Informatische Grundlagen für Chemie und Biologie

Ivo Hofacker
ivo@tbi.univie.ac.at

Institut für Theoretische Chemie
Universität Wien

October 16, 2015

Geschichtliches

700 BC	Abakus
300 BC	Euklidischer Algorithmus : Effiziente Bestimmung des **ggT**
500	Erfindung Dezimalsystem in Indien
800	Muhammed ibn Musa abu Djafar al-Chwarismi Einführung der Null zu den Arabischen Zahlen ”Liber Algorithmi” - Rechenverfahren mit Dezimalzahlen für Kaufleute und Testamentsvollstrecker Aus seinem Namen und griechischem ”arithmos” (Zahl) stammt der Begriff **Algorithmus**
1202	Leonardo v. Pisa (Fibonacci) ”Liber abbaci” - Einführung in dezimales Rechnen
1574	Adam Riese Rechenbuch mit mathematischen Algorithmen
1613	Erste Logarithmentafeln werden algorithmisch berechnet
1615	Nepersche Rechenstäbe zum Multiplizieren
1623	Rechenmaschine von Schickard Addition, Subtraktion mit 6 stelligen Zahlen

Geschichtliches

1641	Blaise Pascal Addiermaschine mit 8 Stellen
1674	Gottfried Wilhelm Leibniz Rechenmaschine für die 4 Grundrechenarten
1774	Philipp Matthäus Hahn Erste zuverlässig arbeitende mechanische Rechenmaschine
1805	Joseph Marie Jacquard Webstuhl mit fester Programmsteuerung
1823	Charles Babbage "Difference engine"
1886	Herman Hollerith Erfindung der Lochkarte
1934	Konrad Zuse programmierbarer Relaisrechner auf Basis des Dualsystems
1936	**Church-Turing These** (zur Berechenbarkeit) *"Die Klasse der turing-berechenbaren Funktionen stimmt mit der Klasse der intuitiv berechenbaren Funktionen überein."*
1941	Konrad Zuse Elektromechanischer programmgesteuerter Rechner Z3

Geschichtliches

1944	Howard Aiken *Mark I*, sequentieller elektromech. Rechner
1946	Mauchly/Eckert *ENIAC*, Röhrentechnologie
1949	Edsac von Wilkes Erster Rechner mit gespeichertem Programm
1954	Rechner der 1. Generation (IBM 650, USA)
1957	Fortran (formula translator) auf Rechnern der 2. Generation (Transistoren, Ferritspeicher)
1960	COBOL, ALGOL
1962	BASIC
1971	Rechner der 3. Generation Integrierte Schaltungen, Mikroprozessoren, PASCAL
1973	Programmiersprache C
1980	Modula, Ada, PCs, Rechnernetze, Workstations
1986	Laptops, Parallel-Rechner
1990	optische Platten, Internet, World WideWeb

Teilgebiete der Informatik

Theoretische Informatik

- Berechenbarkeit und Komplexität
- Automaten und Formale Sprachen

Praktische Informatik

- **Algorithmen und Datenstrukturen**
- Programmiermethoden und -sprachen
- Betriebssystem und Compilerbau
- Datenbanken

Angewandte Informatik

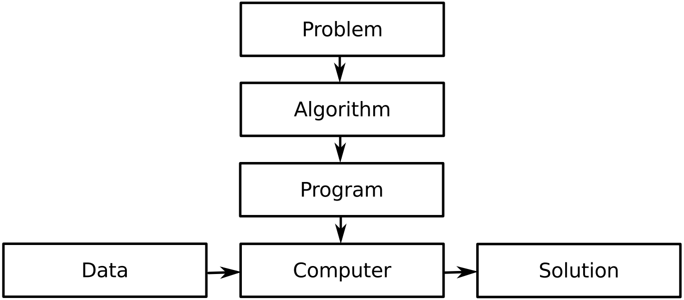
- Informationssysteme und KI
- Modellierung und Simulation
- Anwendungen in Wirtschaft, Wissenschaft, etc.

