(n-1)
```

Verkettete Listen

Weil ein Array den Nachteil hat, dass die Anzahl der Arrayelemente und der dafür nötige Speicherplatz schon im Voraus eingegeben werden muss, beim Entfernen eines Elements die anderen Elemente zusammengeschoben werden müssen und beim Einfügen eines Elements umgekehrt Platz gemacht werden muss, gibt es **verkettete Listen**.

Eine verkettete Liste ist eine dynamische Datenstruktur, die eine Speicherung von miteinander in Beziehung stehenden Daten erlaubt. Die Anzahl der Daten ist durch die Dynamik im Vorhinein nicht bekannt.
Die Liste wird durch in den Knoten enthaltenen Zeigern auf den jeweils nächsten

(bei doppelt verketteten Listen nächsten und vorherigen) Knoten realisiert. Daher kennt jedes Element seinen nächsten, bzw. sogar den vorherigen, Nachbarn.

Die Vorteile liegen klar auf der Hand, sie haben einen geringen Speicherverbrauch und sie sind dynamisch, das heißt wir können Elemente leicht einfügen oder entfernen. Dafür müssen wir aber auch einige Nachteile in Kauf nehmen:
Ein direkter Zugriff auf ein beliebiges Element ist nicht möglich und die Performanz ist auch nicht die Beste.

Einfach verkettete Liste

Der Kopf der Liste besteht, wie auch die anderen aus einem Dateninhalt und einem Zeiger auf das nächste Element. Das heißt, vom Kopf kommt man auf jedes Element, nicht aber von anderen zum Kopf.

Das letzte Element in der Liste ist das NIL („Not in List“) Element, welches durch NULL, None, etc. in den verschiedenen Sprachen realisiert ist. Darauf zeigt das letzte gültige Element.

```
class Node:
    def __init__(self, d):
        self.data = d
        self.next = None
```

E.data = Node(1)
F.data = Node(2)
G.data = Node(3)
H.data = Node(4)
E.next = *F*
F.next = *G*
G.next = *H*
H.next = **None**

Damit erhalten wir die Daten mit Node.data und können auf das nächste Element mit Node.next zugreifen.

Listenoperationen

Fünf Operationen sind für uns von wichtiger Bedeutung:

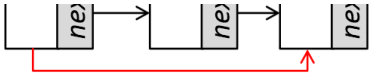
- Prepend: Fügt ein Element am Anfang ein
- Append: Fügt es hinten ein
- Search: Sucht ein bestimmtes Element in der Liste
- Insert: Fügt ein Element an einem beliebigen Punkt ein
- Delete: Löscht ein bestimmtes Element

```
def prepend(data, list):
    node = Node(data)
    node.next = list
    return node
```

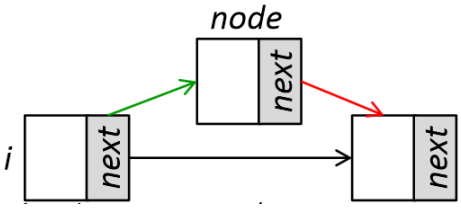
```
def append(data, list):
    node = Node(data)
    if list is None:
        node.next = None
        return node
    i = list
    while i.next is not None:
        i = i.next
    i.next = node
    node.next = None
    return list
```

```
def search(data, list):
    i = list
    while i is not None and i.data is not data:
        i = i.next
    return i
```

```
def delete(data, list):
    if list is None:
        return None
    if list.data is data:
        return list.next
    i = list
    while i.next is not None and i.next.data is not data:
        i = i.next
    if i.next is not None:
        i.next = i.next.next
    return list
```



```
def insert(data, list):
    node = Node(data)
    if list is None or node.data < list.data:
        node.next = list
        return node
    i = list
    while i.next is not None and i.next.data < node.data:
        i = i.next
    node.next = i.next
    i.next = node
    return list
```



Rekursive Algorithmen:

Betriebssysteme und Systemsoftware

Systemsoftware

Sie ist die Grundlage für Anwendungssoftware, die Gesamtheit aller Programme und Dateien, welche sämtliche Abläufe bei Betrieb eines Rechners steuern. Die allgemeine Software eines Systems, sozusagen. Sie stellt eine Verbindung zur Hardware her, steuert die Verwendung der Ressourcen – im Besonderen durch Anwendungssoftware – und verwaltet sowohl die internen, als auch die externen Hardwarekomponenten und kommuniziert mit diesen.

Klassifizierung von Systemsoftware

Zur Systemsoftware gehören grundsätzlich

Betriebssysteme

Systemnahe Software

Dienstprogramme
Datenbank Verwaltungswerkzeuge
Middleware

Obwohl Programmierwerkzeuge definitionsgemäß nicht zur Ausführung von Software benötigt werden, sind sie dennoch Bestandteil vieler Definitionen. Der Vollständigkeit halber schließen wir sie auch in unsere Definition ein. Programmierwerkzeuge wäre z.B. die Ermittlung der Performance, Debugging-Tools, IDEs...

