Algorithmen und Datenstrukturen 1

1. Vorlesung

Martin Middendorf und Peter F. Stadler

Universität Leipzig
Institut für Informatik
middendorf@informatik.uni-leipzig.de, studla@bioinf.uni-leipzig.de

aufbauend auf den Kursen der letzten Jahre von

E. Rahm, G. Heyer, G. Brewka ,U. Quasthoff, R. Der

Übungen

Übungen finden (meist) im 2-Wochenrhythmus statt (**A-** bzw. **B-Woche**)

Es gibt 6 Übungsgruppen (Di. und Do.)

Zeit und Ort siehe auch: **http://www.bioinf.uni-leipzig.de** → **Teaching**

Ausgabe	**Abgabe**	**A Gruppen**	**B Gruppen**
23.10.	30.10.	06.11.+ 06.11.	12.11.+13.11.
30.10.	13.11.	18.11.+ 20.11.	25.11.+27.11.
13.11.	27.11.	04.12.+ 04.12.	09.12.+11.12.
27.12.	11.12.	16.12.+ 18.12.	13.01.+15.01.
18.12.	15.01.	20.01.+ 22.01.	27.01.+29.01.

Beachte für A-Gruppen Dienstag:

- statt Di. 04.11. → Ersatz Do. 06.11., 17:15, Raum 109, Härtelstr.
- statt Di. 02.12. → Ersatz Do. 04.12., 17:15, Raum 109, Härtelstr.

Übungsgruppen

Übungsgruppenleiter: Dipl.-Inf. Matthias Bernt

Dr. Sonja Prohaska

Dr. Stephan Steigele

Mögliche Übungsgruppen:

Gruppe A:	Di. 11.15 - 12.45 Uhr	Woche A	Raum HS R109
Gruppe B:	Di. 11.15 - 12.45 Uhr	Woche B	Raum HS R109
Gruppe C:	Di. 16.30 - 18.00 Uhr	Woche A	Raum HS R109
Gruppe D:	Di. 16.30 - 18.00 Uhr	Woche B	Raum HS R109
Gruppe E:	Do.10.15 - 11.45 Uhr	Woche A	Raum HS R109
Gruppe F:	Do.10.15 - 11.45 Uhr	Woche B	Raum HS R109
Gruppe G:	Fr. 09.15 - 10.45 Uhr	Woche A	Raum SG R512
Gruppe H:	Fr. 09.15 - 10.45 Uhr	Woche B	Raum SG R512

SG=Seminargebäude Brühl, HS=Härtelstraße

Anmeldung für die Übungsgruppen

Anmeldung für die Übungsgruppen unter

http://www.bioinf.uni-leipzig.de

von jetzt bis zum 19.10.08 um 24.00 Uhr

Übungsaufgaben

- Es sind fünf Serien von Übungsaufgaben zu bearbeiten
- Diese finden sich (meist) im Zweiwochenrhythmus auf den Seiten im Netz.
  http://www.bioinf.uni-leipzig.de/Leere/FOLIEN/ADS1/ADS1.html
- Bearbeitungszeit (meistens) 2 Wochen. **Abgabe vor der Vorlesung hier**!

Leistungsbewertung

- Modulabschlussprüfung: Klausur 60‘

Klausur am Do. 5.2.2009 um 16.15 Uhr

- Voraussetzung für Klausurteilnahme: mind. **50%** der Punkte in den Übungsaufgaben erreicht

Inhaltsverzeichnis

Literatur

T. Ottmann, P. Widmayer: Algorithmen und Datenstrukturen, Reihe Informatik, Band 70, BI-Wissenschaftsverlag, 4. Auflage, Spektrum-Verlag, 2002

M.A. Weiss: Data Structures & Algorithm Analysis in Java. Addison-Wesley 2. Auflage, 2007

Weitere Bücher

V. Claus, A. Schwill: Duden Informatik, BI-Dudenverlag, 2. Auflage 1993

D.A. Knuth: The Art of Computer Programming, Vol. 3, Addison-Wesley, 1973

R. Sedgewick: Algorithmen. Addison-Wesley 1992

G. Saake, K. Sattler: Algorithmen und Datenstrukturen - Eine Einführung mit Java. dpunkt-Verlag, 3. Auflage, 2006

A. Solymosi, U. Gude: Grundkurs Algorithmen und Datenstrukturen In Java. Eine Einführung in die praktische Informatik. Vieweg, 2000, 4. Auflage 2008

