Algorithmen und Programme

Protokolliert von Rouven Czerwinski

Version vom 10. Januar 2012

Inhaltsverzeichnis

Tabellenverzeichnis

1 Einführung

- Kleinstcomputer (eingebettete Systeme) mit Alg. in allen Bereichen des täglichen Lebens: Taschenrechner, Handy, DvD-Player, MP3-Player, Waschmaschine, TV, Autos, Funkuhren...
- Programmierkenntnisse werden erwartet:
  - Programmierung und Steuerung komplexer Geräte und Maschinen
  - Erstellung interaktiver Medien (Internet, Videospiele, DVD, BluRay, E-Books...)
  - Verwaltung und Auswertung von Datenbanken

1.1 Algorithmusbegriff

Intuitiv: Alg. = Verarbeitungsvorschrift
Im Alltag z.B. Kochrezept, Spielregeln, Noten, Waschmaschinenprogramme, ...
Man spricht von einem Alg., wenn die Vorschrift präzise, eindeutig, vollständig und ausführbar ist.

Definition:

> Ein Alg. ist eine präzise formulierte Verarbeitungsvorschrift, die unter Verwendung elementarer Operationen einen Eingangszustand bzw. Einganswerte in einen Ausgangszustand bzw. Ausgangswerte überführt

Formal: Abbildung f: Eingabe $\rightarrow$ Ausgabe

Beispiele:

- Mathematische Formeln: $f : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$ z.B. Addition zweier Zahlen $f(q, p) = q + p$
- Primzahlentest: $f : \mathbb{N} \rightarrow \{ja, nein\}$

$$f(n) = \begin{cases} \text{ja, falls } n \text{ Primzahl} \\ \text{nein, sonst} \end{cases}$$

- Euklidischer Alg. ggT(x,y)

Alg. dienen zur Lösung von Problemen, sie werden als Programme so abgefasst, dass sie von einem Rechner ausgeführt werden können:

$$\underbrace{\text{Problem} \rightarrow \text{Algorithmus} \rightarrow \text{Progr}}\text{amme} \rightarrow \text{Maschine}$$

Gegenstand der Vorlesung

2 Algorithmische Grundkonzepte

2.1 Eigenschaften von Algorithmen

- Terminiertheit
  Ein Alg. terminiert, wenn er für jede Wahl von gültigen Eingabewerten nach endlich vielen Schritten anhält

- Determiniertheit
  Ein Alg. ist determiniert, wenn er bei gleicher Eingabe stets auf das gleiche Ergebnis führt.

- Determinismus
  Ein Alg. ist deterministisch wenn er bei gleicher Eingabe stets über die gleichen Zwischenergebnisse zum gleichen Ergebnis führt.

- Beispiel: Berechnung eines Terms
  - hält immer an $\Rightarrow$ terminiert
  - gleiches Ergebnis $\Rightarrow$ determiniert
    $\Rightarrow$ nicht deterministisch

2.2 Daten, Operanden und Operationen

Daten:

- Darstellung von Informationen im Rechner zur Eingabe, Verarbeitung und Ausgabe

- zB. Zahlen, Zeichen, Texte, Tabellen, Graphen, Bilder, ...

- Rechnerinterne Darstellung zB. (komprimiert vs. unkomprimiert)

Datentyp:

- Zusammenfassung von Wertebereich und darauf def. Operationen zu einer Einheit

- Beispiel: Standarddatentypen: int, float, char, ...

- Ein Alg. lässt sich auffassen als Anwenden von Operationenauf Objekte bestimmten Datentyps (=Operanden).

- Operand können Konstanten, Variablen oder Ausdrücke sein.