Zudem wird der Begriff Systemnahe Software bei Systemen ohne Betriebssystem oft auch für die gesamte Software verwendet, z.B. bei Mikrocontrollern.

Betriebssysteme

Definition nach Deutschem Institut für Normung (DIN)

> *Unter Betriebssystem versteht man alle Programme eines digitalen Rechensystems, die zusammen mit den Eigenschaften der Rechenanlage die Basis der möglichen Betriebsarten des digitalen Rechnersystems bilden und insbesondere die Abwicklung von (Anwendungs-)Programmen steuern und überwachen.*

Im Allgemeinen kann man ein Betriebssystem als Schnittstelle zwischen Mensch und Hardware bezeichnen.

Ein Betriebssystem hat zwei **Bedienschnittstellen**:

Konsole, in welcher der Benutzer einen Befehl eingibt und entsprechende Antwort erhält.
Grafische Oberfläche, welche eine intuitive Erledigung von Aufgaben durch Schaltflächen, Menüs und Symbolen erlauben.

Außerdem gibt es eine **Programmierschnittstelle** (API, Application Programming Interface):

Sie sind ein elementarer Bestandteil eines Betriebssystems, um dem Programmierer leicht verständliche, gut handhabbare Schnittstellen zur eigentlichen Maschine anzubieten und die Komplexität der darunterliegenden

Maschine zu verstecken. Damit hebt sich das Betriebssystem selbst zum Ansprechpartner für den Programmierer, anstatt ihn mit der Maschine kommunizieren lassen zu müssen.

Weil UNIX viele Derivate hat, entwickelte die IEEE und Open Group das sogenannte **POSIX** (Portable Operating System Interface), welche als **standardisiertes API** die Schnittstelle neuerer UNIX-Betriebssysteme darstellt. Die API wurde in C geschrieben.

Aufbau und Dienste

Prozessmanagement

Programmen muss der Prozessor und auch alle anderen Ressourcen irgendwie zugeteilt werden – genau das übernimmt das Betriebssystem. So werden einzelne Aufgaben als Prozesse ausgeführt, die vom Betriebssystem verwaltet werden (können).

Speichermanagement

Weil nicht immer alle Programme/Daten in den Speicherplatz passen, sorgt das Betriebssystem auch dafür, dass immer nur die gerade benötigten Speicherinhalte zur Verfügung stehen, ohne dass sich die Anwendungssoftware darum kümmern muss.

Dateiverwaltung

Die Logik der Dateiverwaltung wird in Form von Dateisystemen vom Betriebssystem bereitgestellt, damit alle Programme auf dieselbe Art und Weise darauf zugreifen können.

Steuerung und Abstraktion der Hardware, Geräteverwaltung, E/A-Steuerung

Weil Computersysteme vielfältig (!) und modular von vielen verschiedenen Geräten unterschiedlicher Hersteller aufgebaut sind, müssten Anwendungsprogramme viel zu viele Modelle ansprechen können – das übernimmt das Betriebssystem mit Gerätetreibern.

Gerätetreiber sind gerätespezifische Programme, welche in das Betriebssystem eingebunden/installiert werden und die Steuerlogik für das entsprechende Hardware-Gerät enthalten. Damit kann das Betriebssystem die Zusammenarbeit mit vielen verschiedenen Ein-/Ausgabegeräten, wie Tastatur, Bildschirm, Datenträgern, Netzwerken, Druckern, etc. ermöglichen.

Bereitstellen der Benutzeroberfläche

Selbsterklärend

Weitere Informationen über Betriebssysteme (die zur Praktischen Informatik gehören) werden im 2. Semester, Datenbanksysteme von Prof. Thomas Fahringer besprochen.

Dienstprogramme

Wird oft auch System- oder Hilfsprogramm, im Englischen Utility genannt.

Führt für den Systemadministrator eines Computers (manchmal auch für den Benutzer) allgemeine, oft systemnahe Aufgaben durch, wie z.B.

- Konfigurieren der Hardware und des Betriebssystems

- Verwaltung von Benutzern
- Anzeige und Bearbeitung von Dateien (Hex-, Text-, Bildeditoren)
- Konvertierungen von Dateiformaten
- Kopieren, Sortieren, Sichern, Wiederherstellen v. Daten
- Einplanung und Ausführungssteuerung für Tasks der Stapel- oder Hintergrundverarbeitung
- Festplattenverwaltungsprogramm (Defragmentierung)
- Spooler für Druckaufträge
- Netzwerkverwaltungsprogramm
- Statistik- und Abrechnungsprogramme, zur Berechnung der Auslastung der Hardware, oder Benutzungsgebühren.