Einführung

Algorithmus: Verfahren, das

- durch einen endlichen Text beschreibbar ist
- dessen einzelne Schritte tatsächlich ausführbar sind
- in jedem Schritt nur endlich viel Speicherplatz benötigt
- nach endlich vielen Schritten terminiert

→ Algorithmen stehen im Mittelpunkt der Informatik

Wesentliche Entwurfsziele bei Entwicklung von Algorithmen:

- Korrektheit
- Terminierung
- Effizienz ←←←←← unser Schwerpunkt

  → Wahl der Datenstrukturen v.a. für Effizienz entscheidend

Vorlesungsschwerpunkte:

- Entwurf von effizienten Algorithmen und Datenstrukturen
- Analyse ihres Verhaltens

Wozu Gedanken über Datenstrukturen?

Algorithmen verarbeiten Daten → die Verarbeitung soll **effektiv** sein

Mögliche Aufgaben :

- einen gesuchten Datensatz ermitteln (z. B. Nachschlagen im Telefonbuch)
- Hinzufügen und Löschen von Datensätzen
- Sortieren eines Datenbestandes

Beobachtung: Funktional gleichwertige Algorithmen weisen oft erhebliche Unterschiede in der **Effizienz** (Komplexität) auf

Die Effizienz hängt – insbesondere bei großen Datenmengen - ab von

- der internen Darstellung der Daten
- dem verwendeten Algorithmus

→ somit hängen beide Teilaspekte oft eng zusammen

Beispiel: Telefon-CD

Speicherplatz auf der CD-ROM: ca. 700 MB

Suchmöglichkeiten: kombiniert nach Name und Ort

Abschätzung des Rohdatenvolumens:

- 40 Millionen Einträge zu je 35 Zeichen, d.h. ca. 1.4 GB ASCII-Text

ALSO: Wir brauchen eine Datenstruktur,

- die einen schnellen Zugriff (über einen sog. Index) erlaubt und
- zusätzlich die Daten komprimiert

Außerdem gilt es zu beachten, dass der Zugriff auf die CD-ROM wesentlich länger dauert als ein Zugriff auf die Festplatte.

Beobachtungen zur Telefon-CD

Zum Nachschlagen könnte man ein Programm benutzen, welches die vollständige Verwaltung der Einträge erlaubt: Suchen, Einfügen und Löschen. Die zusätzlich vorhandenen Funktionen werden einfach nicht angeboten.

Aber: Weil sowieso nur das Suchen erlaubt ist, können wir vielleicht eine Datenstruktur verwenden, die zwar extrem schnelles Suchen in einer komprimierten Datei erlaubt, aber möglicherweise kein Einfügen.

Schlussfolgerung: Soll unsere Datenstruktur nur bestimmte Operationen erlauben, so lohnt die Suche nach entsprechenden Strukturen mit hoher Effizienz.

Effizienz: Zeit und Speicher

Die Abarbeitung von Programmen (Software) beansprucht 2 **Ressourcen**: Zeit und Hardware (wichtig: Speicher)

Wie steigt dieser Ressourcenverbrauch bei größeren Problemen (d.h. bei mehr Eingabedaten)?

Es kann sein, dass Probleme ab einer gewissen Größe praktisch unlösbar sind, weil

- ihre Abarbeitung zu lange dauern würde (z.B. >100 Jahre) oder
- das Programm mehr Speicher braucht, als zur Verfügung steht.