Technische Informatik

- Rechnernetze und -organisation
- Prozessoren und andere Schaltwerke
- Mikroprogrammierung

Algorithmen und Datenstrukturen

Lösen von Problemen mit Hilfe eines Computers



Was ist ein Algorithmus

"Berechnungsvorschrift, die genau definiert, wie aus Eingabedaten Ausgabedaten durch systematische Anwendung von Elementaroperationen in endlich vielen Schritten ermittelt werden."

Folgende Bedingungen müssen erfüllt werden:

- Korrektheit
- Allgemeingültigkeit (Abstrahierung, löst eine Problemklasse)
- Ausführbarkeit (Jeder Schritt muss durch eine Rechenmaschine ausführbar sein)
- Determiniertheit (gleiche Eingabe $\rightarrow$ gleiches Ergebnis)
- Finitheit (statisch und dynamisch)
- Terminiertheit (endlich viele Schritte)

Ein Algorithmus sollte verständlich und effizient sein!

Was ist ein Algorithmus

Beschreibungssprache:

- Natürliche Sprache
  zur Kommunikation von Ideen oft ausreichend
- Diagramme
  Darstellung des Kontrollflusses durch Ablaufpläne, Struktogramme, etc.
- Pseudo Code
  Mischung aus Konstrukten einer Programmiersprache und Prosa
- Programmiersprache
  Präzise, Garantiert die Ausführbarkeit auf einer Maschine

Was ist ein Algorithmus

Berechnungenbarkeit

"Eine Funktion ist berechenbar wenn es einen Algorithmus gibt, der sie berechnet!"

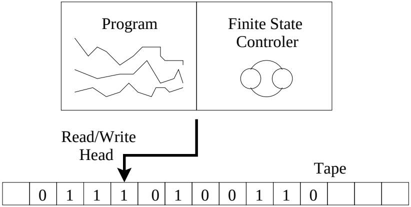
Ansätze zur Präzisierung des Berechenbarkeitsbegriffs:

- **Turing-Berechenbarkeit**
- **While-Berechenbarkeit**
- **Goto-Berechenbarkeit**
- ...

Diese Präzisierungen beschreiben die selbe Klasse an Funktionen

$\rightarrow$ Churchsche These:
"Die Klasse der turing-berechenbaren Funktionen stimmt mit der Klasse der intuitiv berechenbaren Funktionen überein."

Die Turing Maschine



- Die Turing Maschine hat eine endliche Anzahl von **Zuständen** $q_s, q_h, q_1, \ldots, q_n$.
- Das **Programm** ist eine Tabelle mit Anweisungen der Form $\langle q_i, s_j, s_k, m, q_l \rangle$.
- Ein **Band**, das als Speicher sowie als Inpu/Output fungiert.
- Ein **Lese-Schreib-Kopf**, der die aktuelle Position des Bandes lesen und schreiben kann.

Programmausführung auf einer Turing Maschine

1. Lese das Symbol and der aktuellen Position des Bandes
2. Ändere den Zustand der Maschine von q_i to q_l.
3. Schreibe das Symbol s_j.
4. Bewege den Lese-Schreib Kopf in Richtung m.

$\langle q_1, 1, 0, R, q_2 \rangle$
$\langle q_1, 0, 0, R, q_1 \rangle$
$\langle q_2, 1, 1, R, q_2 \rangle$
$\langle q_2, 0, 0, R, q_3 \rangle$
$\langle q_3, 1, 1, R, q_3 \rangle$
$\langle q_3, 0, 1, L, q_4 \rangle$
$\langle q_4, 1, 1, L, q_4 \rangle$
$\langle q_4, 0, 0, L, q_5 \rangle$
$\langle q_5, 1, 1, L, q_5 \rangle$
$\langle q_5, 0, -, -, halt \rangle$

While-Berechnungenbarkeit

- Variablen: $x_0, x_1, x_2, \ldots$, Konstanten: $0, 1, 2, \ldots$
- Operatoren: $+ -$, Trennsymbole ; $\leftarrow$
- Schlüsselwörter: WHILE DO END