- Ausdrücke (Terme) entstehen indem Operanden mit Operationen verknüpft werden
- Datentypen legen die Wertemenge fest, aus der die Operanden Werte annehmen können

2.3 Standard-Datentypen

:
(In den meisten Programmiersprachen vorgegeben)

2.3.1 Integer:

(in der Programmiersprache C/C++: int)
C/C++ Beispiel: (Deklaration einer Variabln a vom Typ Integer)

```
int a; // Kommentar
```

Datentyp, Variablenname, Befehlsende
Wertemenge: Z = {. . . ,-1,0,1,2,...} (im realen Rechner nach oben und unten begrenzt)
Rechenoperationen: +, -, *, /, % (modulo), ++ (Inkrement), - -(Dekrement)
Zuweisungsoperator: =
Vergleichsoperatorn: <, >, == (gleich), != (ungleich), <= (kleiner gleich), >=, . . .
C/C++ Beispiel:

```
int a; // Deklaration der Variablen a
int b; // Deklaration der Variablen b
a = 5; // Wertezuweisung: a wird auf 5 gesetzt
b = a +8;
```

Variablen bestehen aus einem Namen (Referenz auf einen Speicherplatz) und einem Wert (Inhalt des Speicherplatzes)

```
int c;
c = b / a; // Division: c wird auf b/a also 2 gesetzt
```

Problem bei der Integer-Division:
Ergebnis wird ganzzahlig abgerundet: 13/5 ergibt 2,
Allg. zur Berechnung des Restes der ganzzahligen Division b/a:

```
int rest;
rest = b - (b/a)*a;
```

Der Rest kann in C/C++ auch durch den Modulo -Operator % beschrieben werden

```
rest = b % a;
```

2.3.2 Real:

(in C/C++: float und double)
Wertemenge: Q (I'm realen Rechner nur eine Teilmenge von Q)
Operatoren wie oben
C/C++ Beispiel

```
double c,d; // Deklaration der Var. c und d
c = 0,3; // c wird auf 0,3 gesetzt
d = (4,5 - c)*1,5; // d <- 6,3
```

2.3.3 Character:

(in C/C++: char)
Wertemenge zB. {'a', 'b',...,'A','B', ...,'1','2',...,'#', ...}

```
char e;
e = 'z';
char f = '#'
```

2.3.4 String:

(in C kein Standarddatentyp, statt dessen Array vom Typ char, in C++ std::string)
Wertemenge: Zeichenketten, zB. "Hallo", "Guten Tag", ...
C++ Beispiel:

```
char wort[6] = "Hallo"; //Deklaration als char-array
std::string s = "Guten Tag"; // Deklaration als std::string
```

2.3.5 Boolean:

(in C++ bool; in C kein Standarddatentyp, stattdessen int)
Wertemenge: {true, false} btw. {0,1}
Einstelliger: ! (not) Zweistellige Verknüpfungensoperatoren: && (and), || (or), == (gleich), != (ungleich), ...
C++ Beispiel

```
bool ergebnis,op1,op2;
op1 = true;
op2 = !op1; // op2 wird false
ergebnis = op1 && op2; // ergebnis wird false
Ergebnis = op1 || op2; // ergebnis wird true
```

Verknüpfungstabellen:

Tabelle 1: And Tabelle

op1	&& op2	Ergebnis
true	true	true
true	false	false
false	true	false
false	false	false

Tabelle 2: Or Tabelle

op1	|| op2	Ergebnis
true	true	true
true	false	true
false	true	true
false	false	false

Tabelle 3: Negation Tabelle

!op1	Ergebnis
true	false
false	true

Die Reihenfolge der Auswertung von Ausdrücken ergibt sich durch Klammerung und Vorrangregeln (Punkt vor Strich usw z.B. $(a * b + c * d/e)$)
Beispiel:

```
bool Ergebnis,op1,op2,op3;
Ergebnis = op1 || op2 && op3;
Ergebnis = op1 || (op2 && op3);
```

3 Imperative Algorithmen

- Basis für imperative Programmiersprachen wie C, Pascal, Modula, Basic, PHP, ...
- bekannteste/häufigste Art Alg. zu formulieren
- Alternative: Deklarative Programmierung z.B.
  funktionale Programmiersprachen wie z.B
  Lisp, Scheme, Haskell, ...