Wichtig ist der Unterschied zwischen **RAM** und **externem Speicher**, da der Zugriff auf eine Festplatte ca. 100.000 mal langsamer ist als ein RAM-Zugriff (wenige Millisekunden vs. wenige Nanosekunden)

→ deshalb werden manche Algorithmen bei Überschreiten des RAM so langsam, dass sie praktisch nutzlos sind

Komplexität von Algorithmen

Wesentliche Maße:

- Rechenzeitbedarf (Zeitkomplexität)
- Speicherplatzbedarf (Speicherplatzkomplexität)

Programmlaufzeit ist von zahlreichen Faktoren abhängig

- Eingabe für das Programm
- Qualität des vom Compiler generierten Codes und des gebundenen Objektprogramms
- Leistungsfähigkeit der Maschineninstruktionen, mit deren Hilfe das Programm ausgeführt wird
- Zeitkomplexität des Algorithmus, der durch das ausgeführte Programm verkörpert wird

Bestimmung der Komplexität

- **Messungen** auf einer bestimmten Maschine
- Aufwandsbestimmungen für **idealisierten Modellrechner** (Bsp.: Random-Access-Maschine oder RAM)
- Abstraktes Komplexitätsmaß zur **asymptotischen Kostenschätzung** in Abhängigkeit zur Problemgröße (Eingabegröße) n

Asymptotische Kostenmaße

Festlegung der Größenordnung der Komplexität in Abhängigkeit der Eingabegröße:
Best Case, **Worst Case**, **Average Case**

Meist Abschätzung **oberer Schranken** (Worst Case): ***Groß-Oh-Notation***

Definition: Zeitkomplexität T(n) eines Algorithmus ist von der Größenordnung n, wenn es Konstanten n_0 und $c > 0$ gibt, so dass für alle Werte $n > n_0$ gilt:

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

man sagt "T(n) ist in O(n)" bzw. "$T(n) \in O(n)$" oder "T(n) = O(n)"
(geprochen „ … Groß-O von n“)

Allgemeine Definition: Groß-Oh-Notation

Definition: Klasse der Funktionen O(f), die zu einer Funktion (Größenordnung) f gehören ist

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

Beispiel: $6n^4 + 3n^3 - 7\ n \in O(n^4)$

Zu zeigen: $6n^4 + 3n^3 - 7 \leq c\ n^4$ für ein c und alle $n \geq n_0$

→ $6 + 3/n - 7 / n^4 \leq c$

Wähle also z.B. $c = 9$, $n_0 = 1$

Ein Programm, dessen Laufzeit oder Speicherplatzbedarf in O(f(n)) ist, hat demnach eine Wachstumsrate ≤ f(n)

Beispiel: Ist $g(n) = O(n \log n)$ dann folgt $g(n) = O(n^2)$ (wegen $\log n \leq n$)

Jedoch gilt natürlich $O(n \log n) \neq O(n^2)$

Groß-Omega-Notation

Untere Schranke: $g \in \Omega(f)$ oder $g = \Omega(f)$ drückt aus, dass g mindestens so stark wächst wie f

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

Alternative Definition (u.a. Ottmann/Widmayer):

$\Omega(f) = \{h \mid \exists\, c > 0 : \exists$ unendlich viele n: $h(n) \geq c\, f(n)\}$

Exakte Schranke

Definition: Gilt für eine Funktion g sowohl $g \in O(f)$ als auch $g \in \Omega(f)$, so schreibt man $g \in \Theta(f)$

g aus $\Theta(f)$ bedeutet also: die Funktion g verläuft ab einem Anfangswert n_0 im Bereich $[c_1 f, c_2 f]$ für geeignete Konstanten c_1, c_2

Merke: Ist T(n) ein Polynom vom Grade p dann ist $T(n) \in \Theta(n^p)$, d.h. die Wachtsumsordnung ist durch die höchste Potenz im Polynom gegeben

Polynomiales Wachstum

Ist T(n) ein Polynom der Ordnung p dann gilt: $T(n) = \Theta(n^p)$

Beweis $T(n) \in O(n^p)$:

Sei $T(n) = \sum_{k=0}^{p} a_k n^k = a_p n^p + \ldots + a_0 . \; n^p$

Ausklammem liefert $T(n) = n^p U(n)$

mit $U(n) = a_p + a_{p-1} \frac{1}{n} + \ldots + a_0 \frac{1}{n^p}$ und $\lim_{n \to \infty} U(n) = a_p$

Für ein beliebiges $\varepsilon > 0$ wählen wir $c = a_p + \varepsilon$

Es läßt sich damit immer ein $n_0 > 0$ finden so dass

$T(n) < cn^p \;\; \forall \; n > n_o .$

Beweis $T(n) \in \Omega(n^p)$: analog

Wichtige Wachstumsfunktionen

Kostenfunktionen

$O(1)$	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

Wachstumsverhalten

Ressourcenverbrauch in Anhängigkeit von der Problemgröße n bei verschiedenen Kostenfunktionen

