

ADS: Algorithmen und Datenstrukturen

Uwe Quasthoff

Institut für Informatik
Abteilung Automatische Sprachverarbeitung
Universität Leipzig

10. Oktober 2017

[Letzte Aktualisierung: 09/10/2017, 21:01]

Übungsaufgaben, Vorlesungsfolien, Termin- und Raumänderungen, ...

<http://adsprak.informatik.uni-leipzig.de/>

Erreichbar über

<http://asv.informatik.uni-leipzig.de>

→ Lehre → Algorithmen und Datenstrukturen 1

<http://asv.informatik.uni-leipzig.de/de/courses/233>

Anmeldung für Vorlesungen und Übungen

Bitte befolgen Sie die Regeln

- 1 Melden Sie sich zur **Vorlesung** im **AlmaWeb** an.
- 2 Melden Sie sich zu einer der **Übungen** im **AlmaWeb** an.
- 3 Erledigen Sie beides bis spätestens **16.10.2017**.

Achtung!

- Ohne Anmeldung haben sie **keinen** Platz in einer der Übungen.
- Falls Sie sich **nicht rechtzeitig im AlmaWeb** angemeldet haben, so ist das Studienbüro ihr Ansprechpartner, **nicht** wir.
- Wechsel in andere Übungsgruppe nur mit Tauschpartner – um den **Sie** sich kümmern müssen.

- Abzugeben sind Lösungen zu sechs Aufgabenblättern.
- Termine

Aufgabenblatt	1	2	3	4	5+6
Ausgabe	17.10.	7.11.	21.11.	05.12.	19.12.
Abgabe	24.10.	14.11.	28.11.	12.12.	16.01.

- Lösungen sind **spätestens direkt vor** Beginn der Vorlesung im Hörsaal abzugeben. Spätere Abgaben werden **nicht** gewertet
- Lösungen werden bewertet und in der auf den Abgabetermin folgenden Übungsstunde zurückgegeben.

Vorläufige Termine der Übungsgruppen

Gruppe	Uhrzeit	Tag	Raum	Übungsleiter
01	11:15 - 12:45	Mo	SG 3-10	Paul Eisenhuth
02	11:15 - 12:45	Mo	SG 3-12	Andreas Niekler
05	09:15 - 10:45	Di	SG 3-12	Manuela Geiß
06	09:15 - 10:45	Di	SG 3-10	Nathanael Philipp
07	09:15 - 10:45	Mi	SG 3-13	Paul Eisenhuth
08	09:15 - 10:45	Mi	SG 3-12	Lydia Müller
13	17:15 - 18:45	Mi	SG 1-10	Maciej Sumalvico
03	17:15 - 18:45	Mi	SG 3-10	Manuela Geiß

Vorläufige Termine der Übungsgruppen

Gruppe	Uhrzeit	Tag	Raum	Übungsleiter
14	15:15 - 16:45	Do	SG 1-10	Andreas Niekler
15	15:15 - 16:45	Do	SG 4-10	Thomas Efer
09	09:15 - 10:45	Fr	SG 3-12	Sven Findeiss
10	09:15 - 10:45	Fr	SG 3-10	Maximilian Bryan
11	11:15 - 12:45	Fr	Paulinum, P701	Thomas Efer
12	11:15 - 12:45	Fr	SG 2-14	Sven Findeiss
16	11:15 - 12:45	Fr	SG 4-11	Jochen Tiepmar
04	13:15 - 15:45	Fr	SG 2-14	Jochen Tiepmar

Termin: 06.02.2018, 13:00–14:30 Uhr im AUDIMAX

Zulassungsvoraussetzungen

- Modul- UND Prüfungsanmeldung in **AlmaWeb** bis 16.10.2017
- Erreichen von 50% der Punkte in den Übungsaufgaben
- Fähigkeit, abgegebene Lösungen an der Tafel zu erläutern
- → wird in den Übungen überprüft
- → abgeben allein reicht **nicht**
- wer absolut nicht kann: **frühzeitig** Alternative für Überprüfung mit Übungsleiter abklären

- Team-Arbeit bei den Übungsblättern ist prinzipiell gestattet, ABER:
- jeder muß alle abgegebenen Lösungen erklären können;
- jeder muß eine eigene Abgabe haben;
- **keine Kopien** von Ausarbeitungen;
- Abgabe in Papierform, **keine Möglichkeit** per email abzugeben;
- **Lösungen links oben tackern**; kein: kleben, vernähen, Büroklammern, schweißen, sonstwas;
- Loseblattsammlungen in welcher Form auch immer werden **nicht** gewertet;
- **Name, Matrikelnummer** und **Übungsgruppe** auf jedes Blatt.