3.1 Ein- und Ausgabe

C++ Beispiel

```
int a;
cin >> a; //Eingabe des Wertes von a ueber Konsole
cout << "Sie haben" << a << "eingegeben" << ende; // Ausgabe
```

3.2 Verzweigungen (bedingte Anweisung)

Mit Verzweigungen können in Abhängigkeit einer Bedingung unterschiedliche Anweisungen ausgeführt werden.
Syntax:

```
if (BEDINGUNG) ANWEISUNG;
else ANWEISUNG2;
```

Interpretation:
Führe ANWEISUNG1 aus, falls die boolsche BEDINGUNG wahr (true) ist, ansonsten führe ANWEISUNG2 aus. (Der Else-Teil ist optional)
C++ Beispiel:

```
if (a<0) cout << "a ist negativ" << endl;
else cout << "a ist positiv" << endl;

bool b = ((a % 2) == 0); // b wird true, falls a modulo 2 null ist
if(b) cout << "a ist gerade" << endl;
```

Eine ANWEISUNG darf auch ein mit Klammern {} zusammengefasster Block von mehreren Anweisungen sein.

Syntax:

```
if (BEDINGUNG)
{
  Anweisung 1.1
  Anweisung 1.2
}
```

Tabelle 4: Modulo Tabelle

a	$a/2$	a%2	$a/3$	a%3
0	0	0	0	0
1	0	1	0	1
2	1	0	0	2

3.3 Schleifen (Wiederholungen)

Wiederholte Ausführung von Anweisungen in Abhängigkeit von einer Bedingung.
Syntax:

```
while (BEDINGUNG) ANWEISUNG;
```

Interpretation: Solang die boolsche BEDINGUNG wahr ist, wiederhole ANWEISUNG.
C++ Beispiel:

```
int a = -1;
while (a<0) cin >> a; // Wiederholte Eingabe von a, bis a postitiv ist
// Ausgabe aller Zahlen von 0 bis a:
int b = 0; // Startwert
while ( b <= a ) // Schleife laeuft solang b<=a
{
  cout << b << endl; // Ausgabe
  b = b + 1; // Hochzaehlen von b
}
```

Alternativ kann auch die for-Schleife benutzt werden.
Syntax:

```
for ( STARTANWEISUNG; BEDINGUNG; ZAEHLANWEISUNG ) ANWEISUNG;
```

Interpretation: Führe zu Beginn einmal die Startanweisung aus, solang die boolsche BEDINGUNG wahr ist, wiederhole erst ANWEISUNG dann ZAEHLANWEISUNG.

C++ Beispiel:

```
for ( int b = 0; b <= a; b = b + 1 ) // Ausgabe aller Zahlen von 0 bis a.
{
  cout << b << endl;
}
```

3.4 Prozeduren/Funktionen (Unterprogramme)

Zusammenfassung von mehreren Anweisungen zu einer Anweisung. Bestehend aus einem RÜCKGABETYP, dem FUNKTIONSNAMEN und einer Liste von Parametern.
Syntax:

```
RUECKGABETYP FUNKTIONSNAME ( PARAM1, PARAM2,... )
```

C++ Beispiel:

```
int max( int a, int b ) // Selbstdefinierte Funktion "max"
{
  if ( a > b ) return a; // gibt a zurueck
  else return b; // gibt b zurueck
}

--------------------

int m, n;
m = max( 2,5 ); // m wird auf 5 gesetzt
n = max( 8,m ); // n wird auf 8 gesetzt
```

Ganz Ähnlich ist die mathematische Definition:

$$max(a,b) - \begin{cases} \text{a falls } a > b \\ b \text{ sonst} \end{cases}$$

Funktionsaufrufe können verschachtelt werden

```
n = max( 8, max( 2,5 ) );
```

C++ Beispiel: Funktion zur Berechnung von $f(x) = x^2 + 2x + 1$

```
float f( float x )
{
  return x * x + 2*x - 1
}

--------------------

float y = f( 3.4 )
```