$\log n$	3	7	10	13	17	20
$\sqrt{n}$	3	10	30	100	300	1000
n	**10**	**100**	**1000**	$\mathbf{10^4}$	$\mathbf{10^5}$	$\mathbf{10^6}$
$n \log n$	30	700	10^4	10^5	$2 \cdot 10^6$	$2 \cdot 10^7$
n^2	100	10^4	10^6	10^8	10^{10}	10^{12}
n^3	1000	10^6	10^9	10^{12}	10^{15}	10^{18}
2^n	1000	10^{30}	10^{300}	10^{3000}	10^{30000}	10^{300000}

Problemgröße bei vorgegebener Zeit

Welche Problemgröße kann bei verschiedenen Kostenfunktionen in vorgegebener Zeit bearbeitet werden?

Komplexität	1 sec	1 min	1 h
$\log_2 n$	2^{1000}	2^{60000}	-
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

Annahme: Zeit für Problem der Größe n = 1 ist 1 ms.

Problemgröße in Abhängigkeit der Rechengeschwindigkeit

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

Für Problem der Größe n seien die Rechengeschwindigkeiten

Rechner 1: $T(n)$

Rechner 2: $T_K(n)$ (K-fache Geschwindigkeit)

also $T_K(n) = T(n)/K$

Ann.: Bei gleicher Rechenzeit löst Rechner 2 Problem der Größe n_K

Es gilt also $T(n) = T_K(n_K)$ und $KT(n) = T(n_K)$

Explizite Lösung: $n_K = T^{-1}(KT(n))$

Problemgröße und Rechnergeschwindigkeit

Gesuchte Problemgröße n_K ergibt sich also aus $KT(n) = T(n_K)$

Beispiel 1: $T(n) = n^m$ so dass $Kn^m = {n_K}^m$ Lösung: $n_K = K^{\frac{1}{m}} n = \sqrt[m]{K}\ n$

Rechenbeispiel: Für m = 3 gilt, dass Faktor K = 1000 in der Rechengeschwindigkeit nur Faktor 10 in der Problemgröße bringt.

Beispiel 2: $T(n) = 2^n$ so dass $K2^n = 2^{n_K}$ Lösung: $\mathrm{n}^{\mathrm{K}} = n + \log_2 K$

Rechenbeispiel: Faktor K = 1000 in der Rechengeschwindigkeit bringt nur Summand $\log_2(1000) \approx 10$ in der Problemgröße.

Problemgröße in Abhängigkeit der Rechengeschwindigkeit

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

Problemkomplexität	aktuelle Rechner	Rechner 100x schneller	1000x schneller
n	N_1	$100\ N_1$	$1000\ N_1$
n^2	N_1	$10\ N_2$	$32\ N_2$
n^3	N_3	$4.6\ N_3$	$10\ N_3$
n^5	N_4	$2.5\ N_4$	$4\ N_4$
2^n	N_5	$N_1 + 7$	$N_1 + 10$
3^n	N_6	$N_6 + 4$	$N_6 + 6$

Leistungsverhalten bei kleiner Eingabegröße

Asymptotische Komplexität (im Sinne von Groß-O) ist (vor allem) relevant

für große n.

Bei kleinen Problemgrößen haben die Faktoren (c und K) einen wesentlichen Einfluss

→ Verfahren mit besserer (asymptotischer) Komplexität kann schlechter abschneiden als Verfahren mit schlechter (asymptotischer) Komplexität

Alg.	T(n)	Bereiche von n mit günstigster Zeitkomplexität
A_1	$186182 \log_2 n$	$n > 2048$
A_2	$1000\, n$	$1024 \leq n \leq 2048$
A_3	$100\, n \log_2 n$	$59 \leq n \leq 1024$
A_4	$10\, n^2$	$10 \leq n \leq 58$
A_5	n^3	$n = 10$
A_6	2^n	$2 \leq n \leq 9$

Zusammenfassung erste Vorlesung

O-Notation (obere Schranke)

Klasse der Funktionen O(f), die zu einer Funktion (Größenordnung) f gehören ist

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

– die Funktion f(n) majorisiert g(n)

Beispiel: $6n^4 + 3n^3 - 7n \in O(n^4)$

Ω -Notation (untere Schranke)

Klasse der Funktionen Ω(f), die zu einer Funktion (Größenordnung) f gehören ist

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

$g \in \Omega(f)$ oder $g = \Omega(f)$ drückt aus, dass g mindestens so stark wächst wie f.