Abgabe vor der Vorlesung, sortiert nach Übungsgruppen

Wenn Sie direkt vor der Vorlesung nicht abgeben können:

- geben Sie die Abgabe einem Ihrer Kommilitonen mit zur Abgabe direkt vor der Vorlesung
- oder werfen Sie die Abgabe bis spätestens **08 Uhr** (acht Uhr morgens) am Abgabetag in den Briefkasten der Abteilung *Automatische Sprachverarbeitung* in der 5. Etage im Augusteum, Raum A514.

Spätere Abgaben werden nicht gewertet!

<http://adsprak.informatik.uni-leipzig.de/>

Kontaktadresse

adshelp@informatik.uni-leipzig.de

- 1 Einführung: Typen von Algorithmen, Komplexität von Algorithmen
- 2 Einfache Suchverfahren in Listen
- 3 Verkettete Listen, Stacks und Schlangen
- 4 Sortierverfahren
 - Elementare Verfahren
 - Shell-Sort, Heap-Sort, Quick-Sort
 - Externe Sortierverfahren
- 5 Allgemeine Bäume und Binärbäume
 - Orientierte und geordnete Bäume
 - Binärbäume (Darstellung, Traversierung)
- 6 Binäre Suchbäume
- 7 Mehrwegbäume

- Algorithmen stehen im Mittelpunkt der Informatik
- Entwurfsziele bei Entwicklung von Algorithmen:
 - 1 Korrektheit: Macht der Algorithmus, was er soll? Können wir das beweisen?
 - 2 Terminierung: Wird der Algorithmus auch immer irgendwann fertig?
 - 3 Effizient: Funktioniert es auch zügig mit vielen Daten?
- Wahl der Datenstrukturen ist für Effizienz entscheidend.
- Schwerpunkt der Vorlesung: Entwurf von effizienten Algorithmen und Datenstrukturen, sowie die nachfolgende Analyse ihres Verhaltens.

Wozu das Ganze?

- Funktional gleichwertige Algorithmen weisen oft erhebliche Unterschiede in der Effizienz (Komplexität) auf.
- Bei der Verarbeitung soll effektiv mit den Daten umgegangen werden.
- Die Effizienz hängt bei großen Datenmengen ab von
 - internen Darstellung der Daten
 - dem verwendeten Algorithmus
- Zusammenhang dieser Aspekte?

Anforderungen

- 700 MB Speicherplatz
- 40 Millionen Telefone \times 35 Zeichen (Name Ort Nummer)
1.4 GB ASCII-Text
- **passt nicht**
- Wir brauchen eine Komprimierung der Daten, und eine Möglichkeit schnell zu suchen (Index!).
- (Technische Nebenbemerkung: außerdem ist der Speicherzugriff auf der CD sehr langsam)

Design-Überlegungen

- Verwende Datenstruktur, die die vollständige Verwaltung der Einträge erlaubt: Suchen, Einfügen und Löschen.

– ODER –

- Weil sowieso nur das Suchen erlaubt ist:
verwende Datenstruktur, die zwar extrem schnelles Suchen in komprimierten Daten erlaubt, aber möglicherweise kein Einfügen.

Also: Mitdenken hilft

- Die Abarbeitung von Programmen (Software) beansprucht zwei Ressourcen:
Zeit und Hardware (insbesondere: **Speicher**).
- **FRAGE:** Wie steigt dieser Ressourcenverbrauch bei größeren Problemen (d.h. mehr Eingabedaten)?
- Es kann sein, dass Probleme ab einer gewissen Größe praktisch unlösbar sind, weil
 - ① Ihre Abarbeitung zu lange dauern würde (z.B. länger als ein Informatikstudium) oder
 - ② Das Programm mehr Speicher braucht, als zur Verfügung steht.

Wichtig ist auch der Unterschied zwischen **internem (RAM) und externem Speicher**, da z.B. der Zugriff auf eine Festplatten ca. 100.000 mal langsamer ist als ein RAM-Zugriff.