Wenn die Prozedur keine Rückgabe hat, ist der RUECKGABETYP = void

```
void SchreibWas() // Prozedur ohne RUECKGABE und ohne Parameter
{
  cout << "Test" << endl;
}
```

Der Einstiegspunkt eines Programms ist bei C/C++ auch eine Funktion mit Namen „main“

```
int main()
{
  int x,n;
  SchreibWas();
  cout << "Zahl eingeben";
  cin >> x;
  n = max( x,5);
  ...
  return 0; // Fehlercode
}
```

3.5 Strukturierung von Algorithmen/Programmen

Systematischer, übersichtlicher Aufbau; insbesondere bei größeren Programmen

- Zum Entwurf und zur Darstellung von Algorithmen: Struktogramme (nach Nassi-Shneiderman):
  Linearer Ablauf (Sequenz) Verzweigung (Alternative) und Wiederholung (Iteration)
  (Hier müssen noch mit zB. TikZ Bilder gemalt werden. . .)
- Alternative
  - Ablauf-/Flussdiagramme (Auch hier fehlt das Bild. . .)
- UML (Unified Modeling Language)
  Insbesondere objektorientiert

3.6 Beispiel: Euklidischer Algorithmus zur Berechnung des ggT

Problem: Entwickeln sie ein Programm zum Kürzen eines Bruches.

$$\frac{x}{y} = \frac{12}{24} = \frac{1}{2}$$

Teilproblem: Berechnung des größten gemeinsamen Teilers.
Algorithmus:
Gegeben: Zwei zahlen $x, y(element)\mathbb{N}$
Gesucht: ggT(x,y)
Vorgehensweise: von der jeweils größeren Zahl die kleinere Zahl solang subtrahieren, bis beide Zahlen gleich sind => Ergebnis.
(Muss noch als Bild realisiert werden)
Eingabe x, y
solang $x \neq y$

x>y
ja nein
x= x-y | y=y-x
Ausgabe x
Beispiel: ggT(12,20)

Tabelle 5: ggT Tabelle

	x	y
Start	12	20
nach Schritt 1	12	8
nach Schritt 2	4	8
nach Schritt 3	4	4

3.7 Parameterübertragung bei Prozeduren/Funktionen

- Werteaufruf (siehe bisherige Bsp.)
  Übergabeparameter sind lokale Größen, Sie werden beim Funktionsaufruf in lokale (nur für die Funktion sichtbare Variablen) kopiert.

- Referenzaufruf Speicherort der Größen wird übergeben, d.h. Referenz auf den Inhalt der Variablen.
  In C mit Hilfe von Zeigern realisiert. C++ Beispiel

```
int QuadratSumme(int x, int y)
{
  x = x*x;
  y = y*y;
  return x + y;
}

.....

int x = 5;
int y = 3;
int a = QuadratSumme(x,y); // a wird 34
cout << "x=" << x << " y=" << endl; // Ausgabe: x = 5, y = 3
```

  Der Wert von x und y ändert sich durch den Funktionsaufruf nicht.

```
int QuadratSumme(int &x, int &y)
{
  x = x*x;
  y = y*y;
  return x + y;
}
```

```
.....

int x = 5;
int y = 3;
int a = QuadratSumme(x,y); // a wird 34
cout << "x=" << x << " y=" << endl;
// Ausgabe: x = 25, y = 9, wegen der &-Zeichen!
```

Bei Ausführung mit Referenzaufruf sind die Werte von x und y verändert!