Syntax:

1. Eine Wertzuweisung $x_i \leftarrow x_j \pm c$ ist ein While-Programm
2. Wenn P_1 und P_2 While-Programme sind dann auch $P_1 ; P_2$
3. Falls P ein While-Programm ist dann auch WHILE $x_i \neq 0$ DO P END
4. Nur durch 1.-3. beschriebene Programme sind While-Programme

Semantik:

1. Wertzuweisung : x_i wird der Wert von $x_j \pm c$ zugewiesen
2. Sequenz: Zuerst wird P_1, dann P_2 ausgeführt
3. Schleife: P wird solange ausgefürt bis $x_i = 0$ gilt. Test erfolgt vor Ausführung von P

While-Berechnungenbarkeit

Beispiel:

(Multiplikation mit Eingabe x_1, x_2 und Ausgabe x_0)

WHILE $x_1 \neq 0$ DO
  $x_3 \leftarrow x_2$
  WHILE $x_3 \neq 0$ DO
    $x_0 \leftarrow x_0 + 1;$
    $x_3 \leftarrow x_3 - 1$
  END;
  $x_1 \leftarrow x_1 - 1$
END

Abkürzungen:

IF $x \neq 0$ THEN P_1 ELSE P_2 END

an Stelle von

$x_1 \leftarrow 1; x_2 \leftarrow x;$
WHILE $x_2 \neq 0$ DO $P_1; x_2 \leftarrow 0; x_1 \leftarrow 0$ END;
WHILE $x_1 \neq 0$ DO $P_2; x_1 \leftarrow 0$ END

Goto-Berechnungenbarkeit

Sequenzen von Markern M_i und Anweisungen A_i der Form
$M_1 : A_1; M_2 : A_2; \ldots; M_k : A_k$

Anweisungen

- Wertzuweisung: $x_i \leftarrow x_j \pm c$
- unbedingter Sprung: GOTO M_i (damit wird A_i als nächstes ausgeführt)
- bedingter Sprung: IF $x_i = c$ THEN GOTO M_i
- Stopanweisung: HALT

While-Schleifen lassen mit Goto's simulieren:

WHILE $x_i \neq 0$ DO P END wird zu

M_1: IF $x_i = 0$ THEN GOTO M_2;
    P;
    GOTO M_1;
M_2: $\ldots$

GOTO-Programme sind schwer verstehbar und damit kaum verifizierbar.

Ein Nicht-Algorithmus

Wir machen Eierspeis

1. Schlage die Eier in eine Schüssel
2. Fuege Salz und Pfeffer geriebenen Käse und etwas Milch hinzu und rüre gut durch
3. Schneide Schinken in Streifen oder kleine Quadrate und brate ihn in einer Pfanne mit etwas Öl an
4. Gib den Inhalt der Schüssel in die Pfanne
5. Schieb die stockende Masse solange in der Pfanne umher bis sie die richtige Konsistenz hat
6. Nimm die Eierspeis vom Herd und bestreue sie mit Paprikapulver und Petersilie oder Schnittlauch

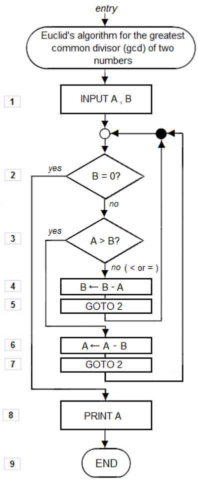
Euklidischer Algorithmus

Finde den grössten gemeinsamen Teiler zweier ganzer Zahlen m, n

1. Wenn m kleiner als n, vertausche Werte von m und n
2. Dividiere m durch n, nenne den Rest r
3. Wenn r gleich 0 ist, so gib n aus und terminiere
4. Wenn r nicht gleich 0 ist, so weise m den Wert von n zu und n den von r
5. Gehe zu 2.