– f(n) minorisiert g(n)

Θ-Notation (Exakte Schranke)

Gilt für eine Funktion g sowohl $g \in O(f)$ als auch $g \in \Omega(f)$, so schreibt man $g \in \Theta(f)$

Zeitkomplexitätsklassen

Drei zentrale Zeitkomplexitätsklassen werden unterschieden

Algorithmus A mit Zeitkomplexität T(n) heißt:

- **linear-zeitbeschränkt**, falls $T(n) \in O(n)$
- **polynomial-zeitbeschränkt**, falls $T(n) \in O(n^k)$
- **exponentiell-zeitbeschränkt**, falls $T(n) \in O(k^n)$

Exponentiell-zeitbeschränkte Algorithmen im allgemeinen (größere n) nicht nutzbar.

Probleme, für die kein polynomial-zeitbeschränkter Algorithmus existiert, gelten als (**in der Praxis**) **unlösbar** (**intractable**).

Berechnung der Zeitkomplexität I

Elementare Operationen (Zuweisung, Ein-/Ausgabe, elementare Rechenoperationen): Zeit O (1)

T_1 und T_2 seien die Laufzeiten zweier Programmfragmente P_1 und P_2; es gelte

$T_1(n) \in O(f(n))$ und $T_2(n) \in O(g(n))$

Summenregel: Für die Hintereinanderausführung von P_1 und P_2 ist

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

Produktregel: (z.B.) Für die geschachtelte Schleifenausführung von P_1 und P_2 ist

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

Berechnung der Zeitkomplexität II

Fallunterscheidung: Kosten der Bedingungsanweisung (= O(1)) + Kosten der längsten Alternative

Schleife: Produkt aus Anzahl der Schleifendurchläufe mit Kosten der teuersten Schleifenausführung

rekursive Prozeduraufrufe: Produkt aus Anzahl der rekursiven Aufrufe mit Kosten der teuersten Prozedurausführung

Beispiel: Berechnung der maximalen Teilsumme

Gegeben: Folge F von n ganzen Zahlen

Gesucht: Teilfolge von 0 <= i <= 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
Gewinn/Verlust (Folge)	+5	-6	+4	+2	-5	+7	-2	-7	+3	+5

1. Lösungsmöglichkeit:

(bessere Lösungsmöglich-keiten folgen später)

```
int maxSubSum1( int [ ] a) {
    int maxSum = 0; // leere Folge
    for ( int i = 0; i < a.length; i++ )
        for ( int 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;
}
```

Komplexitätsbestimmung I

Nenne a.length = n

Der Schleifenkörper der innersten for-Schleife

for (k = i; i <= j; k++)

wird j - i + 1 mal durchlaufen und dabei jeweils die Aktion (Addition) ausgeführt.

Für den nächstinneren Schleifenkörper

for (j = i; j < n; j++) {jeweils j - i + 1 Aktionen}

ergeben sich 1 + 2 + 3 + ... + n-i Aktionen.

In der Summe sind das (Gaussche Formel) (n-i)(n-i+1)/2 Aktionen

Die äußere Schleife entspricht dann der Aufsummation aller dieser Beiträge über i von i = 0 bis i = n – 1

→ insgesamt ist die Gesamtzahl der Aktionen $= n^3/6 + n^2/2 + n/3$

Beispiel: Für ***n=32*** ergeben sich 5984 **Additionen**

Komplexitätsbestimmung II

Ein Schleifendurchlauf innen $= O(1)$

Gesamtzahl der Durchläufe:

$$T(n) = \sum_{i=0}^{n-1} \sum_{j=i}^{n-1} \sum_{k=i}^{j} 1 = \sum_{i=0}^{n-1} \sum_{j=i}^{n-1} (j - i + 1) = \sum_{i=0}^{n-1} \sum_{l=1}^{n-i} l$$

$$= \sum_{i=0}^{n-1} \frac{(n-i)(n-i+1)}{2} = \sum_{k=1}^{n} \frac{k(k+1)}{2}$$

Mit $\sum_{k=1}^{n} k^2 = \frac{1}{3}n^3 + \frac{1}{2}n^2 + \frac{1}{6}n$

ergibt sich $T(n) = \frac{1}{6}n^3 + \frac{1}{2}n^2 + \frac{1}{3}n$

Rekursion vs. Iteration I

Für viele Probleme gibt es sowohl **rekursive** als auch **iterative** Lösungsmöglichkeiten

