3 Effizienz

Programme sollten möglichst effizient arbeiten

- geringer Einsatz an "Betriebsmitteln" wie Zeit, Speicher, Ein-Ausgabeeinheiten, Hilfsprogrammen, Datenübertragungseinrichtungen usw.

Programme sollten eine möglichst geringe Komplexität besitzen.

Umgangssprachliche Definition:
Komplexität eines Algorithmus = erforderlicher Aufwand an Betriebsmitteln, den eine Implementierung des Algorithmus als Programm auf einem Computersystem benötigt.
Komplexität eines Problems = kleinstmögliche Komplexität eines Algorithmus, der das Problem löst.

Sei P Algorithmus, der Problem π löst.
Dann:
- Komplexität von P = *obere Schranke* für Komplexität von π.
- Komplexität von π = *untere Schranke* für Komplexität von P.

wichtigste Betriebsmittel:
Laufzeit und *Speicherplatzbedarf*.

Im folgenden:
weitgehend Beschränkung auf Laufzeitbetrachtungen.
==> "effizient" = "in möglichst kurzer Zeit".

3.1 Laufzeit und Speicherbedarf eines Algorithmus

Gegeben: ein Problem π definiert durch eine Funktion

$f_\pi: X_\pi \rightarrow Y_\pi$

Beispiele:

1) Problem σ: $n \geq 1$ ganze Zahlen $x_1,x_2,...,x_n$ aufsteigend sortieren:

   $X_\sigma = \{(x_1,...,x_n) \mid n \geq 1, x_k \in \mathbb{Z}, k=1,...,n\}$ und

   $Y_\sigma = \{(x_{i_1},...,x_{i_n}) \mid x_{i_j} \in \mathbb{Z}, n \geq 1, \text{ mit } x_{i_1} \leq x_{i_2} \leq ... \leq x_{i_n}\}$.

   $f_\sigma: X_\sigma \rightarrow Y_\sigma$ liefert zu jeder Zahlenfolge $x \in X_\sigma$ die zugehörige aufsteigend sortierte Folge $f_\sigma(x)=y \in Y_\sigma$.

2) Problem μ: zwei natürliche Zahlen a und b multiplizieren:

   $X_\mu = \{(a,b) \mid a,b \in IN\}$ und $Y_\mu = IN$.

   f_μ liefert zu je zwei natürlichen Zahlen a,b das Produkt $f_\mu(a,b)=a*b$.

3) A endliche Menge.

   Problem η: zu $x \in A$ und Wort $w \in A^*$ feststellen, ob x in w vorkommt oder nicht:

   $X_\eta = \{(x,w) \mid x \in A \text{ und } w \in A^*\}$, $A_\eta = \{\text{"ja"}, \text{"nein"}\}$.

   f_η liefert zu jedem Paar (x,w) die Antwort "ja", falls x in w vorkommt, und sonst "nein".

Definition A:
Sei P eine Implementierung von π. Dann bezeichnen wir für $x \in X_\pi$ mit $\tau_P(x)$ bzw. $\sigma_P(x)$ die **Laufzeit** bzw. den neben der Eingabe *zusätzlichen* **Speicherplatz**, den P benötigt, um die korrekte Antwort auf die Eingabe x zu ermitteln. τ_P bzw. σ_P sind folglich Abbildungen der Funktionalität $X_\pi \rightarrow IN$.

$\tau_P(x)$: Anzahl der elementaren Einzelschritte, um f(x) zu ermitteln, und der für jeden Einzelschritt benötigten Zeit.

Eine Möglichkeit: τ_P experimentell für jede beliebige Fragestellung mit einer Stoppuhr bestimmen.
$\sigma_P(x)$: Zahl der Speicherzellen, die P für die Ermittlung des Ergebnisses *zusätzlich* zur Eingabe benötigt.

Allgemein: Gesucht sind **nicht** Laufzeit oder Speicherplatz eines Programms für konkrete Eingaben
Gesucht ist, wie sich das Programm qualitativ verhält, insbesondere wie sich die Laufzeit vergrößert, wenn man "schwierigere" Eingaben vorgibt.

1. Maß für „Schwierigkeit“: **Länge der Eingabe**:
- Sortierung von 100 Zahlen erfordert mehr Zeit als Sortierung von 5 Zahlen
- Multiplikation von 10-stelligen Zahlen dauert länger als Multiplikation von 2-stelligen.
- ==> Zusammenfassung von Eingaben zu sinnvollen „gleichschweren“ Klassen

Was verstehen wir genau unter der Länge einer Eingabe?

Formal: Zu Problem π **Längenfunktion**

$L_\pi: X_\pi \to IN$,

Eingabe $x \in X \to$ Länge $L_\pi(x)$.

