Algorithmen und Programmierung

8. Kapitel
Komplexität

Prof. Matthias Werner
Professur Betriebssysteme

8.1 Motivation

- Das Kino-Problem (Beispiel 7.5) hat gezeigt, dass es unterschiedlich effiziente Algorithmen gibt
- Das Traveling-Salesman-Problem (Beispiel 7.6) hat gezeigt, dass Probleme existieren, die zwar theoretisch algorithmisch, aber **nicht** praktisch durchführbar gelöst werden können
- **Effizienz** ➡ Ressourcenverbrauch
  - Rechenzeit
  - Speicherplatz

Messung der Rechenzeit

- **Ansatz**: Direktes Messen der Rechenzeit eines Programmes
  - Stoppuhr
  - Betriebssystemfunktionen (➡ `time`)
- **Problem**: Hier gibt es (zu) viele Einflüsse, die gemessen werden
  - Compiler
  - Rechnerkonfiguration
  - Rechnerlast
  - Betriebssystem

Merke:

Zur Bestimmung der Qualität einer konkreten Implementation auf einem konkreten Rechner geeignet, aber nicht für allgemeine Aussagen über die Qualität von Algorithmen

Messung der Rechenzeit (Forts.)

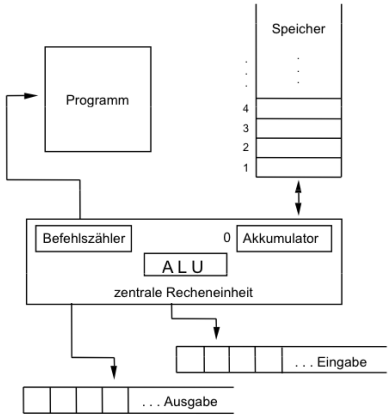
- Rechner in Realität unterschiedlich schnell, **abhängig vom Instruktionsmix** eines Programmes
- **Beispiel**:
  - CPU1 führt alle Befehle in 2 Takten aus
  - CPU2 führt alle Integerbefehle in einem Takt aus, alle anderen in drei Takten
  - Programm A bestehe aus 60% Integerbefehlen und 30% sonstigen Befehlen
  - Programm B bestehe aus 90% Integerbefehlen und 10% sonstigen Befehlen
  - Programm A ist schneller auf CPU1
  - Programm B ist schneller auf CPU2

Messung der Rechenzeit (Forts.)

- Um die Qualität von Algorithmen (nicht Implementationen) zu beurteilen ist eine genaue Zeit gar nicht nötig
- Dazu nutzen wir **abstrakte Computermodelle**

8.2 Das RAM-Modell

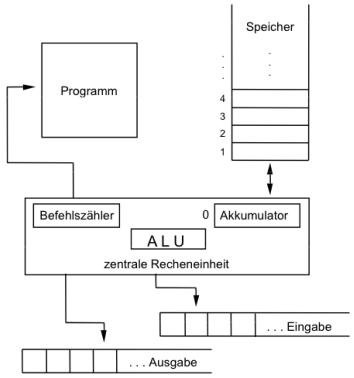
- **Random Access Machine**: etwa Maschine mit freiem Zugriff
- Ähnlich von-Neumann-Modell, aber andere Motivation ➡ Abschätzung von Laufzeiten



Das RAM-Modell (Forts.)

Bestandteile der RAM

- **Programm**:
  - Nummerierte, endliche Folge von Befehlen
- **Speicher**:
  - abzählbar (unendlich) viele Speicherstellen (Register)
  - wahlfrei adressierbar
  - jedes Register kann beliebige (ganze) Zahl speichern
- **Ein-/Ausgabe**:
  - Kontinuierliche Sequenzen (Bänder)
  - Jeweils nur Eingebe (Lesen) oder nur Ausgabe (Schreiben)
- **Zentrale Recheneinheit**:
  - Befehlszähler enthält Nummer des auszuführenden Befehls
  - Akkumulator: Zielregister von Berechnungen, Adresse 0
  - Arithmetic Logic Unit: Funktionseinheiten für die Ausführung von Operationen



Befehle und Kosten

- In einer RAM gibt es die „üblichen“ Befehle
  - Grundrechenarten: $+$, $-$, $\cdot$, $/$, $\mod$
  - Vergleiche: $>$, $<$, $=$, $\geq$, $\leq$
  - Verzweigungen (`IF`)
  - Sprünge (`GOTO`) ➡ Schleifen sind Verzweigungen + Sprünge
  - Laden/Speichern (`LOAD`, `STORE`)
  - Ein-/Ausgabe (`READ`,`WRITE`)
- **Operanden**:
  - Register (wahlfrei), auch indirekt
  - Implizit Akkumulator
  - Eingabesequenz/Ausgabesequenz (nicht wahlfrei)