Datenbank Verwaltungswerkzeuge

Datenbanksysteme dienen der elektronischen Datenverwaltung, in welcher große Datenmengen effizient, widerspruchsfrei und dauerhaft gespeichert werden können. Benötigte Teilmengen werden in unterschiedlichen, bedarfsgerechten Darstellungsformen für Benutzer und Anwendungsprogramme bereitgestellt.

Die **Datenbank-Verwaltungswerkzeuge werden verwendet**, um Verbindungen zu den gewünschten Datenbanken aufzubauen, Tabellen darin abzufragen, die Daten darin grafisch und übersichtlich darzustellen – auch über die Datenbankstruktur Informationen anzuzeigen. Aber auch zum Anlegen und Ändern v. Datenbanken, Tabellen, Prozeduren, zur Autovervollständigung und Syntaxhervorhebung bei SQL-Statements, Import-Exportmöglichkeiten und beim Designen von „Entity-Relationships“ (Beziehungsdiagrammen).

Genauer Informationen über Datenbanksysteme (die zur Praktischen Informatik gehören) werden im 3. Semester, Datenbanksysteme von Prof. Günther Specht besprochen.

Middleware

Ist eine Diensteschicht oder Zwischenanwendung, welche die Komplexität und Infrastruktur einer Applikation verborgen lässt und damit zwischen Anwendung „vermittelt“.

So kann das Betriebssystem als eine Middleware zwischen Anwendungen und Hardware angesehen werden. In der Rechnerkommunikation hingegen unterstützt Middleware die Kommunikation zwischen Prozessen (im Gegensatz zu tieferen Netzwerkdiensten, welche einfache Kommunikation zwischen Rechnern handhaben).

BIOS

Basic Input / Output System, ist eine Systemsoftware, welche grundsätzlich beim Einschalten versucht, den ersten Programmbefehl von einer festgelegten Adresse aus dem Arbeitsspeicher zu laden.

Das BIOS beinhaltet die Basis-Steuerlogik für den Start eines Rechners, dazu gehört:

- Selbsttest
  - Wichtigsten Hardwarekomponenten (Grafikkarte, RAM, etc.) testen
  - Laufwerk suchen, von dem ein Betriebssystem gestartet werden kann

- Einfache Kommunikation mit der Hardware
  - Rechneruhr einstellen
  - Betriebssystem von CD laden
- Übergabe der Kontrolle an den Datenträger
  - Programm vom MBR (Master Boot Record, Startsektor) d. Laufwerks wird gestartet
  - Im MBR befindet sich ein Boot-Loader

Entwicklung der Systemsoftware

1. Generation: Röhren und Steckbretter (1940 - 1950)

Programme wurden direkt in Maschinensprache geschrieben und mussten komplett umgesteckt werden, um die Grundfunktion zu kontrollieren -> Sehr viel Kabelsalat. Programmiersprachen gab es nicht/waren unvorstellbar.

2. Generation: Transistoren und Stapelsysteme/Batchsysteme (1950 - 1960)

Mit der Lochkarte war es nun möglich, Programme auf eine Lochkarte zu schreiben und einzulesen. Damit waren schon leistungsfähigere Rechner möglich, man konnte zwischen Designern, Computerarchitekten, Operatoren, Programmierern und Wartungspersonal unterscheiden.

Die sogenannten Batchsysteme (Stapelsysteme) wurden auf Magnetbändern implementiert, um vergeudete Computerzeit des Hin- und Herlaufens des Operators zwischen Maschinenraum, Ein- und Ausgaberaums zu verringern (weil die Computer verdammt riesig waren!)

3. & 4. Generation: Betriebssystem (1960 - 1975)

OS/360 von IBM System/360 wurde veröffentlicht. Damit war erstmals eine Multiprogramming Technik möglich, indem der Speicher in mehrere Teile unterteilt wurde und während ein Job auf die Beendigung einer E/A-Operation wartet, konnte ein anderer Job die CPU nutzen.

Dies erfolgte über das sogenannte Spooling *(Simultaneous Peripheral Operation On Line)* realisiert:

Jobs wurden von Lochkarten auf Festplatte (in eine Warteschlange) geschrieben. Immer wenn ein Job beendet wurde, konnte das Betriebssystem sofort einen neuen Job von der Festplatte in den nun leeren Speicherbereich laden und starten.