Beispiele:

1) Sortierproblem σ:

   $L_\sigma: X_\sigma \to IN$ mit

   $L_\sigma(x_1,...,x_n)=n$,

   (Anzahl der Zahlen in der Folge oder

   $L'_\sigma: X_\sigma \to IN$ mit

   $L'_\sigma(x_1,...,x_n)=|x_1|+...+|x_n|$,

   (tatsächlich aufzuwendender Platz für die Zahlenfolge)

   L' ist offenbar nur von geringem Interesse, da die Komplexität des Sortierens von n abhängt.

2) Multiplikationsproblem μ:

   $L_\mu: X_\mu \to IN$ mit

   $L_\mu(a,b)$="Anzahl der Ziffern von a"+"Anzahl der Ziffern von b"
   +"zwei Vorzeichen".

   Wegen

   $[\log_{10} x]+2$="Anzahl der Ziffern von x"+"Vorzeichen",

   gilt:

   $L_\mu(a,b)=[\log_{10} a]+[\log_{10} b]+4$.

[u] = größte ganze Zahl kleiner oder gleich u (Gauß-Klammer).

Weitere Klassenbildung:

Zusammenfassung aller Laufzeiten für Eingaben gleicher Länge

--> Laufzeit im schlimmsten Fall (engl. *worst case*)

Für jedes $n \in IN$ greift man sich die Eingabe der Länge n heraus, für die das Programm die *größte* Laufzeit besitzt. Dies definiert die Laufzeitfunktion $T_P(n)$ und die Speicherplatzfunktion $S_P(n)$.

Definition B:

Sei π ein Problem und P ein Programm, das π implementiert.

Die **Laufzeit** T_P **im schlimmsten Fall** (**worst case**) des Programms P ist eine Abbildung

$T_P: IN \rightarrow IN$ mit

$T_P(n)=\max\{\tau_P(x) \mid x \in X_\pi \text{ und } L_\pi(x)=n\}$.

Der **Speicherbedarf** S_P **im schlimmsten Fall** des Programms P ist eine Abbildung

$S_P: IN \rightarrow IN$ mit

$S_P(n)=\max\{\sigma_P(x) \mid x \in X_\pi \text{ und } L_\pi(x)=n\}$.

Im folgenden immer:

"Laufzeit" meint Laufzeit im schlimmsten Fall

"Speicherplatz" meint Speicherbedarf im schlimmsten Fall.

Beispiel: P liest eine ganze Zahl x ein und stellt fest, ob in der nachfolgenden Zahlenfolge die Zahl x vorkommt oder nicht. P löst also Problem η:

```
program P(input,output);
var x,y: integer;
    ende: boolean;
begin
    read(x);  {Zeit C}
    ende:=false;  {Zeit C'}
    while not (eof or ende) do  {Zeit C"}
    begin
        read(y);  {Zeit C}
        ende:=x=y  {Zeit C'"}
    end;
    if ende then writeln ('Zahl ist vorhanden')
       else writeln ('Zahl ist nicht vorhanden')  {Zeit C""}
end.
```

Wir berechnen die Laufzeit des Programms.

1. Längenfunktion

$L_\eta: X_\eta \rightarrow \mathbb{N}$ mit $X_\eta=\{(x, y_1,...,y_n)| x,y_1,...,y_n \in \mathbb{Z}, n\geq 0\}$ und

$L_\eta(x, y_1,...,y_n)=n+1$.

Eingabe der Länge 1:

$T_P(1)=C+C'+C"+C'"$.

Eingabe der Länge 2:

$T_P(2)=C+C'+C"+C+C'"+C"+C'"=2C+C'+2C"+C'"+C'"$.

Eingabe der Länge 3: Laufzeit

$2C+C'+2C"+C'"+C'"$,

falls x erste der beiden Zahlen. Falls x zweite Zahl oder gar nicht enthalten, so kommt Zeit C+C"+C'" hinzu. Gesamtzeit also:

$3C+C'+3C"+2C'"+C'"$.

Laufzeit im schlimmsten Fall also:

$T_P(3)=3C+C'+3C''+2C'''+C''''$.

Allgemein für $n \geq 2$:

$2C+C'+2C''+C'''+C''''$, falls x an der 1. Stelle vorkommt, oder
$3C+C'+3C''+2C'''+C''''$, falls x an der 2. Stelle vorkommt, oder
$4C+C'+4C''+3C'''+C''''$, falls x an der 3. Stelle vorkommt, oder
…
$nC+C'+nC''+(n-1)C'''+C''''$, falls x an der letzten Stelle oder gar nicht vorkommt.

Schlimmster Fall also:

$T_P(n)=nC+C'+nC''+(n-1)C'''+C''''$.

Nebenbei: Speicherbedarf konstant 3, also $S_P(n)=3$ für alle n.

Weitere Abstraktion und Klassenbildung:
Recht problematisch und unübersichtlich: Zeitkonstanten C, C' usw., die für jeden Vergleich und jede Elementaranweisung vergeben müssen.
Folge: $T_P(n)$ extrem unübersichtlich.

