

Theoretische Informatik II

Einheit 6

Komplexitätstheorie



1. Konkrete Komplexitätsanalyse
2. Das \mathcal{P} - \mathcal{NP} Problem
3. NP-vollständige Probleme
4. Grenzen überwinden
5. Die Komplexitätsklassenhierarchie

KOMPLEXITÄTSTHEORIE

WAS KANN MIT VERTRETbareM AUFWAND GELÖST WERDEN?

- **Berechenbarkeitsanalyse alleine reicht nicht**

- Klärt nur die Grundsatzfrage: berechenbar/entscheidbar oder nicht
- Für praktische Lösbarkeit muß **Berechnungsaufwand vertretbar** sein

- **Analyse benötigter Ressourcen (Komplexität)**

- **Zeitbedarf** des Algorithmus
- **Speicherbedarf** des Verfahrens (RAM, Harddisk)
- Netzzugriffe, Zugriff auf andere Medien, ...

Time
Space

- **Die Meßgröße muß objektiv sein**

- Unabhängig von konkreter Hardware und Programmiersprache
Optimierungsfähigkeiten des Compilers
Auswahl der Testdaten

Komplexitätsmaße sollten abstrakt formuliert sein

GRÖSSENORDNUNG IST WICHTIGER ALS DETAIL

- **Genaue Betrachtungen sind unpraktikabel**

- Mühsam bei nichttrivialen Algorithmen
- Abhängig von Programmierdetails und Maschinenmodell

Welches Maschinenmodell sollte der Standard sein?

- **Abschätzung der Komplexität ist sinnvoller**

- Bestimme Aufwand relativ zur Eingabegröße

- $T_M(n) = \max\{t_M(w) \mid |w|=n\}$ (worst-case)

- $S_M(n) = \max\{s_M(w) \mid |w|=n\}$ Einheit 5.1, Folie 7

- Asymptotisches Verhalten für große Eingabedaten ist wichtig

- **Ignoriere Konstanten**

- Additive Konstanten und konstante Faktoren werden durch Hardwaresteigerungen ausgeglichen

Analyse des wirklich relevanten Aufwands

DIE MATHEMATIK ASYMPTOTISCHER VERGLEICHE

- $g \leq_a f$ (f wächst asymptotisch schneller als g)

- Es gibt ein $n_0 \in \mathbb{N}$ mit $g(n) \leq f(n)$ für alle $n \geq n_0$

Ab einer bestimmten Stelle ist f immer mindestens so groß wie g

- **(Größen-)Ordnung einer Funktion**

- f als obere Schranke: $\mathcal{O}(f) = \{g: \mathbb{N} \rightarrow \mathbb{R}^+ \mid \exists c > 0. g \leq_a c * f\}$

- f als echte obere Schranke: $\mathcal{o}(f) = \{g: \mathbb{N} \rightarrow \mathbb{R}^+ \mid \forall c > 0. g <_a c * f\}$

- f als untere Schranke: $\mathcal{\Omega}(f) = \{g: \mathbb{N} \rightarrow \mathbb{R}^+ \mid \exists c > 0. c * f \leq_a g\}$

- f als exakte Schranke: $\Theta(f) = \{g: \mathbb{N} \rightarrow \mathbb{R}^+ \mid \exists c, c' > 0. c * f \leq_a g \leq_a c' * f\}$

Schreibweisen auch: $\mathcal{O}(1) \hat{=} \mathcal{O}(\lambda n.1)$, $\mathcal{O}(n) \hat{=} \mathcal{O}(\lambda n.n)$, $\mathcal{O}(n^2) \hat{=} \mathcal{O}(\lambda n.n^2) \dots$

- **Beispiele für Ordnung konkreter Funktionen**

- Konstante Funktion: $g_1(n) = k$ für alle n

$$g_1 \in \mathcal{O}(1)$$

- Polynome: $g_2(n) = c_0 + c_1 * n + \dots + c_m * n^m$

$$g_2 \in \mathcal{O}(n^m)$$

- Fakultätsfunktion: $g_3(n) = n! = 1 * 2 * \dots * n$

$$g_3 \in \mathcal{O}(n^n)$$

- **Komplexität einer Maschine**

- M hat **Zeit-/Platzkomplexität $\mathcal{O}(f)$** , falls $T_M \in \mathcal{O}(f)$ bzw. $S_M \in \mathcal{O}(f)$

RECHENZEIT: WO LIEGT DIE GRENZE DES HANDHABBAREN?