Euklidischer Algorithmus

Finde den grössten gemeinsamen Teiler zweier ganzer Zahlen *m*, *n*



Euklidischer Algorithmus

Algorithm 1 Finde den grössten gemeinsamen Teiler gcd(n,m) zweier ganzer Zahlen m, n

```
if m = 0 then
    return n
end if
while m ≠ 0 do
    if n > m then
        n ← n − m
    else
        m ← m − n
    end if
end while
return n
```

(while-Schleife und Zuhilfenahme von primitiven Operationen)

Euklidischer Algorithmus

Algorithm 2 Finde den grössten gemeinsamen Teiler gcd(n,m) zweier ganzer Zahlen m, n

```
while m ≠ 0 do
  if n < m then
    t ← m
    m ← n
    n ← t
  end if
  t ← m
  m ← n mod m
  n ← t
end while
return n
```

(while-Schleife und Zuhilfenahme von komplizierten Operationen)

Euklidischer Algorithmus

Algorithm 3 Finde den grössten gemeinsamen Teiler gcd(n,m) zweier ganzer Zahlen m, n

```
if n < m then
    swap n, m
end if
if m = 0 then
    return n
else
    return gcd(m, n mod m)
end if
```

(Rekursion und Zuhilfenahme von komplizierten Operationen)

Euklidischer Algorithmus

Nochmal in Kurzfassung:

Solange $m > 0$:
    Wenn $n > m$, Vertausche n und m.
    Subtrahiere n von m.
n ist der Größte Gemeinsame Teiler

Zu Zeigen ist:

- **Korrektheit**
- **Termination**

- **Komplexität**

Euklidischer Algorithmus

Korrektheit
Gegeben $m > 0$ und $n > 0$, sei $g = gcd(n, m)$

1. Wenn $m = n$ dann $m = n = gcd(n)$, m wird auf 0 gesetzt und n zurückgeliefert, eindeutig korrekt.
2. Wenn $m > n$, dann $m = p \cdot g$ und $n = q \cdot g$ wobei p und q teilerfremd sind. Behauptung: $gcd(m - n, n) = g$, und damit $m - n = p \cdot g - q \cdot g = (p - q) \cdot g$.
   Es muss also gelten, dass $(p - q)$ und q teilerfremd sind. Wenn **nicht**, dann $p - q = a \cdot c$ und $q = b \cdot c$ für $a, b, c > 1$. Daraus folgt aber $p = q + a \cdot c = b \cdot c + a \cdot c = (a + b) \cdot c$.
   Weil $q = b \cdot c$ können damit p und q nicht teilerfremd sein, **Kontradiktion**!
   Also sind $(p - q)$ und q teilerfremd und $gcd(m - n, n) = gcd(m, n)$.
3. Wenn $n < m$ werden n und m vertauscht, so dass $m \geq n$. Dieser Fall ist bereits abgehandelt.

Euklidischer Algorithmus

Terminierung
Beim Start jedes Schleifendurchlaufs (Iteration) ist entweder $n > m$ oder $m \geq n$.

1. Wenn $m \geq n$ wird m durch $m - n$ ersetzt. Damit ist m kleiner als zuvor und nicht negativ
2. Wenn $n > m$ werden m und n vertauscht und im nächsten Schritt gilt wieder 1.

Damit wird das Maximum $\max(m, n)$ in jedem Schritt entweder kleiner oder bleibt für eine Iteration gleich.
Das widerum kann aber nicht für immer so weiter gehen, da m und n ganze Zahlen sind und irgendwann ein unteres Ende erreicht werden muss, nämlich genau dann wenn $m = 0$ und $n = g$.
Also terminiert der Algorithmus.