Lösung: **Einheitskostenmodell**

- Idee: genormten Rechner
- jede Elementaroperation **eine** Zeiteinheit an.
  - **hier** elementar: elementaren Operationen, also Arithmetik, Vergleich, Zugriff zu Feldern, Zuweisung usw. jeweils ein Rechenschritt, also eine Zeiteinheit.
- analog für Speicher: Jedes elementare Datum kann in einer Speicherzelle untergebracht werden.

Ist das Einheitskostenmodell realistisch?

Voraussetzungen:

- auftretende Operanden (z.B. in Zwischenrechnungen) nicht beliebig groß, sondern in der Größenordnung der Länge der Eingabe --> praktisch fast immer der Fall
- Ausnahme: arithmetische Algorithmen, wie z.B. Multiplikationsalgorithmen. Hier: Zeit für einen Rechenschritt in Beziehung zur Größe der beteiligten Operanden (Anzahl ihrer Ziffern).

Konsequenz:

logarithmisches Kostenmodell:

$$L(n)=[\log_2 n]+1.$$

Beispiel: Addition mit den Operanden a und b:

$$L(a)+L(b)=[\log_2 a]+[\log_2 b]+2$$

Rechenschritte als Zeiteinheiten an.

Im folgenden **nur** Einheitskostenmodell.

Beispiel: im Einheitskostenmodell (s. besp. oben):

$$C=C'=C'''=1 \text{ und } C''=C''''=2.$$

Laufzeit im schlimmsten Falle also

$$T_P(n)=4n+2.$$

Ziel: Suche zu jedem Problem den schnellsten und bezgl. Speicherplatz anspruchslosesten Algorithmus

Aber: Wie vergleicht man Algorithmen?

Beispiel: Gegeben vier Algorithmen A, B, C und D zu einem Problem π:

$T_A(n)=100n+30$

$T_B(n)=100n \cdot \log_2 n$,

$T_C(n)=10n^2$,

$T_D(n)=2^n$.

- $2 \leq n \leq 9$: D am schnellsten
- $n=10$: C am schnellsten
- $n>10$: A am schnellsten.
- B niemals der schnellste.

Empfehlung: Bis auf wenige (endlich viele) Ausnahmen Algorithmus A verwenden.

Zwei Vergleichsmaßstäbe:

1. natürliche Vorstellung von "schneller"
2. „meist schneller“: Situation, in der ein Algorithmus für Eingaben kurzer Länge zwar schneller ist als ein anderer, von einer gewissen Eingabegröße an jedoch den zweiten stets übertrifft.

Definition C:
Gegeben seien zwei Algorithmen/Programme A und B. Dann gilt:

a) A ist **schneller** als B, falls

$$T_A(n) \leq T_B(n) \text{ für alle } n \in \mathbb{N} \text{ ist.}$$

b) A ist **asymptotisch schneller** als B, falls

$$\lim_{n \to \infty} \frac{T_A(n)}{T_B(n)} = 0.$$

Analog für Speicherplatz.

Beispiel:

- A asymptotisch schneller als B
- B asymptotisch schneller als C
- C asymptotisch schneller als D,

denn

$$\lim_{n \to \infty} \frac{T_A(n)}{T_B(n)} = \lim_{n \to \infty} \frac{100n+30}{100n \cdot \log_2 n} = 0.$$

Merke: Ist ein Programm asymptotisch schneller als ein anderes, so ist es bis auf endlich viele Ausnahmen "um eine Größenordnung" schneller.

Ziel: Finde asymptotisch schnellstes Programm

3.2 Die Ordnung einer Funktion

Abstraktion: qualitativer Verlauf einer Funktion $T_P(n)$, ihre *Größenordnung* oder kurz *Ordnung*,

Wunsch: Aussagen der Art: $T_P(n)$ oder auch $S_P(n)$ verhalten sich wie eine quadratische Funktion.

Definition A:
Sei $f: IN \to IN$ eine Funktion. Dann ist

$$O(f)=\{g: IN \to IN \mid \text{es gibt Zahlen } c, n_0 \in IN, \text{ so daß } g(n) \leq c \cdot f(n) \text{ für alle } n \geq n_0\}.$$

Falls $g \in O(f)$ ist, so hat g die **Ordnung** f. O heißt auch **Landausches Symbol**.
Man spricht O(f) als "groß O von f" oder kurz "O von f".

O(f) ist also die Menge aller Funktionen $g: IN \to IN$, die von f für genügend große Argumente bis auf einen konstanten Faktor majorisiert werden.