- **Wesentliche Ressourcen:**
 - Rechenzeitbedarf
 - Speicherplatzbedarf
- **Programmlaufzeit abhängig von**
 - Eingabe für das Programm
 - Qualität des vom Compiler generierten Codes
 - Leistungsfähigkeit der Maschine, die das Programm ausgeführt
 - “Kompliziertheit” des Algorithmus, den das Programm implementiert
- **Maß für “Algorithmus-Kompliziertheit”: Komplexität**
 - Rechenzeitbedarf \rightarrow Zeitkomplexität
 - Speicherplatzbedarf \rightarrow Platzkomplexität

- **Komplexität:** Maß für Kosten eines Algorithmus
- **Bedeutung:** Komplexität hängt nur vom Algorithmus ab; unabhängig von übrigen Faktoren, die Rechenzeit und Hardwarebedarf beeinflussen
- **Bestimmung der Komplexität**
 - Nicht: Messungen auf einer bestimmten Maschine
 - Sondern: Aufwandsbestimmungen für idealisierten Modellrechner (Bsp.: Random-Access-Maschine oder RAM)
 - *Abstraktes* Komplexitätsmaß zur *asymptotischen* Kostenabschätzung in Abhängigkeit von Problemgröße/Eingabegröße n

abstrakt \equiv unabhängig von konkreter Maschine (konstanter Faktor)

asymptotisch \equiv muss nur für große Probleme gelten

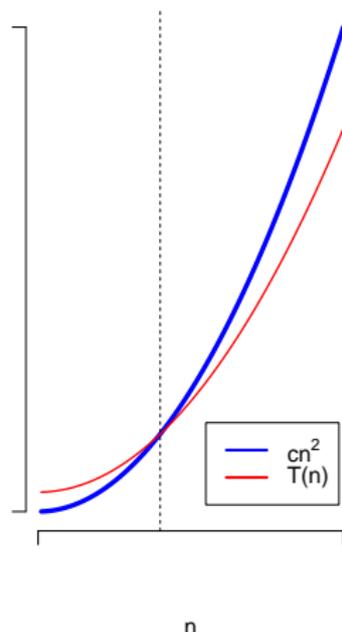
- Komplexität hängt ab
 - von **Eingabegröße**; z.B. Anzahl der Datensätze, über die gesucht werden soll
 - **außerdem**: von weiteren **Eigenschaften der Eingabe**; z.B. Reihenfolge der Datensätze (unsortiert – grob vorsortiert – sortiert)
- Meist Abschätzung der worst case Komplexität, d.h. unter Annahme der für den Algorithmus ungünstigsten Eingabe der jeweiligen Größe (z.B. bestimmte Reihenfolge)
- Analyse der *average case* Komplexität oft viel schwerer; *best case* weniger interessant.

- Meist Abschätzung oberer Schranken: **Groß-Oh-Notation**

Beispiel: Die asymptotische Komplexität $T(n)$ eines Algorithmus ist durch die Funktion $f(n) = n^2$ nach oben beschränkt, wenn es Konstanten n_0 und $c > 0$ gibt, so daß für alle Werte von $n > n_0$ gilt:

$$T(n) \leq c \cdot n^2$$

man sagt “ $T(n)$ ist in $O(n^2)$ ”, in Zeichen $T(n) \in O(n^2)$ oder einfach $T(n) = O(n^2)$ (oder im Informatikerslang: “der Algo hat quadratische Komplexität”).



Wir definieren eine Menge $O(f)$ aller Funktionen der Größenordnung f , d.h. aller Funktionen die durch f nach oben beschränkt sind.

Definition: Die Klasse $O(f)$ der Funktionen von der Größenordnung f ist

$$O(f) = \{g \mid \exists c > 0 \exists n_0 > 0 : \forall n \geq n_0 : g(n) \leq cf(n)\}$$

$$O(f) = \{g \mid \exists c > 0 \exists n_0 > 0 : \forall n \geq n_0 : g(n) \leq cf(n)\}$$

Beispiel. $T(n) = 6n^4 + 3n^2 - 7n + 42 \log n + \sin \cos(2n)$

Behauptung: $T(n) \in O(n^4)$

Beweis: Für $n \geq 1$ gilt: $n^4 \geq n^3 \geq n^2 \geq n \geq \log n$ und $n^4 \geq 1 \geq \sin(\text{irgendwas})$. Also ist

$(6 + 3 + 7 + 42 + 1)n^4$ auf jeden Fall größer als $T(n)$.