Beispiel: Funktion die den Inhalt zweier Variablen vertauscht.

```
// void SwapValues(int x, int y)
void SwapValues(int &x, int &y)
{
  int tmp = x;
  x = y;
  // y = x;
  y = tmp;
}

--------------

int x = 5;
int y = 3;
SwapValues(x,y);
```

3.8 Beispiel: Berechnung von Fibonacci-Zahlen

Jede Fibonacci-Zahl ist gleich der Summe der beiden vorhergehenden Fibonacci-Zahlen:

$$F_0 = 0, F1 = 1; \text{für } n > 1 : F_n = F_{n-1} + F_{n-2}$$

$$\text{Folge: } 0, 1, 1, 2, 3, 5, 6, 13, ...$$

C++ Beispiel:

```
int fib(int n) // Funktion zur Berechnung der n-ten Fibonacci-Zahl
{
  int n2 = 0;
  int n1 = 1;
  int Ergebnis = n1 +n2;

  if (n<=0) return 0;

  for (int i=2; i<=n, i=i+1)
  {
    Ergebnis = n1 +n2;
    n2 = n1;
    n1 = ergebnis;
  }
  return ergebnis;
}
```

Tabelle 6: fib(4)

fib(4)	i	n1	n2	ergebnis
Initial	2	1	0	1
Nach 1. Schleifendurchlauf	3	1	1	1
Nach 2. Schleifendurchlauf	4	2	1	2
Nach 3. Schleifendurchlauf	5	3	2	3

3.9 Rekursionen

Prozeduren können sich selbst direkt oder indirekt aufrufen.
Bsp: Fibonacci-Zahlen rekursiv: $F_0 = 0, F_1 = 1; \text{für } n > 1 : F_n = F_{n-1} + F_{n+2}$
C/C++:

```
int fib{int n}
{
  if (n<=0) return 0; // Abbruchbedingungen
  if (n==1) return 1; // Abbruchbedingungen
  return fib(n-1) + fib(n-2); // Rekursionsschritt
}
```

Ablaufbeispiel fit(4)
(An dieser Stelle sollte eines erstellt werden...)

- Beispiel ineffizienter als iterative Lösung(mit fortschleife), da identische Berechnungen mehrfach wiederholt werden müssen.
- Oft lassen sich Algorithmen durch Rekursion einfach und anschaulicher Formulieren.

3.10 Grundlegende statische Datenstrukturen

statisch = konstante Anzahl von Variablen bzw. konstante Größen

3.10.1 Array:

Vektor bzw. Feld aus Elementen desselben Datentyps in C/C++:

```
int a[100]; // Deklaration eines Arrays mit 100 Integer-Werten 0-99!
a[0] = 30;
a[99] = 40;
a[100] = 1; // Falsch!

char name[5] = "Karl"; // letztes Zeichen immer NULL-Zeichen '\0'
```

```
// alternativ:
char name[] = "Hans"; // Compiler erkennt die Laenge automatisch
name[3] = 'd'; // Aus "Hans" wird "Hand"
name = "Hans"; // FALSCH! Zuweisung an Feld nicht möglich!
```

Zweidimensionales Array: Matrix bzw Tabelle
Deklaration C/C++ z.B:

```
float m[M][N]; // M und N ganze konstante Zahlen > 0!
// Indizierung beginnt beu 0!
```

$$\begin{bmatrix} m[0][0] & m[0][1] & \dots & m[0][N-1] \\ m[1][0] & \dots & \dots & m[1][N-1] \\ \vdots & \vdots & \vdots & \vdots \\ m[M-1][0] & \dots & \dots & m[M-1][N-1] \end{bmatrix}$$

Im Speicher werden alle Zeilen hintereinander abgelegt:
`m[0][0]...m[0][N-1] | m[1][0]...m[1][N-1] | m[M-1][N-1]`
Wenn das Element `m[0][0]` an der Speicherposition s abgelegt ist, dann ist das Element `m[j][i]` an der Speicherpostion s + j ∗ N + i abgelegt
C/C++ Beispiel:

```
double m[5][12]; // 2d-Array mit 5 Zeilen und 12 Spalten
m[0][0] = 1.3; // setzt Element nach oben links
m[4][11] = 2.5; // setzt Element nach unten rechts
for (int y=0; y<5; j=j+1)
{
  for (int i=0; i<12; i=i+1)
  {
    m[j][i] = j * 12 + i; // Alle Elemente mit fortlauf. Nummern füllen
  }
}
```