Rechenzeiten auf 3.3 Ghz Prozessor									
Größe n	10	20	30	40	50	60	...	1000	1.000.000
Wachstum									
$\log_2 n$	1ns	2ns		3ns				10ns	100ns
n	3ns	6ns	9ns	12ns	15ns	18ns		300ns	300 μ s
n^2	30ns	120ns	270ns	480ns	750ns	1.1 μ s		300 μ s	300s
n^3	300ns	2.4 μ s	8.1 μ s	19.2 μ s	37.5 μ s	64 μ s		300ms	9.5y
2^n	300ns	300 μ s	300ms	300s	83.3h	9.5y			
3^n	17.8 μ s	1.1s	17.3h	116y	2.500.000.000y				

**Wieviel mehr kann man in der gleichen Zeit berechnen,
wenn Computer um den Faktor 1000 schneller werden?**

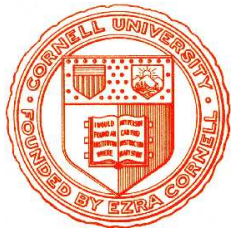
	$\log_2 n$	n	n^2	n^3	2^n	3^n
Problemsteigerung	10^{300} -fach	1000-fach	31-fach	10-fach	plus 10	plus 6

KONSEQUENZ FÜR KOMPLEXITÄTSANALYSEN

- **Polynomielle Lösbarkeit ist entscheidend**
 - Exponentieller Aufwand ist für die Praxis unakzeptabel
 - Unterschiede innerhalb polynomieller Komplexität sind tolerierbar aber durchaus relevant für konkrete Implementierungen
- **Bessere Hardware ist selten eine gute Lösung**
 - Wenn Algorithmen schlecht sind, nützt die beste Hardware wenig
 - Es lohnt sich, in die Verbesserung von Algorithmen zu investieren
- **Fragestellungen der Komplexitätstheorie**
 - Welche Probleme haben (in etwa) den gleichen Schwierigkeitsgrad?
 - Kann man effiziente Lösungen wiederverwenden? (Problemreduktion)
 - Zusammenhang zwischen Platzbedarf und Laufzeitverhalten?
 - Welche Problemstellungen sind (nicht) polynomiell lösbar?
 - Welche Verbesserung können unkonventionelle Ansätze erreichen?
(Nichtdeterministische/parallele, approximierende, probabilistische Verfahren)

Theoretische Informatik II

Einheit 6.1



Komplexität von Algorithmen und Problemen



1. Zeitkomplexität spezifischer Algorithmen
2. Zeitkomplexität von Problemstellungen

Obere Schranken für die Laufzeit von Verfahren

- **Analyse auf Ebene abstrakter Algorithmen**
 - Asymptotische Komplexität hängt nicht von Programmiersprache ab
 - **Konstanter Expansionsfaktor** bei Übersetzung in Maschinsprache (Simulation durch Turingmaschinen würde zu polynomieller Expansion führen)
- **Elementaroperationen gelten als ein Schritt**
 - **+, -, *, /, ... Einzelschritte**, wenn Zahlengröße beschränkt (z.B. 64-bit)
 - Liefert einfache modellunabhängige Zählung von Elementaroperationen
 - **Höherer Aufwand bei beliebig großen Zahlen**
- **Fokus auf sequentielle Algorithmen**
 - **Parallele/nichtdeterministische Maschinen** haben evtl. bessere Laufzeit

SEQUENTIELLE SUCHE: KOMMT x IN L VOR?

• Durchsuche Liste L von links nach rechts

```
function searchseq(x,L) ≡  
    found := false;  
    for i = 1 to length(L) do  
        if L[i]=x then found:=true  
    od;  
    return found;
```

Verfahren ist anwendbar auf beliebige Listen

• Laufzeitanalyse

- Eine Operation für Initialisierung `found:=false`
- Je 2 Operationen pro Element von L in der `for`-Schleife
- Eine Operation für Ausgabe des Ergebnisses
- Insgesamt $2n+2$ Schritte, wenn n die Größe der Liste L ist

Sequentielle Suche ist in $\mathcal{O}(n)$

BINÄRE SUCHE

Nur anwendbar, wenn Liste L geordnet ist

- **Teste mittleres Element; dann rechts oder links**

```
function searchbin(x,L) ≡  
  let function searchb(x,L,left,right) ≡  
    if left>right then return false  
    else  
      mid := (left+right) div 2;  
      if x<L[mid] then searchb(x,L,left,mid-1)  
      elseif x>L[mid] then searchb(x,L,mid+1,right)  
      else return true  
    fi;  
  return searchb(x,L,1,length(L))
```