Damit folgt die Behauptung aus $\forall n \geq 1 : T(n) < 59f(n)$. □

Alles klar?



Dann: Warum gilt

$g(n) \in O(n) \Rightarrow g(n) \in O(n \log n) \Rightarrow g(n) \in O(n^2)?$

So wie $T(n) \in O(f)$ eine obere Schranke ausdrückt, beschreibt $T(n) \in \Omega(f)$ eine untere Schranke.

$g \in \Omega(f)$ bedeutet, dass g (asymptotisch und in der Größenordnung) mindestens so stark wächst wie f .

Definition:

$$\Omega(f) = \{h \mid \exists c > 0 \exists n_0 > 0 : \forall n > n_0 : h(n) \geq cf(n)\}$$

Exakte Schranke (d.h., nach oben *und* unten)

Gilt sowohl $g \in O(f)$ als auch $g \in \Omega(f)$ schreibt man $g \in \Theta(f)$.

Also:

$g \in \Theta(f)$ bedeutet: die Funktion g verläuft für hinreichend große n im Bereich $[c_1 f, c_2 f]$ mit geeigneten Konstanten c_1 und c_2 .

[Formale Definition zur Übung analog zu $O(f)$ und $\Omega(f)$]

Satz: Ist $T(n)$ ein Polynom vom Grad p , dann ist $T(n) \in \Theta(n^p)$

Wachstumsordnung = höchste Potenz

Satz: Ist $T(n)$ ein Polynom vom Grad p , dann ist $T(n) \in \Theta(n^p)$

Wachstumsordnung = höchste Potenz

① $T(n) = c_1 n^p + c_2 n^{p-1} + \dots + c_{p+1} n^0$

② $T(n) \in O(n^p)$

③ $T(n) \in \Omega(n^p)$

$\Rightarrow T(n) \in \Theta(n^p)$

Wichtige Wachstumsfunktionen

$O(1)$ konstante Kosten

$O(\log n)$ logarithmisches Wachstum

$O(n)$ lineares Wachstum

$O(n \log n)$ $n \log n$ -Wachstum

$O(n^2)$ quadratisches Wachstum

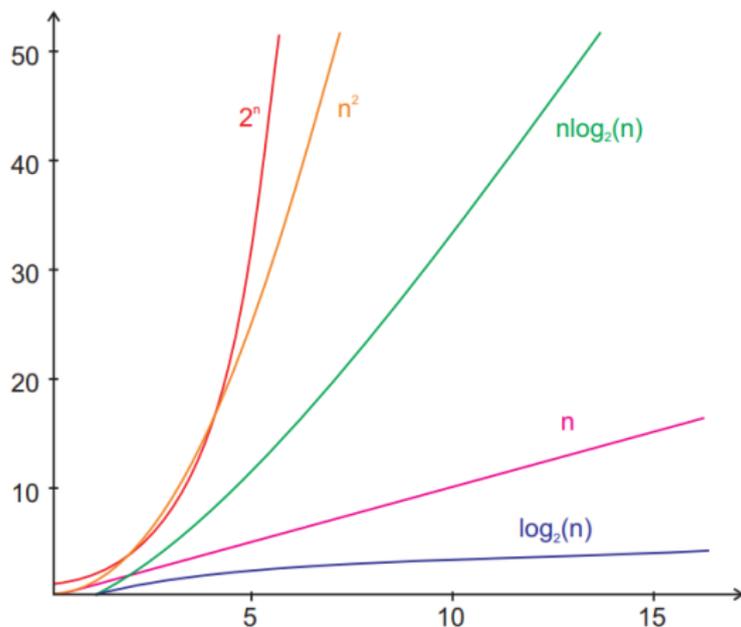
$O(n^3)$ kubisches Wachstum

$O(2^n)$ exponentielles Wachstum

$O(n!)$ Wachstum der Fakultät

$$n! = 1 \cdot 2 \cdot 3 \cdot \dots \cdot (n-1) \cdot n$$

Vergleich der Wachstumsfunktionen



Das asymptotische Wachstumsverhalten ist unabhängig von einer multiplikativen Konstante!