Erstmals kam auch Timesharing ins Spiel:

Die CPU-Zeit wurde in Zeitscheiben eingeteilt. Jeder Anwender bekam mit seinem Terminal (mit Bildschirm und Tastatur), welches direkt mit dem Computer verbunden war für kurze Zeitspanne diese Zeitscheibe zugeteilt. Damit konnte er seine Jobs auf dieser Zeitscheibe durchführen lassen – kurze Jobs wurden meist in einer Zeitscheibe erledigt, zeitaufwändigere Jobs wurden vom Betriebssystem wieder aus der CPU entfernt und in die Warteschlange eingereiht.

5. Generation: Computernetze und Personal Computer (1975 - heute)

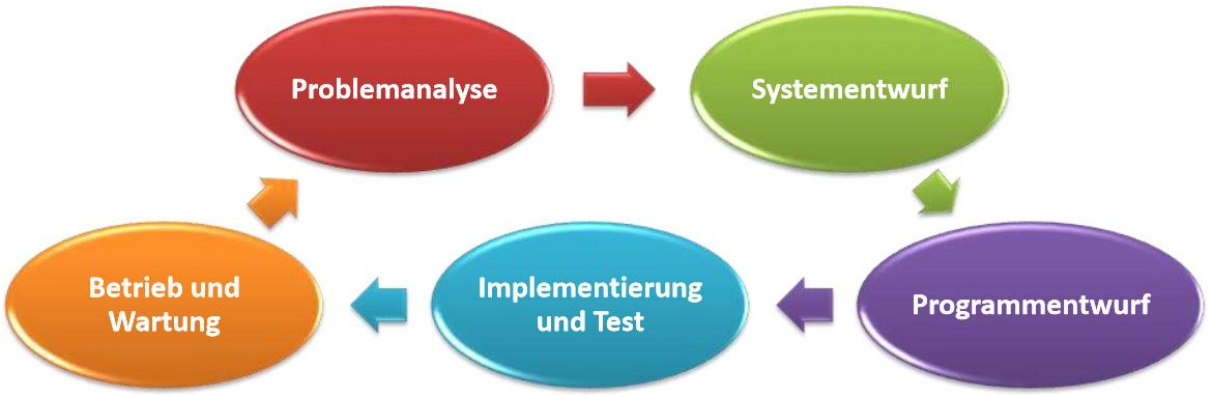
Mit MS-DOS wurde das erste standardisierte Betriebssystem für Personal Computer zur Verfügung gestellt. Es war noch Single-User.

Mit den Windows-Systemen als Nachfolger von MS-DOS wurde dann Multitasking möglich – Multiuser nicht, das kam erst 1970er Jahren mit UNIX, welches durch AT&T entwickelt wurde. **1990** wurde die erste frei erhältliche Version von Unix verfügbar: **Linux**.

Es gibt aber (vorallem bei eingebetteten Systemen) unglaublich viele Betriebssysteme mehr!

Systementwurf

Gehört thematisch nicht korrekterweise in den Bereich Systemsoftware, stellt allerdings für die Anwendungssoftware, welche auf dem Gebiet der Systemsoftware aufbaut, eine wichtige Rolle. Weils sonst nirgends reingepasst hat, kommt‘s halt hier her.



Problemanalyse

Wird in enger Zusammenarbeit mit dem Auftraggeber durchgeführt, das Ergebnis ist ein Pflichtenheft. Wird auch Anforderungs- oder Systemanalyse genannt.

Systementwurf

Die zu lösenden Aufgaben werden in Module unterteilt, um eine höhere Übersichtlichkeit und verbesserte Korrektheit/Zuverlässigkeit zu garantieren. Dann können die Module von unterschiedlichen Arbeitsgruppen getrennt implementiert werden. Das Ergebnis ist eine Systemspezifikation, die als Grundlage für die Implementierung dient.

Programmentwurf

Die einzelnen Module werden weiter verfeinert, indem Algorithmen und Datenstrukturen aufgebaut werden, das Ergebnis ist eine oder mehrere Programmspezifikationen

Implementierung und Test

Hier werden die Module letztendlich programmiert und auf ihre Spezifikationen getestet. Wenn die Module zusammengesetzt wurden, erhält man das Programm

Betrieb und Wartung

Diese umfasst die Pflege der Software, Erweiterungen und Änderungen, Fehlerbehebung und führt unter Umständen zur Problemanalyse zurück, wodurch der Zyklus entsteht.

Laufzeitwerkzeuge zum Verifizieren/Testen

Profiler

Er profiliert das Programm und wird eingesetzt, um nach Performanz-Problemen zu suchen.

Debugger

Sucht nach Fehlern, indem man schrittweise den Code durchgeht. Dabei können die Befehle in ihrer Hochsprache, aber auch in Assembler- und Maschinensprache dargestellt werden. Folglich lassen sich auch die Prozessorregister und Inhalte des Arbeitsspeichers betrachten. Dazu müssen die Programme typischerweise mit Debugging-Option kompiliert werden.