- **Eine grobe Laufzeitanalyse reicht aus**

- Konstante Anzahl von Operationen pro Aufruf von `searchb`
- **Wie oft wird `searchb` aufgerufen?**

BINÄRE SUCHE – ANALYSE

```
function searchbin(x,L) ≡  
  let function searchb(x,L,left,right) ≡  
    if left>right then return false  
    else  
      mid := (left+right) div 2;  
      if x<L[mid] then searchb(x,L,left,mid-1)  
        elseif x>L[mid] then searchb(x,L,mid+1,right)  
        else return true  
      fi;  
  return searchb(x,L,1,length(L))
```

Abstand von left und right halbiert sich pro Aufruf (mit Abrundung)

Anzahl von Operationen pro Aufruf von search_b ist eine Konstante k

Abstand zu Beginn ist $n-1$ (n ist die Größe der Liste L)

search_b terminiert bei Erfolg oder wenn Abstand Null ist

Lösung der Gleichung $time(n) = k + time(\lfloor n/2 \rfloor)$ ist $time(n) = k * \log_2 n$



Binäre Suche ist in $\mathcal{O}(\log_2 n)$

SORTIERVERFAHREN

- **Ordne Elemente in aufsteigender Reihenfolge**

- Geordnete Listen unterstützen **effizienten Zugriff** auf Elemente
- Eine der häufigsten Operationen in der Programmierung

- **Viele Verfahren bekannt**

<http://www.sortieralgorithmen.de>

- **Insertion Sort**: Einfügen des Listenanfangs in geordnete Teilliste
- **Selection Sort**: Auswahl des jeweils kleinsten Elements als Listenanfang
- **Bubblesort**: Austauschen benachbarter Elemente
- **Quicksort**: Aufteilung nach Größe, Sortieren der entstehenden Teillisten
- **Mergesort**: Aufteilen in Teillisten, Sortieren und Mischen der Teillisten
- **Mergesort (II)**: Identifizieren und Mischen geordneter Teillisten

‘Bestes’ Verfahren hängt von Problemgröße ab

BUBBLESORT

Fortlaufender Vergleich benachbarter Elemente Austausch bei falscher Reihenfolge

```
function bubblesort(L) ≡  
  for upper = length(L)-1 downto 1 do  
    for j = 1 to upper do  
      if L[j]>L[j+1] then  
        aux := L[j];  
        L[j] := L[j+1];  
        L[j+1] := aux  
      fi  
    od  
  od
```

• Beispiel einer Sortierung mit Bubblesort

1	2	5	6	7	8	9
---	---	---	---	---	---	---



Elemente steigen wie Blasen auf, bis sie auf größere treffen

BUBBLESORT - LAUFZEITANALYSE

```
function bubblesort(L) ≡
  for upper = length(L)-1 downto 1 do
    for j = 1 to upper do
      if L[j]>L[j+1] then
        aux := L[j];
        L[j] := L[j+1];
        L[j+1] := aux
      fi
    od
  od
```

- **Feste Anzahl von Operationen im Schleifenrumpf**
 - Vergleich benachbarter Elemente
 - ggf. Austausch unter Verwendung einer Hilfsvariablen
- **Anzahl Schleifen abhängig von Listengröße n**
 - Innere Schleife wird jeweils genau $upper$ -mal durchlaufen
 - Insgesamt $n-1 + n-2 + \dots + 2 + 1 = n*(n-1)/2$ Durchläufe



Bubblesort ist in $\mathcal{O}(n^2)$

SORTIEREN SCHNELLER ALS $O(n^2)$

- **Identifiziere Läufe**, d.h. geordnete Teilfolgen

9	7	8	2	1	5	6
---	---	---	---	---	---	---

- **Verschmelze Läufe zu neuen Läufen**

9	7	8	2	1	5	6
7	8	9	1	2	5	6

– Länge der Läufe wächst – Anzahl halbiert sich

- **Wiederhole bis Folge geordnet**

7	8	9	1	2	5	6
1	2	5	6	7	8	9

– Liste ist eine einzige (komplett) geordnete Teilfolge ✓

Abstrakte Skizze reicht für Laufzeitanalyse

- **Verschmelzen ist in $\mathcal{O}(n)$**
 - Folge wird jeweils komplett durchlaufen
- **Verschmelzen halbiert Anzahl der Läufe**
 - Je zwei Läufe werden zu einem gemischt
- **Man braucht maximal $\log_2 n$ Verschmelzungen**
 - Danach ist nur ein einziger Lauf übrig, d.h. die Liste ist sortiert