Problemgröße bei vorgegebener Zeit

Annahme $n = 1$ in 1ms ...

	1 Sekunde	1 Minute	1 Stunde
$\log_2 n$	2^{1000}	2^{60000}	$2^{3600000}$
n	1000	60000	3600000
$n \log_2 n$	140	4893	20000
n^2	31	244	1897
n^3	10	39	153
2^n	9	15	21

Welche Problemgröße kann bei verschiedenen Kostenfunktionen mit Rechnern verschiedener Geschwindigkeit bearbeitet werden?

Zeitbedarf $T(n)$ für Rechner #1 bei gegebener Rechengeschwindigkeit, $T_k(n)$ bei Rechner #2 mit k -facher Geschwindigkeit

Bei gleicher Rechenzeit: $T(n) = kT_k(n)$

Alternativ:

Rechner #2 löst in bestimmter Zeit ein Problem der Größe n_k . Wie groß ist diese Zahl verglichen mit n ?

$$T(n) = T_k(n_k) = T(n_k)/k$$

$$\text{Lösung: } n_k = T^{-1}(k(T(n)))$$

Beispiele: (1) $T(n) = n$, (2) $T(n) = n^2$, (3) $T(n) = 2^n$

- linear-beschränkt, $T \in O(n)$
- polynomial-beschränkt, $T \in O(n^k)$
- exponentiell-beschränkt, $T \in O(\alpha^n)$

Exponentiell-zeitbeschränkte Algorithmen sind im Allgemeinen, d.h. für große n , nicht nutzbar. (Vergleiche das gerade betrachtete Beispiel $T(n) = 2^n$)

Probleme, für die kein polynomial-zeitbeschränkter Algorithmus existiert, heißen deshalb *intractable* (\approx “unlösbar”, hart).

(umgekehrt: polynomial-beschränkt \equiv effizient / effizient lösbar)

Rechenregeln für die Zeitkomplexität I:

Mit folgenden Regeln können sie die Komplexität eines Algorithmus (nach oben) abschätzen:

- **Elementare Operationen:** $O(1)$
z.B. Zuweisungen, 32/64bit-Arithmetik, Ein-/Ausgabe, etc.
- **Fallunterscheidungen** $O(1)$
- **Schleife** Produkt aus Anzahl der Schleifendurchläufe mit Kosten der teuersten Schleifenausführung
- **Summenregel:** Das Programm P führe die Programmteile P_1 und P_2 hintereinander aus. Die Komplexitäten von P_1 und P_2 seien $T_1(n) \in O(f(n))$ und $T_2(n) \in O(g(n))$. Die Komplexität von P wird berechnet nach der Summenregel

$$T_1(n) + T_2(n) \in O(\max\{f(n), g(n)\}).$$

Beispiel:

```
x=0;                               /* P1 */
for (i=1; i<=n; i++) { x=x+i; }    /* P2 */
```

- **Produktregel:** Das Programm P entstehe durch Einsetzen von P_1 in der innersten Schleife von P_2 . Die Komplexitäten von P_1 und P_2 seien $T_1(n) \in O(f(n))$ und $T_2(n) \in O(g(n))$. Die Komplexität von P wird berechnet nach der Produktregel

$$T_1(n) \cdot T_2(n) \in O(f(n)g(n)).$$

Beispiel:

```
for (i=1; i<=n; i++) {  
    for (j=1; j<=n; j++) {  
        x=x+i*j;  
    }  
}
```

Rechenregeln für die Zeitkomplexität III:

- **Rekursive Prozeduraufrufe:** Produkt aus Anzahl der rekursiven Aufrufe mit Kosten der teuersten Prozedurausführung.

- **Beispiele:**

```
① int sum(n) {  
    if (n<=1) return 1;  
    return n+sum(n-1);  
}
```

```
② int sumsum(n) {  
    if (n==0) return 1;  
    return sum(n)+sumsum(n-1);  
}
```



Schätzen sie die Komplexitäten von $\text{sum}(n)$ und $\text{sumsum}(n)$ mit Hilfe der Rechenregeln ab.

Maximale Teilsumme: Problemstellung

Gegeben: Folge F von n ganzen Zahlen.

Gesucht: Teilfolge von $1 \leq i \leq n$ aufeinander folgenden Zahlen in F , deren Summe maximal ist.