Arrays mit mehr als zwei Dimensionen entsprechend.
z.B. `float m[`N_1`][`N_2`]...[`N_k`]`

Zugriffsbeispiel:

```
for (int i=0; i<100; i=i+1)
{
  a[i] = i*i;
}
```

3.10.2 Struktur (structure/record)

Zusammenfassung von Elementen unterschiedlicher Typen
C/C++ Beispiel:

```
#include <string.h> // für den Befehl strcpy(...)
struct Person
{
  char Vorname[20];
  char nachname[28];
```

Tabelle 7: Array Access

Index	Inhalt
0	0
2	4
...	...
99	9801

```
  int alter;
  float groesse;
  ...
};

int main()
{
  person k; // deklariert eine Variable k von Typ Person
  k.alter = 29;
  k.groesse = 1.65;
  strcpy(k.vorname, "Markus");
  strcpy(k.nachname, "Mueller");
}
```

Für die Variable k (vom typ Person) reservierter Speicher: C/C++ Beispiel:

Tabelle 8: struct Speicherreservierung

Adresse	Inhalt
30 001	'M'
30 002	'a'
...	...
30 021	'M'
...	'u'
...	'e'
30049	(Beginn des int32 mit Nullen gefüllt) 0001 1101

```
...
Person pt[100]; // Array mit 100 Personen
...
pt[37].alter = 29;
strcpy(pt[37].nachname, "Beyer");
pt[37].nachname [0] = 'M' // Aus " Beyer" wird "Meyer"
...
```

- Geeignet zur Organisation/Strukturierung von Daten
  $\Rightarrow$ übersichtliche und effiziente Algorithmen

4 Komplexität von Algorithmen

4.1 Einführung

Alg1:

```
void Alg1(int n)
{
  for(int i=0; i < n*4; i=i+1)
  {
    tue_etwas(...);
  }
}
```

Alg2:

```
void Alg2(int n)
{
  for(int i=0; i < n; i=i+1)
  {
    for(int j=0; j<n; j=j+)
    {
      tue_etwas(...);
    }
  }
}
```

Aufwandsvergleich(Anzahl der Aufrufe von `tue_etwas`):

Tabelle 9: Aufwandsvergleich

n	1	2	3	4	5	6	7
Alg1	4	8	12	16	20	24	28
Alg2	1	4	9	16	25	36	49

Alg1 ⟵ linear (4n)
Alg2 ⟵ quadratisch (n^2)

- Aufwand abhängig von
  - Größe n der Eingabe, Anzahl der Daten usw.
  - der Komplexität des Algorithmus
- Unterscheidung zwischen
  (a) Zeitkomplexität (Laufzeit der Alg. s.o.)
  (b) Speicherkomplexität (benötigte Speichermenge)

- 90/10-Regel beim Programmieren:
„Ungefähr 90% der Laufzeit wird in ca. 10% des Programmcodes verbraucht“

4.2 Asymptotische Komplexität und Groß-O-Notation

- Abschätzung der Komplexität als Funktion von n
- Idee: “Wachstumsverhalten„möglichst allgemein für große n beschreiben
(Konstante Faktoren und Summanden werden nicht berücksichtigt)
(Zeichnung als Foto auf Handy)
- Die Groß-O-Notation weist einer Funktion $g : \mathbb{N} \to \mathbb{N}$ eine Menge von Funktionen $O(g(n))$ zu, die in gleicher Wachstumsbeziehung zu g stehen.
- Eine Funktion f ist Element von $O(g(n))$, falls sie nicht wesentlich schneller als g wächst.
Für große n muss gelten $f(n) \leq c - g(n)$ mit einer beliebigen Konstanten c.
- Hier z.B. $g(n)$ wächst so schnell wie $f(n)$;
$\Rightarrow f(n) \in O(g(n))$
in Bsp. aus 4.1.
Alg1: $f(n) = 4n \Rightarrow f(n) \in O(n)$
Alg2: $f(n) = n^2 \Rightarrow f(n) \in O(n^2)$