Komplexität

Es gilt festzustellen wie effizient ein bestimmter Algorithmus ist

Effizienzmaße:

- Speicherbedarf
- Laufzeit
- weitere Problemabhängige Parameter, z.B. Anzahl an Vergleichsoperationen bei Suchverfahren oder Anzahl der Bewegung von Datensätzen

Analysestrategien:

- **experimentelle Bestimmung**
  schlecht geeignet da Abhängig von Programmiersprache, Compiler und Computer
- **asymptotische Analyse**
  Wie schnell steigt der Aufwand mit steigender Problemgrösse? Betrachtung des *Worst-case*, *Best-case* und *Average-case* Szenarios

Euklidischer Algorithmus

Komplexität

- **Zeit**
  Eine Iteration braucht konstante Zeit, ist aber abhängig davon ob $m > n$ oder nicht.
  Wenn $m = n$, dann gibt es nur eine Iteration (**best-case**)
  Wenn $n = 1$, dann erfolgen m Iterationen (**worst-case**)
  average-case ist schwierig zu analysieren
- **Speicher**
  Der Speicherbedarf ist konstant, da nur drei ganze Zahlen gespeichert werden müssen.

Komplexität

Beispiel zur asymptotischen Analyse (Mittelwert über n Zahlen in Liste L:

	Kosten	Wie oft?
initialize list L with values	c_1	1
initialize variable *sum* with 0	c_2	1
initialize variable *length* with 0	c_3	1
for each value v of List L	c_4	n
do		
add v to *sum*	c_5	n
add 1 to *length*	c_6	n
end		
set mean equal to *sum* divided by *length*	c_7	1

Laufzeitfunktion $T(n)$:

$$T(n) = c_1 + c_2 + c_3 + n \cdot c_4 + n \cdot c_5 + n \cdot c_6 + c_7$$

Aymptotisch steigt die Laufzeit linear, der Algorithmus ist $\mathcal{O}(n)$

Komplexität

Die Landau Symbole: Θ-, $\mathcal{O}$- und Ω-Notation

Sei $g(n)$ eine Schrankenfunktion, dann ist eine

- *Obere* Schranke
  $\mathcal{O}(g(n)) =$
  $\{f(n) | (\exists c, n_0 > 0), (\forall n \geq n_0) : 0 \leq f(n) \leq cg(n)\}$
- *Untere* Schranke
  $\Omega(g(n)) =$
  $\{f(n) | (\exists c, n_0 > 0), (\forall n \geq n_0) : 0 \leq cg(n) \leq f(n)\}$
- *Genaue* Wachstumsrate
  $\Theta(g(n)) =$
  $\{f(n) | (\exists c_1, c_2, n_0 > 0), (\forall n \geq n_0) : 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)\}$

Beispiel: $\mathcal{O}(n^2)$ ist die "Menge aller Funktionen, die nicht schneller als quadratisch wachsen".

Komplexität

Regeln für Θ-, $\mathcal{O}$- und Ω-Notationen

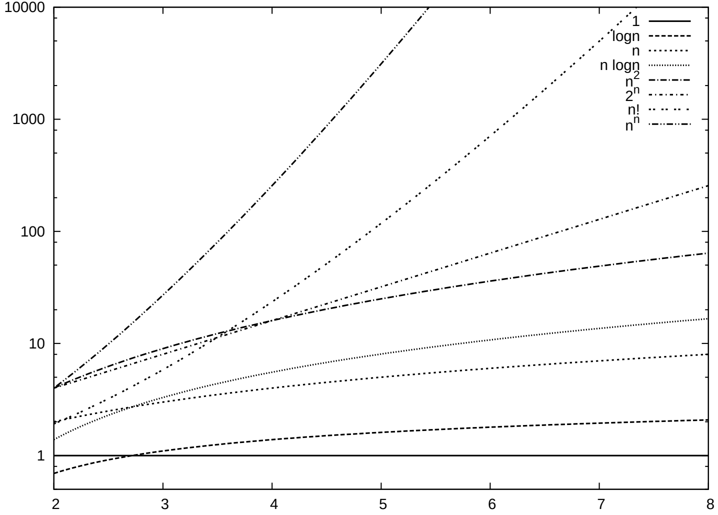
1. Ignoriere konstante Faktoren
   $\mathcal{O}(c \cdot f(n)) = \mathcal{O}(f(n))$
   z.B. $\mathcal{O}(20 \cdot n^3) = \mathcal{O}(n^3)$
2. Ignoriere kleine Terme
   Wenn $a < b$ dann $\mathcal{O}(a + b) = \mathcal{O}(b)$
   z.B. $\mathcal{O}(n^2 + n) = \mathcal{O}(n^2)$
3. Für *obere* Schranke gilt:
   Wenn $a < b$ dann ist ein $\mathcal{O}(a)$- auch ein $\mathcal{O}(b)$-Algorithmus
   z.B. Ein $\mathcal{O}(n)$- ist auch ein $\mathcal{O}(n^2)$-Algorithmus
   Aber nicht anders herum!
4. Ignoriere addition von konstanten Termen
   Da n und $\log n$ größer als jede Konstante k sind, gilt z.B.
   $\mathcal{O}(\log n + k) = \mathcal{O}(\log n)$
5. Vereinfachung durch geschachtelte Anwendung der oben genannten Regeln
   z.B. ist ein $\mathcal{O}(n \cdot \log n + n)$-Algorithmus ein $\mathcal{O}(n \cdot (\log n + 1))$-Algorithmus, also ein $\mathcal{O}(n \cdot \log n)$-Algorithmus