Anwendungsbeispiel: Entwicklung von Aktienkursen (tägliche Änderung des Kurses). Maximale Teilsumme bestimmt optimales Ergebnis.

Tag	1	2	3	4	5	6	7	8	9	10
Δ	+5	-6	+4	+2	-5	+7	-2	-7	+3	+5

Maximale Teilsumme: Algorithmus

Tag	1	2	3	4	5	6	7	8	9	10
Δ	+5	-6	+4	+2	-5	+7	-2	-7	+3	+5

```
int maxSubSum( int[] a) {
    int maxSum = 0;
    for( i=0; i<a.length; i++)
        for( j=i; j<a.length; j++) {
            int thisSum =0;
            for (int k = i; k<=j; k++)
                thisSum += a[k];
            if(thisSum>maxSum) maxSum=thisSum;
        }
    return maxSum;
}
```

Maximale Teilsumme: Analyse

- $n = a.length$
- Innerste Schleife `for (k=i; k<=j; k++)`
 $j - i + 1$ mal durchlaufen für jedes i, j .
- Mittlere Schleife `for (j=i; j<n; j++)`
jeweils $j - i + 1$ Aktionen
→ $1 + 2 + 3 + \dots + n - i = (n - i)(n - i + 1)/2$ Aktionen
- äußere Schleife `for (i=0; i<n; i++)`
aufsummieren über den Aufwand des Durchlaufes für jedes i
- Beispiel $n = 32$: 5984 Additionen

$$\begin{aligned} T(n) &= \sum_{i=0}^{n-1} \sum_{j=i}^{n-1} \sum_{k=i}^j 1 = \sum_{i=0}^{n-1} \sum_{j=i}^{n-1} (j - i + 1) \\ &= \sum_{i=0}^{n-1} (n - i)(n - i + 1)/2 = \sum_{k=1}^n k(k + 1)/2 = n^3/6 + n^2/2 + n/3 \end{aligned}$$

$$\sum_{k=1}^n k^2 = n^3/3 + n^2/2 + n/6 \text{ (wie sieht man das?)}$$

$$\sum_{k=1}^n k^2 = n^3/3 + n^2/2 + n/6 = 1/6(n)(n+1)(2n+1)$$

(i) $n = 1$: $\sum_{k=1}^1 k^2 = 1 = 1/3 + 1/2 + 1/6$

(ii) $n - 1 \rightarrow n$:

① $\sum_{k=1}^{n-1} k^2 + n^2 = 1/6(n-1)(n)(2n-1) + 6n^2/6$

② $= 1/6(2n^3 - n^2 - 2n^2 + n) + 6n^2/6$

③ $= n^3/3 + n^2/2 + n/6$

“Versteckte” Zeitkomplexität

Beispiel: Fibonacci-Zahlen 0, 1, 1, 2, 3, 5, 8, 13, 21, ...

Definition $F_0 = 0$, $F_1 = 1$, $F_n = F_{n-1} + F_{n-2}$ für $n > 1$.

```
❶ int fibE (int n) {  
    if (n <= 0) return 0;  
    else if (n ==1) return 1;  
    else return fibE(n-2) + fibE(n-1)  
}
```

fibE(n) berechnet F_n , aber **exponentieller Aufwand!**

```
❷ int fibL (int n, int f1, int f2) {  
    if (n <=0) return 0;  
    else if (n==1) return f1+f2;  
    else return fibL(n-1,f2+f1,f1)  
}
```

fibL(n,1,0) berechnet F_n mit **linearem Aufwand**

z.B. fibL(4,1,0)=fibL(3,1,1)=fibL(2,2,1)=fibL(1,3,2)=5

“Versteckte” Platzkomplexität am Beispiel $n!$

```
int fakL(int n) {  
    if(n<=1) return 1;  
    else return n*fakL(n-1);  
}
```

fakL(n) $\rightarrow n!$ mit **linearem Speicheraufwand** $\Theta(n)$

```
int fakK(int n,int a) {  
    if(n<=1) return a;  
    else return fakK(n-1,n*a);  
}
```

fakK($n, 1$) $\rightarrow n!$ mit **konstantem Speicheraufwand** $\Theta(1)$



Bessere Abschätzung: Wie wächst der Aufwand bei der Multiplikation großer Zahlen?