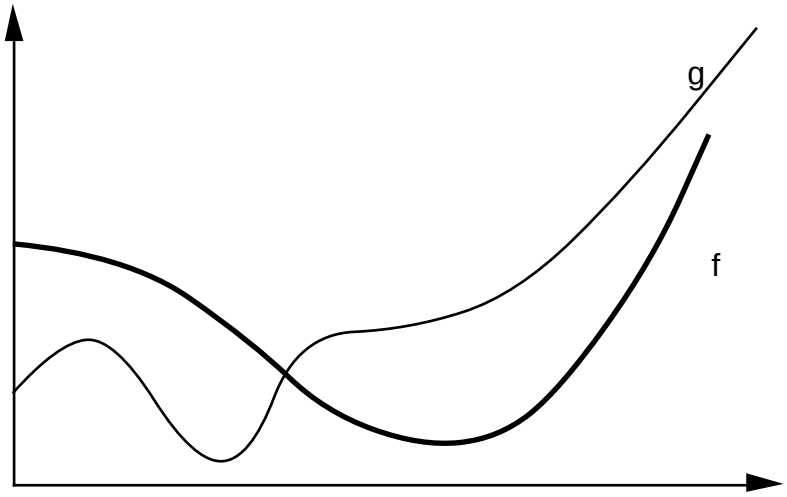


Abb.: Zwei Graphen f und g

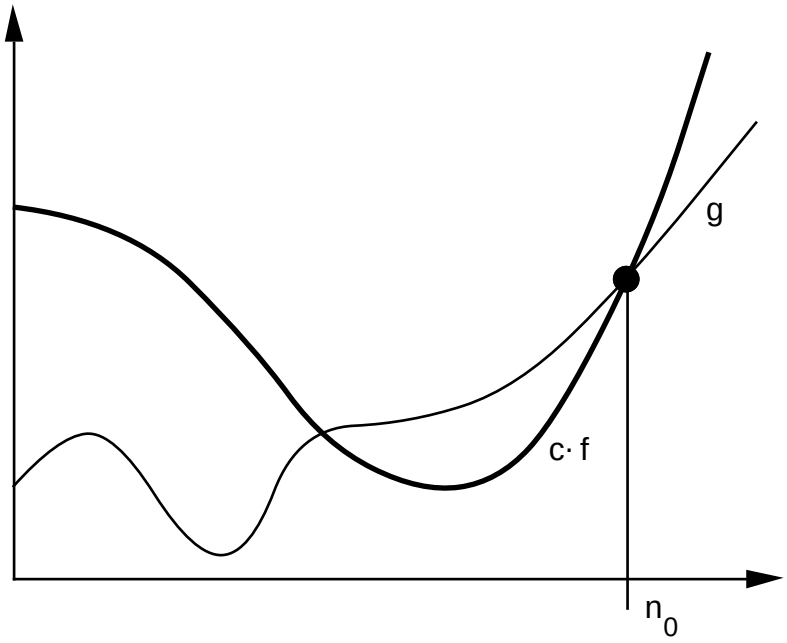


Abb.: Veranschaulichung von $g \in O(f)$

Beispiele:

1) g_1 konstante Funktion, also für festes $a \in IN$:

   $g_1(n)=a$ für alle $n \in IN$.

   Offensichtlich: $g_1 \in O(id)$ mit $id(n)=n$ für alle $n \in IN$, wenn $c=1$ und $n_0=a$ gesetzt wird; denn für $n \geq n_0=a$ ist $g_1(n)=a \leq n=id(n)$.

   Gleichzeitig: $g_1 \in O(1)$. Dazu wähle $n_0=1$ und $c=a$.

   Dann für alle $n \geq n_0$

   $g_1(n)=a \leq c \cdot 1=c$.

2) g_2 Polynom vom Grad m, also

   $g_2(n)=c_m n^m+c_{m-1}n^{m-1}+...+c_2n^2+c_1n^1+c_0, \ c_m \geq 0$.

   Dann: $g_2 \in O(n^m)$.

   Beweis: Es gibt stets einen Wert n_0, der von m und den Koeffizienten c_m, c_{m-1}, ... ,c_0 abhängt, so daß für alle $n \geq n_0$ gilt:

   $c_m n^m \geq c_{m-1}n^{m-1}+...+c_2n^2+c_1n^1+c_0$,

   da n^m schneller wächst als die Summe der übrigen Potenzen. Dann ist für $n \geq n_0$

   $g_2(n) \leq 2c_m n^m$,

   also gilt die Behauptung mit diesem n_0 und $c=2c_m$.

   Gegeben: $h(n)=3n^2+n$. Dann gilt $h \in O(n^2)$, aber auch $h \in O(n^k)$ für beliebiges $k>2$, oder auch $h \in O(n^2 \cdot \log_2 n)$.

3) g_3 Fakultätsfunktion:

   $g_3(n)=n!=1 \cdot 2 \cdot 3 \cdot ... \cdot n$.

   Gilt:

   $g_3(n) \leq n^n$ und somit $g_3 \in O(n^n)$.