Komplexität

Wichtige Schrankenfunktionen

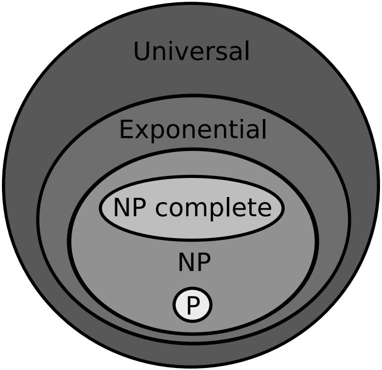
Notation	Name	Beispiel
$\mathcal{O}(1)$	konstant	Berechnen ob eine Zahl gerade oder ungerade ist
$\mathcal{O}(\log n)$	logarithmisch	Eine Zahl in einer sortierten Liste mittels Binärer Suche suchen
$\mathcal{O}(n)$	linear	Eine Zahl in einer unsortierten Liste suchen
$\mathcal{O}(n \log n)$	loglinear	Eine Liste mittels Quicksort sortieren
$\mathcal{O}(n^2)$	quadratisch	Eine Liste mittels Insertionsort sortieren
$\mathcal{O}(10^n)$	exponentiell	Travelling Salesman Problem (Dynamic Programming)
$\mathcal{O}(n!)$	faktoriell	Travelling Salesman Problem (Brute Force)

Komplexität



Komplexitätsklassen

- Messung der Komplexität am Ressourcenverbrauch (Laufzeit, Speicherbedarf)
- Klassifizierung aller lösbaren Probleme (Abgrenzung von effizient lösbaren von schwierigen Problemen)



Komplexitätsklassen

- Für Probleme der Klasse **P** existieren effiziente Algorithmen, die das Problem in polynomialer Laufzeit lösen.
  Es genügt also zu zeigen, dass deren Komplexität $\mathcal{O}(n^k)$ für eine Konstante k ist.
- Für Probleme der Klasse **NP** lässt sich die Richtigkeit einer Lösung in polynomialer Laufzeit prüfen.
- Probleme, die nicht in **P** sind, gelten als hart. Für praktische Zwecke sind sie oft nicht lösbar, da der Aufwand schneller als polynomial ansteigt. Viele interessante Probleme liegen in **NP**.
  Es gibt jedoch *nichtdeterministische Algorithmen* die diese Probleme in polynomialem Zeitaufwand lösen.
- **NP-schwer (NP-hard)**: Ein Entscheidungsproblem L heißt NP-schwer genau dann, wenn alle Probleme aus der Klasse NP polynomiell auf L reduzierbar sind.
- **NP-vollständig (NP-complete)**: Ein Entscheidungsproblem L heißt NP-vollständig genau dann, wenn L in der Klasse NP liegt und L NP-schwer ist.