Unterschiede bezüglich

- Einfachheit, Verständlichkeit
- Zeitkomplexität
- Speicherkomplexität

Rekursion vs. Iteration II: Berechnung der Fibonacci-Zahlen

Definition

$F_0 = 0$

$F_1 = 1$

$F_n = F_{n-1} + F_{n-2}$ für $n>1$

Rekursive Lösung verursacht exponentiellen Aufwand.

Die iterative Lösung ist mit linearem Aufwand möglich.

Idee: Speicherung von Zwischenergebnissen würde die Komplexität bei der rekursiven Lösung verringern. Wie?

```
Int fibRekursiv (int n) { // erfordert n>0
   if (n <=0) return 0;
   else if (n ==1) return 1;
   else return fibRekursiv (n-2) +
   fibRekursiv (n-1)
   }

Int fibIterativ (int n) { // erfordert n>0
   if (n <=0) return 0;
   else {
   int aktuelle=1, vorherige=0, temp=1,
   for (int i=1; i<n; i++) {
        temp = aktuelle;
        aktuelle += vorherige;
        vorherige = temp;}
   return aktuelle;
   }
```

Herkunft der Fibonacci-Zahlen

Modell einer Kaninchenpopulation: $F_0 = 0, F_1 = 1, F_n = F_{n-1} + F_{n-2}$ für $n>1$

Fibonacci stieß auf diese Folge bei der einfachen mathematischen Modellierung des Wachstums einer Kaninchenpopulation nach folgender Vorschrift:

1. Zu Beginn gibt es ein Paar neugeborener Kaninchen.
2. Jedes neugeborene Kaninchenpaar wirft nach 2 Monaten ein weiteres Paar.
3. Anschließend wirft jedes Kaninchenpaar jeden Monat ein weiteres.
4. Kaninchen leben ewig und haben einen unbegrenzten Lebensraum.

- Jeden Monat kommt zu der Anzahl der Paare, die im letzten Monat gelebt haben, eine Anzahl von neugeborenen Paaren hinzu, die gleich der Anzahl der Paare ist, die bereits im vorletzten Monat gelebt haben, da genau diese geschlechtsreif sind und sich nun vermehren. Das entspricht aber gerade der oben angegebenen Rekursionsformel (**Quelle**: Wikipedia)

Rekursion vs. Iteration III: n!

```
int fakRekursiv (int n) { // erfordert n > 0
        if (n <= 1) return 1;
        else return n * fakRekursiv (n-1);
}
```

```
int fakIterativ (int n) { // erfordert n > 0
    int fak = 1;
    for (int i = 2; i <= n; i++) fak *= i;
    return fak;
}
```

Speicherkomplexität:

Die rekursive Lösung verbraucht mehr Speicher als die iterative Lösung

Für genauere Abschätzungen müssen wir wissen, ob der Aufwand für die Multiplikation großer Zahlen mit der Länge der Faktoren wächst

Zeitkomplexität – Multiplikation

Multiplikation zweier n-stelliger Zahlen

1. Schulmethode: Zeit $O(n^2)$

```
 5432 • 1995
-----------
   5432
  48888
   48888
    27160
-----------
 10836840
```

2. Besseres Verfahren: Zerlege vierstelligen Faktoren in jeweils zwei zweistellige Zahlen

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

(vier Multiplikationen mit halber Länge sind nicht besser, aber:)

$$= 10000AC + 100((A+B)(C+D)-AC-BD) + BD$$

(nur noch drei Multiplikationen!)