4) g_4 Logarithmus zur Basis $b \in IR$, $b>1$ beliebig, also

$$g_4(n)=\log_b n.$$

Wegen:

$$\log_b n=\log_b 2 \cdot \log_2 n.$$

Gilt: $g_4 \in O(\log_2 n)$. Dazu wähle $c=\log_b 2$.

„Kuriose“ Schreibweisen:

O-Notation mit Gleichheitszeichen (bzw. Ungleichheitszeichen) statt mit den Mengensymbolen $\in$ und $\subseteq$ (bzw. $\notin$), z.B. anstelle

$$5n^3+2n^2 \in O(n^3+2n^2) \subseteq O(n^3) \subseteq O(n^7 \cdot \log_2 n)$$

schreibt man

$$5n^3+2n^2=O(n^3+2n^2)=O(n^3)=O(n^7 \cdot \log_2 n).$$

Da es sich in dieser "Gleichung" eben **nicht** um Gleichheitszeichen handelt, ist Vorsicht geboten: Man darf eine "Gleichung", in der O-Ausdrücke vorkommen, nur von links nach rechts lesen.

In einer "Gleichung" der Form

$$f_1=O(f_2)=O(f_3)=...=O(f_m)$$

repräsentieren die O-Ausdrücke von links nach rechts gelesen immer größere Mengen von Funktionen; die zugehörigen Schranken f_i werden bezüglich f_1 also gröber und gröber. Sprechweise für $f_1=O(f_2)$: f_1 ist von höchstens der Größenordnung f_2.

Weitere Abstraktion:
Einteilung von Probleme und Algorithmen in *Komplexitätsklassen* ein.

$T(n) = O(n)$: *lineare Laufzeit* (kurz: *Linearzeit*).

$T(n) = O(n^2)$: *quadratischr Laufzeit*

$T(n) = O(n^3)$: *kubische Laufzeit*

$T(n) = O(n^k)$: *polynomielle Laufzeit* (kurz: *Polynomialzeit*),

$\uparrow\uparrow\uparrow\uparrow\uparrow\uparrow\uparrow$ praktisch relevant (*leicht*)

$\downarrow\downarrow\downarrow\downarrow\downarrow\downarrow\downarrow$ unzugänglich (intractable), *hart*

$T(n) = O(2^{p(n)})$: *exponentielle Laufzeit* (kurz: *Exponentialzeit)*, p Polynom

$T(n) = O(2^{2^{p(n)}})$: *superexponentiell*

$T(n)$ \ n	20	30	40	50	100	
n	0.0002	0.0003	0.0004	0.0005	0.001	Sekunden
n^2	0.004	0.009	0.016	0.025	0.1	Sekunden
n^5	32	243	1024	3125	100000	Sekunden
2^n	10 Sek.	3 Stunden	4 Monate	360 Jahre	$4 \cdot 10^{17}$	Jahre

Tab.: Polynomial- und Exponentialzeit

Die Rechengeschwindigkeit eines Computers hilft kaum, die Hürde zwischen Polynomialzeit und Exponentialzeit zu überspringen.

Leider: für eine große Klasse von wichtigen Problemen, die sog. **NP-vollständigen Probleme**, kennt man bisher keinen Polynomialzeitalgorithmus. berechtigter Verdacht: gibt auch keine Polynomialzeitalgorithmen.

3.3 Beispiele: Suchen und Sortieren

Beispiel 1: **Suchen in einem Feld.**

Gegeben: aufsteigend sortiertes lineares Feld a mit n≥1 Elementen vom Typ integer
ganze Zahl x; a ist.

Gesucht: Programm, das ausgibt, ob x in a vorkommt oder nicht.

1. Lösung: **sequentielles Suchen**:

```
program seqsuche(input,output);
const n=...;
var a: array [1..n] of integer;
    i,x: integer;
    gefunden: boolean;
begin
    {Das Feld a sei gegeben}
    read(x); i:=1; gefunden:=false;
    while (not gefunden) and (i≤n) do
        if a[i]=x then gefunden:=true else i:=i+1;
    write(gefunden)
end.
```

Laufzeit:
T(n)=4n+6, d.h. T(n)=O(n).
Speicherbedarf: S(n)=O(1).
Linearzeitalgorithmus.

2. Lösung: Hilfsannahme: n gerade.

Falls n ungerade, ersetze im folgenden überall n durch (n+1).

Vergleiche x mit $a[n/2]$.

Falls $x=a[n/2]$: Suche erfolgreich

Sonst wende das Verfahren

für $x<a[n/2]$ *rekursiv* auf das linke Teilfeld von $a[1]$ bis $a[n/2-1]$ oder

für $x>a[n/2]$ auf das rechte Teilfeld von $a[n/2+1]$ bis $a[n]$ an,

Beispiel: n=8, x=12.