Was ist der Speicheraufwand für n und für $n!$?

Multiplikation für Fortgeschrittene

Wie in der Schule: $5432 \cdot 1995 \implies O(\ell^2)$ für ℓ -stellige Zahlen.

$$\begin{array}{r} 5432 \\ 48888 \\ 48888 \\ 27160 \\ \hline 10836840 \end{array}$$

Besser:

$$(100A + B) \times (100C + D) = 10000AC + 100(AD + BC) + BD$$

Nach $AD + BC = (A + B) \times (C + D) - AC - BD$, nur noch 3 Multiplikationen von Zahlen der halben Länge:

$$T(n) = 3T(n/2),$$

wobei der Aufwand für die Addition vernachlässigt wird.

Lösung: $T(n) = n^{\log_2 3} \approx n^{1.585} \ll n^2$

Für *große* Zahlen geht's also intelligenter als in der Schule!

- Allgemeines Theorem zur Lösung von Funktionalgleichungen (Rekursionsgleichungen) der Form

$$T(n) = aT\left(\frac{n}{b}\right) + g(n), \quad a \geq 1, b > 1$$



Warum können wir i.d.R. $T(1) = 1$ annehmen?

- Funktionalgleichung beschreibt *algorithmische Strategie* "Divide-and-Conquer":
 - Zerlege Gesamtproblem in gleich große Teilprobleme der Größe n/b .
 - Für Gesamtproblem löse a solche Teilprobleme.
 - Für Zerlegung und Kombination der Teillösungen entstehen jeweils Overhead-Kosten $g(n)$.

- Es gibt mehrere Lösungen je nach Verhalten von $g(n)$.
- Sei jetzt $g(n)$ polynomial, d.h. $g(n) = \Theta(n^k)$:

$$T(n) = aT\left(\frac{n}{b}\right) + \Theta(n^k), \quad a \geq 1, b > 1$$

- Dann unterscheide 3 Fälle:

$$T(n) = \begin{cases} \Theta(n^k) & \text{falls } a < b^k \\ \Theta(n^k \log n) & \text{falls } a = b^k \\ \Theta(n^{\log_b(a)}) & \text{falls } a > b^k \end{cases}$$

Das Mastertheorem — Beispiele

Sei wieder $g(n)$ polynomial, $g(n) = \Theta(n^k)$.

Setze $k = 2$ und $b = 3$ und betrachte Beispiele für $a \left\{ \begin{array}{l} \leq \\ = \\ > \end{array} \right\} n^k$.

$$a = 8 \quad T(n) = 8T\left(\frac{n}{3}\right) + \Theta(n^2) \quad \Rightarrow \quad T(n) = \Theta(n^2)$$

$$a = 9 \quad T(n) = 9T\left(\frac{n}{3}\right) + \Theta(n^2) \quad \Rightarrow \quad T(n) = \Theta(n^2 \log_2 n)$$

$$a = 10 \quad T(n) = 10T\left(\frac{n}{3}\right) + \Theta(n^2) \quad \Rightarrow \quad T(n) = \Theta(n^{\log_3 10})$$

Zur Erinnerung: $T(n) = \begin{cases} \Theta(n^k) & \text{falls } a < b^k \\ \Theta(n^k \log n) & \text{falls } a = b^k \\ \Theta(n^{\log_b(a)}) & \text{falls } a > b^k \end{cases}$

- Komplexität / Effizienz
= wesentliche Eigenschaft von Algorithmen
- meist asymptotische Worst-Case-Abschätzung in Bezug auf Problemgröße n
 - Unabhängigkeit von konkreten Umgebungsparametern (Hardware, Betriebssystem, ...)
 - asymptotisch “schlechte” Verfahren können bei kleiner Problemgröße ausreichen
- wichtige Klassen: $O(1)$, $O(\log n)$, $O(n)$, $O(n \log n)$, $O(n^2)$, \dots , $O(2^n)$
- zu gegebener Problemstellung gibt es oft Algorithmen mit stark unterschiedlicher Komplexität
 - unterschiedliche Lösungsstrategien
 - Raum/Zeit- “Tradeoff”: Zwischenspeichern von Ergebnissen statt mehrfacher Berechnung
- Abschätzung der Komplexität aus Programmfragmenten nach Rechenregeln