Wir brauchen nur drei Multiplikationen von Zahlen halber Länge

→ ergibt folgende Funktionalgleichung (Aufwand für Additionen wird dabei nicht berücksichtigt)

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

Zeitkomplexität – Multiplikation

Funktionalgleichung:

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

Lösung:

Methode 1 (iterativ): Annahme T(1)=1, $\log_2$ = Logarithmus zur Basis 2

$$T(n) = 3T(n/2) = 3 \cdot 3T(n/4) = (\prod_{i=1}^{\log_2 n} 3)T(1) = 3^{\log_2 n} = n^{\log_2 3} \approx n^{1,585}$$

Methode 2 (Induktion: Lösung „raten“ und Substituieren):

Ansatz: $T(n) = n^m$

mit $T(n) = 3T\left(\frac{n}{2}\right)$ folgt $n^m = 3\frac{n^m}{2^m}$ und damit $m = \log_2 3 \approx 1.58$

ALSO: Verbesserung von $O(n^2)$ auf $O(n^{1.585})$

Zeitkomplexität – Exponentielles Wachstum

Funktionalgleichung:

$$T(n+1) = a\,T(n)$$

Lösung:

Iteration: $T(n) = a\,T(n-1) = a^2\,T(n-2) = \ldots = a^n\,T(0)$

Schreibe a^n als Zweierpotenz, $a^n = 2^{kn}$ mit $k = \log_2 a$ so dass

$$T(n) = 2^{kn}\,T(0)$$

$k = \log_2 a$ hat die Funktion eines Skalenfaktors für die Problemgröße

Wichtige Eigenschaft: Obige Funktionalgleichung führt zu exponentiellem Wachstum

Zeitkomplexität - Fibonacci-Zahlen

Funktionalgleichung:

$$F_n = F_{n-1} + F_{n-2} \text{ für } n>1$$

mit den Anfangsbedingungen:

$F_0 = 0$

$F_1 = 1$

Binetsche Formel (1843):

$$F_n = \frac{1}{\sqrt{5}}\left(A^n - B^n\right) \text{ mit } A = \frac{1+\sqrt{5}}{2},\ B = \frac{1-\sqrt{5}}{2}$$

Schreibe das als

$$F_n = \frac{1}{\sqrt{5}} A^n \left(1 - \left(\frac{B}{A}\right)^n\right)$$

$$\left(\frac{B}{A}\right)^n = (-1)^n \left(\frac{\sqrt{5}-1}{\sqrt{5}+1}\right)^n \to 0 \text{ für } n \to \infty$$

Ergebnis: Für bel. $n_0 > 0$ findet sich immer ein c so dass $F_n < cA^n \ \forall n > n_0$

Folglich gilt $F_n \in O\left(A^n\right)$ mit $A = \frac{1+\sqrt{5}}{2} \approx 1.62$

Das Mastertheorem

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

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

Funktionalgleichung beschreibt **algorithmische Strategie**: Zerlege Problem der Größe n in b Teilprobleme

- Lösung des Gesamtproblems koste das a-fache der Lösung eines Teilproblems
- zusätzlich entstehen einmalige *overhead* Kosten g(n)

Das Mastertheorem - Polynomial

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

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

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

Das Mastertheorem – Beispiele

- Sei wieder g(n) polynomial, $g(n) = \Theta(n^k)$
  Betrachte den Fall k = 2 und b = 3:

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

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

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

Das Mastertheorem - Allgemein

Setze $u = \log_b(a)$

- Zur Erinnerung: Falls $g(n) = 0$ dann ist $T(n) = \Theta(n^u)$ → vgl. Analyse des verbesserten Multiplikationsverfahrens . Dieses Gesetz markiert den Fall ohne *overhead* Kosten.

Allgemeine Lösung:

$$\text{Falls } g(n) = O(n^{u-\varepsilon}) \text{ für ein } \varepsilon > 0 \text{ dann ist } T(n) = \Theta(n^u)$$

$$\text{Falls } g(n) = \Theta(n^u) \text{ dann ist } T(n) = \Theta(n^u \log_2 n)$$

$$\text{Falls } g(n) = \Omega(n^{u+\varepsilon}) \text{ für ein } \varepsilon > 0 \text{ und } ag\left(\frac{n}{b}\right) \le cg(n) \text{ dann ist}$$

$$T(n) = \Theta(g(n))$$

Zusammenfassung

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