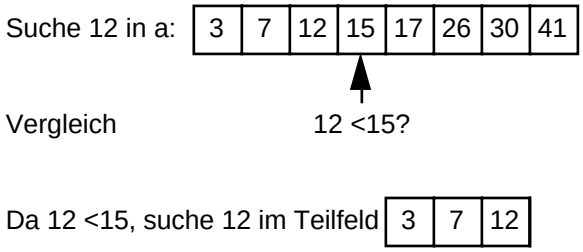


Abb. 3: Prinzip des binären Suchens

binäres Suchen:

```
program binaersuche(input,output);
const n=...;
var a: array [1..n] of integer;
    x: integer;
function binsuche(i,j,x: integer): boolean;
{binsuche sucht die Zahl x im Feld von a[i] bis a[j]}
var m: integer;
begin
    if j<i then binsuche:=false else
    begin
        m:=(i+j) div 2;
        if x<a[m] then binsuche:=binsuche(i,m-1,x) else
            if x>a[m] then binsuche:=binsuche(m+1,j,x) else
                binsuche:=true
    end
end;
begin
    {Das Feld a sei gegeben}
    read(x);
    write(binsuche(1,n,x))
end.
```

Laufzeit T(n) im schlimmsten Fall?

Was ist der schlimmste Fall ? Wenn x nicht in a vorkommt.

Je Halbierung konstant viele Rechenschritte, z.B C.

Je rekursive Anwendung der Funktion binsuche auf ein Feld der Länge $n/2$ nach Annahme $T(n/2)$ Schritte.

Folglich:

$T(n)=C+T(n/2)$ zu lösen.

$T(1)=C'$.

Genaue Größen von C und C' unerheblich, daher einfach C=C' setzen.

Folglich:

$T(n)=C+T(n/2)$ für $n>1$,

$T(1)=C$.

Zur Lösung setze $n=2^k$:

$$T(2^k)=C+T(2^{k-1}),$$

$$T(2^0)=C.$$

Offensichtlich gilt:

$$T(2^k)=C(k+1).$$

Ersetzung rückgängig machen, d.h. $k=\log_2 n$, so folgt

$$T(n)=C(\log_2 n+1).$$

Übergang zur Ordnung:

$$T(n)=O(\log_2 n).$$

Laufzeitgewinn anschaulich: Feld mit 1 Million Elementen im schlimmsten Fall größenordnungsmäßig

mit sequentieller Suche 1 Million Schritte,

mit binärer Suche 20 Schritte.

Speicherbedarf:

Auf den ersten Blick: konstant.

Zweiter Blick: Größe des Stacks kommt hinzu.

- Je Rekursionsaufruf konstante Zahl von Daten auf den Stack.
- Stackgröße selbst hängt von Zahl der geschachtelten rekursiven Aufrufe ab.
- Schlimmster Fall: gesuchtes Element nicht im Array.
- $O(\log_2 n)$ Aufrufe nötig; dann maximal $O(\log_2 n)$ Einträge. Folglich:

$$S(n)=O(\log_2 n).$$

Effizenter: nicht-rekursiven Algorithmus ermitteln (ohne Stack).

Laufzeit wie bisher $O(\log_2 n)$ Zeit

Speicherbedarf nur $O(1)$.

Beispiel 2: **Sortieren eines Feldes.**

Idee: vertausche zwei benachbarte Feldelemente miteinander, wenn sie in der falschen Reihenfolge stehen:

```
program sort;
const n=...;
var a: array [1..n] of integer;
    i,j,t: integer;
begin
    for i:=n-1 downto 1 do
        for j:=1 to i do
            if a[j]>a[j+1] then
            begin
                t:=a[j];
                a[j]:=a[j+1];
                a[j+1]:=t
            end
end.
```

Bubblesort

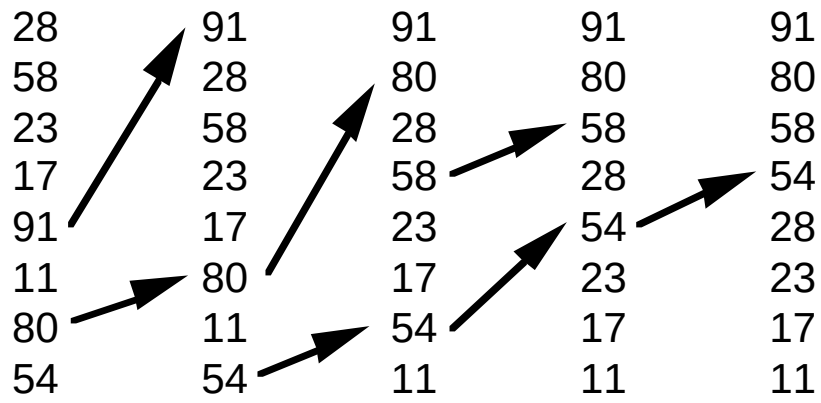


Abb.: Arbeitsweise von Bubblesort

Laufzeitanalyse

Schleifenrumpf: C Rechenschritte an.