- <u>formale Definition</u>
$O(g(n))$ ist die Menge aller Funktionen $f(n)$, die asymptotisch beschränkt sind durch ein beliebiges aber konstantes Vielfaches der Funktion $g(n)$.
Mathematisch:
$O(g(n)) = \{f(n)|$Es gibt positive Konstanten c und n_0,so dass für alle $n \geq n_0$ gilt:$f(n) \leq c * g(n)\}$
  - $O(a)$ Konstanter Aufwand
  - $O(log(n))$ logarithmischer Aufwand
  - $O(n)$ linearer Aufwand
  - $O(n^2)$ quadratischer Aufwand
  - $O(n^k)$ polynomialer Aufwand
  - $O(2^n)$ exponentieller Aufwand
  - $O(n^n)$ exponentieller Aufwand
- Exponentielle Probleme sind im Allgemeinen nicht lösbar für große n.

- Wichtig: Reduktion der Komplexität durch Entwicklung effizienter Algorithmen.
  z.B. Fouriertransformation $O(n^2) \rightarrow$ schnelle Fouriertransformation (FFT): $O(n * log(n))$

- Rechenregeln:
  - Linearität: $O(c_1 * g(n) + c_2) = O(g(n)) \Rightarrow 4n + 3 \in O(n)$
  - Distributivität: $O(g_1(n)) + O(g_2(n)) = O(g_1(n) + g_2(n))$
    $O(g_1(n)) * O(g_2(n)) = O(g_1(n) * g_2(n))$
  - Außerdem: $O(O(g(n))) = O(g(n))$
    $g_1(n) * O(g_2(n)) = O(g_1(n) * g_2(n))$
  - Merke: $f(n) = a_k n^k + a_{k-1} n^{k-1} + \cdots + a_1 n + a_0$
    $\Rightarrow f(n) \in O(n^k) \Rightarrow 3n^5 + 2n^2 - 18n + 5 \in O(n^5)$

4.3 Beispiele

Berechnung von x^n (Version 1)

C/C++ Beispiel

```
double pow1(double x, int n)
{
  double y = 1;
  for(int i=0; i <n; i = i+1) // n Durchläufe
  {
    y = y*x;
  }
  return y;
}
```

Schleife wird (n)-mal durchlaufen $1 * x * x * x * x * x * \cdots * x$
$\Rightarrow$ Zeitkomplexität $f(n) = 1 \Rightarrow f(n) \in O(n)$

Berechnung von x^n (schneller) Rekursive Lösung des Problems durch fortlaufende Halbierung des Exponenten.
z.B. $x^8 = x^4 * x^4 \rightarrow x^4 = x^2 * x^2 \rightarrow x^2 = x * x$
anstatt 8 nur noch 3 Multiplikationen
$x^9 = x^4 * x^4 * x \rightarrow x^4 = \ldots$
C/C++

```
double pow2(double x, int n)
{
  if(n==0) return 1; // x^0 = 1
  if(x==1) return x; // x^1 = x
  double y = pow2(x, n/2) // Rekursionsaufruf
  y = y*y;
  if(n%2=1) y=y*x; //für ungerade n
  return y;
}
```

Wie oft lässt sich n halbieren?
$\rightarrow$ Logarithmus n (zur Basis 2)
$\Rightarrow$ Zeitkomplexität $f(n) = log_2 n$
$\Rightarrow$ $f(n) \in O(logn)$
Algorithmisches Prinzip: „Divide and Conquer“

5 neue Mitschrift

Whd:
statisches Array:

```
int a[4]; // Grö ß e konstant
a[0] = 35;
```

```
int size;
cin >> size;
int b[size]; // Compilerfehler
```

stattdessen ein dynamisches Array:

```
int* pa = new int[size];
pa[0] = 35;
delete[] pa; // Gibt den Sepicher wieder frei
```