Sortieren durch Verschmelzen ist in $\mathcal{O}(n * \log_2 n)$

- **Fundamentale Datenstruktur vieler Anwendungen**

- Ordnungstruktur für effiziente Verwaltung großer Datenmengen
- Beschreibung der Topographie von Netzwerken

⋮

- **Graphen haben Knoten und Kanten** ($G = (V, E)$)

- Eine Kante e zwischen zwei Knoten $v \neq v'$ kann **gerichtet** ($e = (v, v')$) oder **ungerichtet** ($e = \{v, v'\}$) sein
- Beschreibbar als Liste $v_1, \dots, v_n, \{v_{i_1}, v'_{i_1}\}, \dots, \{v_{i_m}, v'_{i_m}\}$
- In **gewichteten** Graphen ist jede Kante mit einer Zahl markiert

- **Bäume sind zyklensfreie ungerichtete Graphen**

- Ein Baum **spannt einen Graphen auf**, wenn jeder Knoten von G von der Wurzel des Baums aus erreichbar ist
- Ein **MWST** ist ein aufspannender Baum mit minimalem Gewicht

DER KRUSKAL ALGORITHMUS

Bestimme einen MWST in einem Graphen G

- **Erzeuge Zusammenhangskomponenten in G**

- Initialwert ist $\{v\}$ für jeden Knoten $v \in V$ ($Z := \{\{v\} \mid v \in V\}$)
- Betrachte eine neue Kante $e \in E$ mit geringstem Gewicht
Falls e Knoten aus verschiedenen Zusammenhangskomponenten verbindet, füge e dem MWST hinzu und vereinige die beiden Komponenten
- Wiederhole dies, bis alle Knoten in einer Komponente sind oder alle Kanten betrachtet wurden

- **Implementierbar mit Laufzeit $\mathcal{O}(|V| + |E| \log |E|)$**

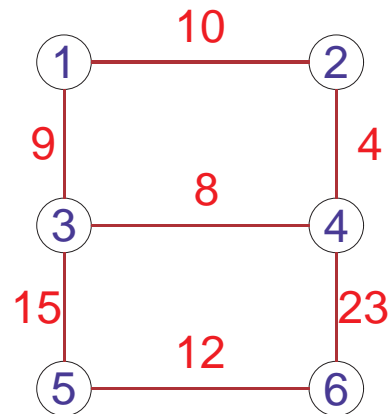
- Liste der Kanten muß zuerst nach Gewicht sortiert werden
- Zusammenhangskomponenten müssen mit Pointern repräsentiert werden
- Turingmaschine würde Laufzeit $\mathcal{O}((|V|+|E|)^4)$ benötigen HMU §10.1.2

DAS PROBLEM DES HANDLUNGSREISENDEN

Gegeben n Städte, eine Kostentabelle von Kosten c_{ij} um von Stadt i nach Stadt j zu reisen und eine Kostenbeschränkung B . Gibt es eine Rundreise durch alle n Städte, deren Kosten unter dem Limit B liegt?

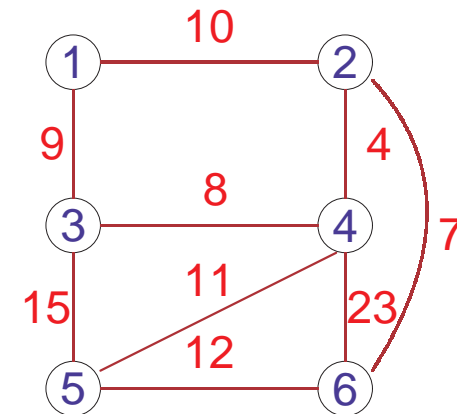
● Formulierung als Graphenproblem

- Ein **Hamiltonscher Kreis** im Graphen $G = (V, E)$ ist ein Kreis, der nur aus Kanten aus E besteht und jeden Knoten genau einmal berührt.
- **TSP**: Finde einen Hamiltonschen Kreis mit minimalen Kosten B



Nur eine Rundreise: [1,3,5,6,4,2]