Für i=1 wird die innere Schleife einmal durchlaufen, für i=2 zweimal, ..., für i=n-1 entsprechend (n-1)-mal, also insgesamt

1+2+3+...+(n-2)+(n-1) mal

durchlaufen. Zeit daher

$$T(n)=C(1+2+3+...+(n-2)+(n-1))=C/2 \cdot n(n-1)=O(n^2)$$

Sortieren durch Mischen

Beispiel: Gegeben seien die beiden Folgen

f: 7 19 2 13

g: 1 24 3 4

1. Schritt:

f: 7 19 2 13

h: 1

g: 24 3 4

2. Schritt:

f: 19 2 13

h: 1 7

g: 24 3 4

3. Schritt:

f: 2 13

h: 1 7 19

g: 24 3 4

usw.

8. Schritt:

h: 1 7 19 2 13 24 3 4

Was wird durch das Mischen erreicht?

Lauf: maximale aufsteigend sortierte Teilfolge der Folge
Anzahl <=> Grad der Sortierung aus.

Beispiel: drei Läufe: 1,7,19 und 2,13,24 und 3,4.

Mischen reduziert die Anzahl der Läufe auf höchstens die Hälfte.

Beispiel: Die Folge h aus obigem Beispiel spaltet man auf in

f: 1 7 19 3 4

g: 2 13 24

und mischt sie zusammen zu

h: 1 2 7 13 19 3 4 24.

Nun zwei Läufe: 1,2,7,13,19 und 3,4,24.

In jedem Fall: aus maximal $n/2$ Läufen der vorangegangenen Phase werden maximal $n/4$ Läufe in der nächsten Phase.

Verfahren: Spalte fortlaufend die Folge in zwei Hälften und mische die beiden Folgen zu einer einzigen. Besteht die Ergebnisfolge nur noch aus einem einzigen Lauf, dann ist die Sortierung beendet.

Laufzeit:
Mischphase: O(n) Schritte.
Einmaliges Aufspalten einer Folge: O(n) Schritte.
Laufzeit des Gesamtalgorithmus:

$$T(n)=O(n\cdot\text{"Anzahl der Mischphasen"}).$$

Wie oft muß man eine Folge aufspalten und mischen?

Nach dem ersten Mischen: $\leq n/2$ Läufe

nach dem zweiten Mischen $\leq n/4$ Läufe

nach dem dritten Mischen $\leq n/8$ Läufe

…

nach dem k-ten Mischen $\leq n/2k$ Läufe usw.

Nach spätestens $\log_2 n$ Phasen ein Lauf=eine sortierte Folge vor.

Laufzeit also:

$$T(n)=O(n\cdot\log_2 n).$$

Bemerkung: In der Praxis verwendet man nur Sortierverfahren, die in $O(n\cdot\log_2 n)$ Schritten arbeiten. Langsamere Verfahren sind unbrauchbar.

4.4 Untere Schranken für die Laufzeit

Grundfrage: Wie schnell kann man sortieren?

O(n2)
O(n log n)
$O(n \cdot \log_2 \log_2 n)$
…
O(n) ?

Grundannahme: nur Sortieralgorithmen, die auf Vergleichen basieren.
Je nach Ergebnis des Vergleichs ≤ oder > vertauscht man zwei Elemente und vergleicht anschließend zwei andere Elemente usw.
Wir zählen die Anzahl der Vergleiche.

Methode: Entscheidungsbaum.

Beispiel:

Sortierung von drei Elementen a_1,a_2,a_3, (alle verschieden).

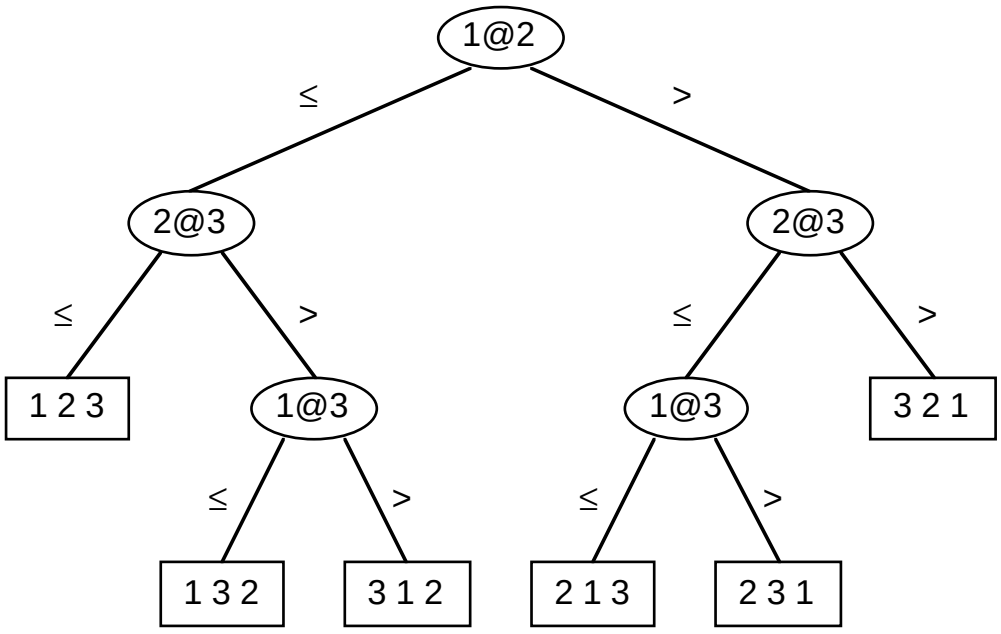
Vorgehen:

Vergleiche a_1 mit a_2, anschließend a_2 mit a_3.

- Falls $a_1<a_2$ und $a_2<a_3$, so war Folge schon sortiert.
- Falls $a_1<a_2$ und $a_2>a_3$, so vergleiche noch a_1 mit a_3.
    - Falls $a_1<a_2$ und $a_1>a_3$ galt, so ist a_3,a_1,a_2 die sortierte Folge.

Darstellung aufeinanderfolgende Vergleiche als binären Baum B (sog. *Entscheidungsbaum*):

- Innere Knoten: jeweils zu vergleichende Elemente (i@j -> a_i mit a_j vergleichen).
- Je nach Ergebnis des Vergleichs ≤ oder > in linken oder rechten Teilbaum verzweigen.
- Blätter: Reihenfolge der Elemente a_1,a_2,a_3, die aufsteigend sortiert ist (statt a_i nur i).



Definition A:
Ein **Entscheidungsbaum** für eine Folge $a_1,a_2,...,a_n$ ist ein binärer Baum, dessen Knoten mit Ausnahme der Blätter mit Markierungen der Form i@j versehen sind. Die beiden Kanten zu den Söhnen jedes Knotens sind mit ≤ bzw. > markiert. Jedes Blatt ist mit *der* Umordnung der Folge $a_1,...,a_n$ markiert, die alle Vergleiche erfüllt, die auf dem Weg von der Wurzel zu diesem Blatt auftreten.

Algorithmus, der auf Vergleichen beruht ==> ein Entscheidungsbaum.

Laufzeit des Algorithmus im schlimmsten Fall

$\geq$

Maximalzahl von Vergleichen zum Sortieren der Folge

$=$

Länge des *längsten* Weges von der Wurzel zu einem Blatt (minus Eins)

Bezeichnung:
$V_B(n)$:= Maximalzahl von Vergleichen, die zur Sortierung von n Objekten mit Entscheidungsbaum B nötig ist.

Algorithmus mit wenigsten Vergleichen
entspricht
Entscheidungsbaum, in dem Länge des längsten Weges möglich klein.

Gesucht also:

$$\min\{V_B(n) | \text{ B ist ein Entscheidungsbaum für n Elemente}\}$$

(untere Schranke für die Laufzeit von Sortieralgorithmen)

Grundüberlegung: Wieviele Blätter haben Entscheidungsbäume?

- Für n Elemente n! verschiedene Anordnungen.
- Eine Anordnung ist gesuchte sortierte Reihenfolge.
- Jede Anordnung kommt als Blatt des Entscheidungsbaumes vor und gibt dort an, wie die Ausgangsfolge $a_1,...,a_n$ umgeordnet werden muß, um eine sortierte Reihenfolge herzustellen.
- Jeder Entscheidungsbaum muß mindestens so groß sein, daß er alle n! Blätter aufnehmen kann.

- $V_B(n)=1$ ==> max. zwei Blättern.
- $V_B(n)=2$ ==> max. 4 Blätter.
- Allgemein: $V_B(n)$ Vergleiche ==> max. $2^{V_B(n)}$ Blätter.

Umgekehrt: n! Blätter, daher

$$2^{V_B(n)} \geq n!$$

bzw.

$$V_B(n) \geq \log_2(n!).$$

Fertig: Ein optimaler Sortieralgorithmus benötigt im schlimmsten Fall mindestens $\log_2(n!)$ Vergleiche, um n Elemente zu sortieren.

Ausrechnen liefert:

$$\log_2(n!) = \sum_{i=1}^{n} \log_2 i \geq \sum_{i=n/2}^{n} \log_2 i \geq {}^n/_2 \cdot \log_2({}^n/_2) = {}^n/_2 \cdot (\log_2 n - 1).$$

Satz B:
Jedes allgemeine auf Vergleichen beruhende Sortierverfahren benötigt im schlimmsten Fall mindestens $O(n \cdot \log_2 n)$ Vergleiche bzw. Rechenschritte.

"Sortieren durch Mischen" ist optimal.