Achtung! Klammern[] dürfen nicht vergessen werden; da sont nur das erste Element des Arrays freigegeben würde! In C analog zu ńewúnd ́delete:́

```
int* pa = (int*)malloc(size*4); //size*4 Byte reserviert
pa[0] = 1001;
...
free(pa);
```

oder besser:

```
int* pa = (int*)malloc(size*sizeof(int));
```

5.1 Dynmisches Erzeugen einer Struktur

```
struct Person
{
    string name;
    int alter;
    ...;
}
void main()
{
    Person z; //Statisch erzeugt
    z.alter = 29;
    Person* p; //Zeiger auf eine Person
    p = new Person; //Speicher reservieren
    (*p).alter = 29; // setzt Alter auf 29 C-Syntax
    p -> alter=29; // setzt alter in C++-Syntax
    ...
    delete p;
}
```

Bisher: Liste von Daten in Arrays, Problem bei Arrays:

- Die Elemente müssen alle hintereinander im Speicher abgelegt werden

- Einfügen und Löschen langsam, da die hinteren Elemente umkopiert werden müssen

5.2 Lineare Listen

- Lineare Listen sind verkettete Folgen von Datenelementen
- Jedes Element enthält neben den eigentlichen Daten einen zeiger auf das nächste Element

C/C++-Beispiel

```
struct Element
{
    string name;
    int nummer; // Personalnummer
    Element *next; // Zeiger auf nächstes Element
}
```

Element: name nummer next
Verkettete Liste von Elementen:
Anfang ⇒ 1. Element | next ⇒ ...
Nullzeiger (NULL bzw 0 markiert, dass ein Zeiger auf keine gültige Speicherstelle zeigt
Erzeugen und Verketten von einer Liste mit zwei Elementen (C++):

```
Element* anfang = new Element;
anfang -> name = "Peter Meier";
anfang -> nummer = 230001;
anfang -> next=NULL;
Element *zweites = new Element;
zweites -> name "Gabi Schmidt";
zweites -> nummer = 230002;
zweites -> next = NULL;
anfang -> next = zweites; // Verkettung vom ersten und zweiten Element
```

- Größe und Verkettung können während der Programlaufzeit verändert werden (Einfügen/Löschen von Elementen)
- Speicherplatz wird dynamisch zugeteilt
- Die einzelnen Listenelemente können sich an belibigen Stelle im Speicher befinden.

Operationen auf Listen:
Füge neues Element hinter dem Element auf das Zeiger p zeigt ein.
C++ Beispiel

```
void insertElemen(/Element *p, const string &name, int nummer)
{
    Element *q = new Element;
    q -> name = name;
    q -> nummer = nummer;
    q -> next = p -> next;
}
// Beispielaufruf (Einfügen eines neuen 3. elements):
insertElement(anfang->next, "Ute Müller", 230120);
```

Lösche das Element hinter dem Element auf das p zeigt:

```
void deleteElement(Element *p)
{
    if(p -> next !NULL) // Teste ob Nachfolger existiert
    {
        Element* q = p -> next;
        p -> next = q -> next; // neu verketten
        delete q;
    }
}
```

5.3 Bäume

- Ermöglicht Speicherung hierarchischer Strukturen (Gliederung in Dokumenten, Dateiverzeichnisse, verschachtelte terme, Vererbung, Organisationhierachien...)
- Elemnt des Baumes (Knoten) sind durch gerichtete Kanten verbunden (keine Zyklen!)
- Alle Elemnte außer der Wurzel haben einen Vorgänger
- Binärer Baum maximal 2 Nachfolger pro Knoten
- Definition: Ein binärer Baum B ist entweder leer oder besteht aus einer Wurzel w, sowie aus einem linken Teilbaum $B_l(w)$ und einem rechten Teilbaum $B_r(w)$.