Kosten: 73

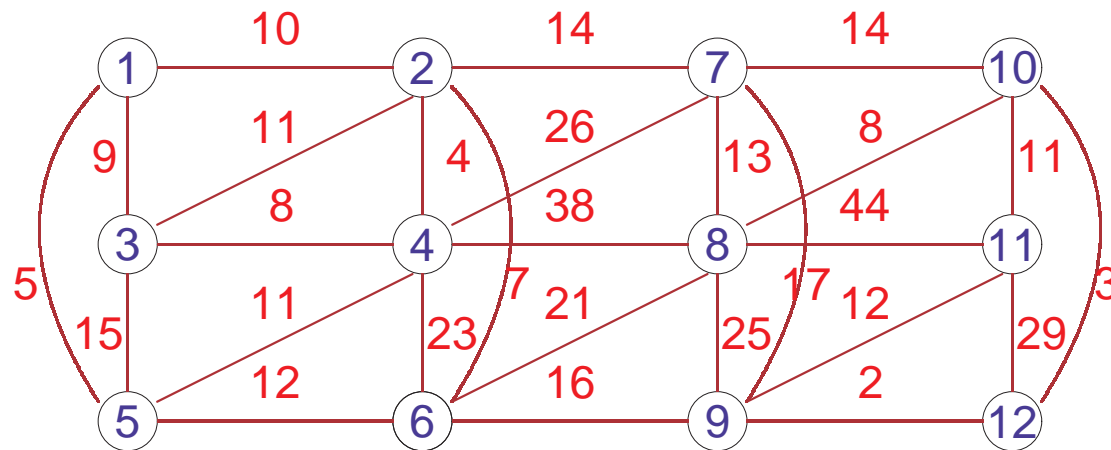


Billigere Rundreise: [1,2,6,5,4,3]

Kosten: 57

DAS PROBLEM DES HANDLUNGSREISENDEN

- **Graphen können sehr komplex sein**



- **Keine effiziente allgemeine Lösung bekannt**
 - Bester Ansatz ist “Generate & Test”
 - Test ist polynomiell, aber es gibt **exponentiell viele Möglichkeiten**
- **Approximative Lösungen möglich**
 - Rundreise mit Kosten 50% über Optimum polynomiell bestimmbar
 - Benötigt Rahmenbedingung $c_{ij} \leq c_{ik} + c_{kj}$ (Dreiecksungleichung)

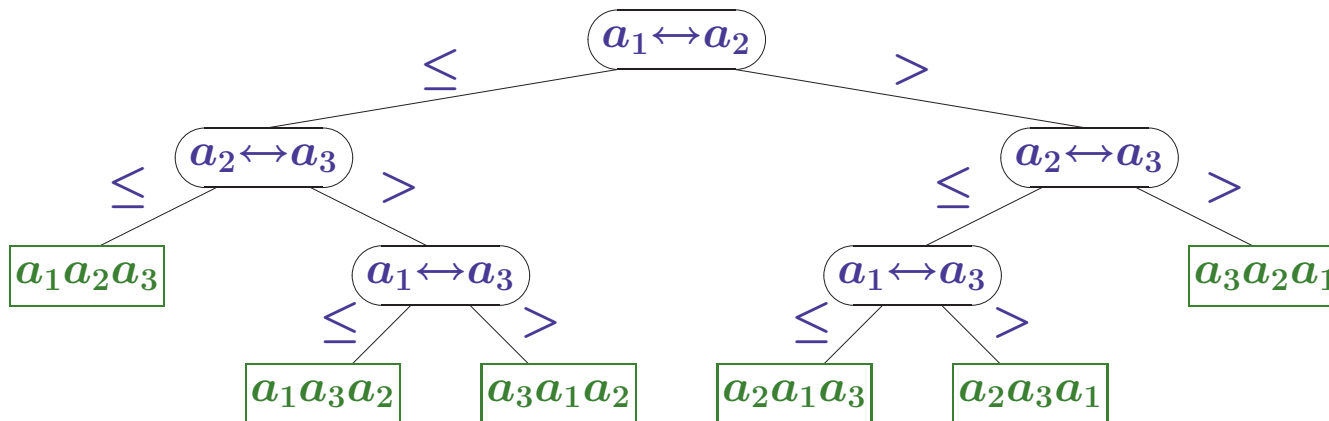
Untere Schranken für Komplexität von Lösungen