Komplexitätsklassen

NP vollständige Probleme

- Alle gleich schwierig da sie aufeinander reduzierbar sind
- Sollte für eines ein Polynomialzeit Algorithmus gefunden werden, wären automatisch alle NP Probleme in Polynomalzeit lösbar
In diesem Fall wäre $P = NP$.
- Da kein polynomialer Algorithmen für NP-vollständige Probleme bekannt ist gilt wahrscheinlich $P \neq NP$.
Allerdings ist auch noch kein Beweis für $P \neq NP$ gelungen.

Beispiele

- **Travelling Salesman**: Ein Vertreter, dessen Bezirk eine bestimmte Anzahl von Städten umfaßt, beginnt seine Reise stets von seiner Basis aus, besucht jede Stadt genau einmal und kehrt dann zur Basis zurück. Um die Reisekosten zu minimieren, hat er eine Tabelle der gegenseitigem Entfernungen der Städte seines Bezirks zusammengestellt.
- **Rucksackproblem**: Gegeben sind eine Anzahl von Waren verschiedener Größe und von unterschiedlichem Wert. Man fülle den Rucksack so, daß man Waren von maximalem Gesamtwert transportiert.

Sortieren

- Sortieren ist eines der häufigsten Probleme
- Vielzahl von Algorithmen
- Erlaubt schnelle Suche in sortierten Daten

Problemstellung:
Gegeben: Datensatz $s_1, s_2, \ldots, s_n$ mit Schlüsseln $k_1, k_2, \ldots, k_n$, und eine Ordnungsrelation "$\leq$" auf den Schlüsseln.
Gesucht: Permutation

$$\pi : 1, 2, \ldots, n \to 1, 2, \ldots, n$$

die die Schlüssel in Aufsteigende Reihenfolge bringt:

$$k_{\pi(1)} \leq k_{\pi(2)} \leq \ldots \leq k_{\pi(n)}$$

Der Einfachheit halber, nehmen wir im folgenden immer an das Listen von Zahlen sortiert werden.

Insertion-Sort

Idee: 2 Listen mit sortierten und unsortierten Zahlen; nehme eine Zahl der unsortierten Liste füge sie in sortierte Liste ein.

Algorithm 4 Insertion-Sort(A)

Input: List of numbers in array A (A[1], A[2], . . ., A[n])
Output: Sorted list in A

for $j = 2$ to n **do**
  $k = A[j]$
  $i = j - 1$
  while $i > 0$ and $A[i] > k$ **do**
    $A[i+1] = A[i]$
    $i = i - 1$
  end while
  $A[i+1] = k$
end for

$A[1] \dots A[j]$ is sorted, rest still unsorted

Selection-Sort

Idee: Finde kleinste Zahl, bringe sie an den Anfang der unsortierten Liste

---

Algorithm 5 Selektion-Sort(A)

Input: List of numbers in array A (A[1], A[2], . . ., A[n])
Output: Sorted list in A

for $j = 1$ to n **do**
  $minpos = j$
  for $i = j + 1$ to n **do**
    if $A[i] < A[minps]$ **then**
      $minpos = i$
    end if
    if $minpos > j$ **then**
      swap $A[minpos]$ and $A[j]$
    end if
  end for
end for

Bubble-Sort

Idee: Vertausche benachbarte Zahlen falls Reihenfolge falsch
Wiederhole bis Liste sortiert ist

Algorithm 6 Bubble-Sort(A)

Input: List of numbers in array A (A[1], A[2], . . ., A[n])
Output: Sorted list in A

for $i = 1$ to n **do**
  for $j = 1$ to $n - i$ **do**
    if $A[j] > A[j+1]$ **then**
      swap $A[j], A[j+1]$
    end if
  end for
end for

Rechenaufwand offensichtlich in $\mathcal{O}(n^2)$
Beschleunigung um 50% durch vorzeitigen Abbruch, falls keine Vertauschung passiert ist.