- **Lösungen eines Problems sind unterschiedlich gut**
 - Suchen: Lineare Suche $\mathcal{O}(n)$ — Binärsuche $\mathcal{O}(\log_2 n)$
 - Sortieren: Bubblesort $\mathcal{O}(n^2)$ — Mergesort $\mathcal{O}(n \cdot \log_2 n)$
- **Wie effizient kann ein Problem gelöst werden?**
 - Gibt es eine Mindestkomplexität für eine optimale Lösung?
 - Wann ist eine Lösung gut genug?
- **Antwort könnte von Art der Frage abhängen**
 - **Entscheidungsproblem:** *Gibt es überhaupt eine Lösung der Aufgabe?*
 - **Optimierungsproblem:** *Was ist die bestmögliche Lösung?*
 - **Berechnungsproblem:** *Bestimme eine konkrete Lösung*
- **Nachweis ist im Normalfall aufwendig**
 - Man muß über alle möglichen Algorithmen argumentieren

KOMPLEXITÄT VON SORTIERVERFAHREN

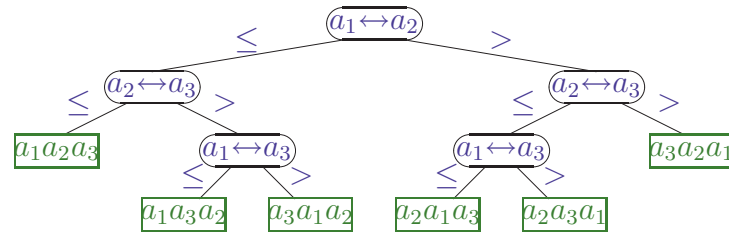
Geht es schneller als $\mathcal{O}(n * \log_2 n)$?

- **Sortierverfahren müssen Elemente vergleichen**
 - Sonst kann die Anordnung der Elemente nicht garantiert werden
 - **Wieviele Vergleiche** werden benötigt um a_1, \dots, a_n zu ordnen?
 - Bestimme Anzahl der Vergleiche für den **ungünstigsten Fall**
- **Betrachte Entscheidungsbaum** von Algorithmen



- Innere Knoten entsprechen den durchgeführten **Vergleichen**
- Kanten markiert mit **Vergleichsergebnis** ($\leq, >$)
- Blätter sind resultierende **Anordnung der Elemente**

KOMPLEXITÄT VON SORTIERVERFAHREN (II)



- **Algorithmen entsprechen Entscheidungsbäumen**

- Abarbeitung für konkrete Eingaben entspricht einem Ast im Baum
- Konkrete Laufzeit des Algorithmus entspricht Länge des Astes
- Komplexität des Algorithmus entspricht Tiefe des Entscheidungsbaumes
Komplexität von Sortieren \equiv minimale Tiefe von Entscheidungsbäumen

- **Wie tief ist ein Entscheidungsbaum?**

- Jeder Entscheidungsbaum für hat a_1, \dots, a_n hat $n!$ Blätter
- Ein binärer Baum der Tiefe k hat maximal 2^k Blätter
- Jeder Entscheidungsbaum hat mindestens Tiefe $\log_2 n!$
- $\log_2 n! = \log_2(\prod_{i=1}^n i) = \sum_{i=1}^n \log_2 i \geq \sum_{i=n/2}^n \log_2(n/2) = n/2 * (\log_2 n - 1)$

Sortieren ist in $\Omega(n * \log_2 n)$

KOMPLEXITÄT ANDERER PROBLEMSTELLUNGEN

● **Arithmetik auf großen (k -stelligen) Zahlen**

- **Addition:** Einstellig von rechts nach links mit Übertrag $\mathcal{O}(k)$
- **Multiplikation:** Jede Stelle muß mit jeder multipliziert werden $\mathcal{O}(k^2)$
- **Division:** Schriftliche Division von links nach rechts $\mathcal{O}(k^2)$

● **Matrixmultiplikation $n \times n$ -Matrizen** $\mathcal{O}(n^3)$

● **Berechnung von $n!$** $\mathcal{O}(n^2 * (\log_2 n)^2)$

- **Obergrenze:** n -fache Multiplikation von n und $n!$: $n * \log_2 n * \log_2(n^n)$
- **Untergrenze:** $n/2$ -fach $n/2 * (n/2)!$: $n/2 * \log_2(n/2) * n/4 * (\log_2 n - 2)$

● **Primzahltest bei k -stelligen Zahlen** $\mathcal{O}(k^{12})$

- AKS Algorithmus auf Basis tiefer mathematischer Einsichten (2002)
- Alle früheren Verfahren waren exponentiell
- Alle bekannten Faktorisierungsverfahren sind exponentiell
- Ergebnis gut für offene kryptographische Systeme (wähle $k > 500$)