Zeitkosten

- Für die RAM gibt es zwei Zeitkostenmodelle:
  - **Uniforme Kostenmaß**: Jeder Befehl hat die Zeitkosten 1 Zeiteinheit[1]
  - **Logarithmische Zeitkosten:** Die Länge der bearbeiteten Zahlen bestimmen die Zeit
    - Länge $l(x)$ von $x \in \mathbf{G}$: $l(0) = 1, l(x) = \lfloor \log_2 |x| \rfloor + 1$
    - Die logarithmischen Zeitkosten eines Befehls sind gleich der Summe der Längen der bearbeiteten Zahlen

Anmerkung:

Das logarithmische Kostenmaß sollte immer dann verwendet werden, wenn die Größe der im Algorithmus vorkommenden Zahlen von entscheidender Bedeutung ist.

Beispiel: Primfaktorenzerlegung

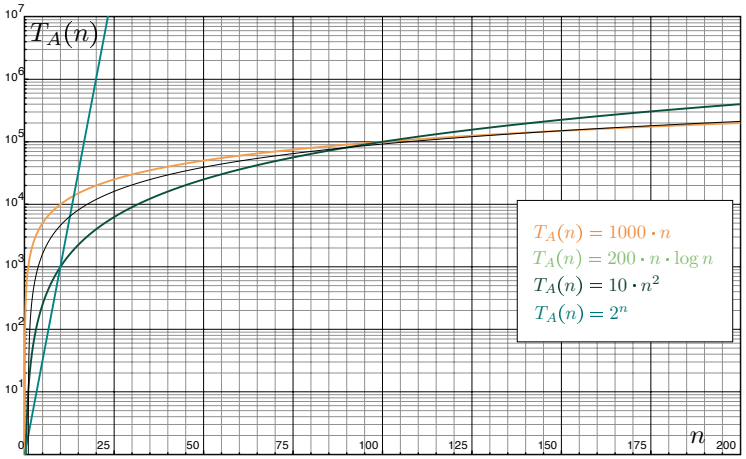
[1] Z.B. Takt oder Millisekunde

Zeitkosten (Forts.)

- Da jeder Befehl (im uniformen Kostenmaß) die gleiche Länge hat, kommt es auf die Anzahl T_A von Befehls**ausführungen** an
- Im logarithmischen Modell muss zusätzlich die Größe der Operanden berücksichtigt werden
- Jeder Befehl, der in einer Schleife ausgeführt wird, zählt mehrmals
- Anzahl von Befehlsausführungen (und Schleifendurchläufen) häufig von der Eingabegröße n abhängig ➡ $T_A(n)$
- Beispiel: Betrachte folgende Algorithmen zur Lösung eines Problems

Algorithmus	$T_A(n)$	**optimal für**
A_1	$1000 \cdot n$	$n \geq 101$
A_2	$200 \cdot n \cdot \log n$	nie
A_3	$10 \cdot n^2$	$10 \leq n \leq 100$
A_4	2^n	$1 \leq n \leq 9$

Zeitkosten (Forts.)



- Man beachte: $T_{A_4}(10) \approx T_{A_3}(10)$, aber $T_{A_4}(25) \approx 5400 \cdot T_{A_3}(25)$

Laufzeitanalyse

- Auch bei konstanten n ist T_A nicht immer gleich
- Betrachten:
- **Worst-Case**-Analyse
  - Für jedes n definiere Laufzeit $T(n) = \max(t(Input)), \forall |Input| = n$
  - Garantierte Schranke für jede Eingabe
  - Standard
- **Average-Case**-Analyse
  - Für jedes n definierte Laufzeit $T(n) = \bar{t}(Input), \forall |Input| = n$
  - Hängt von der Definition des Durchschnitts ab ➡ Verteilung der Eingaben
  - Seltener benutzt, trotz z.T. höherer praktischer Relevanz
- **Best-Case**-Analyse
  - Für jedes n definierte Laufzeit $T(n) = \min(t(Input)), \forall |Input| = n$
  - Minimale Laufzeit
  - Aufzeigen von Entwurfsfehlern

8.3 Die $\mathcal{O}$-Notation

Merke

Da es relativ einfach ist, bei besten Fall zu mogeln, hat die Best-Case-Analyse praktisch keine Bedeutung.

Da es häufig sehr schwer ist, den Durchschnitt zu berechnen, wird es meist unterlassen.

- Allgemein ist die genaue Analyse von T_A sehr schwierig
- Deshalb: Konzept von **Größenordnungen**
- Bekannt aus dem täglichen Leben, z.B. für Geschwindigkeiten:

$$v_{Laufen} < v_{Rad} < v_{Auto} < v_{Flugzeug}$$

Die $\mathcal{A}$-Notation

- Es ist interessant eine **maximale Schranke** anzugeben
- Definieren Menge $\mathcal{A}(x)$ von Funktionen/Werten, die **höchstens** den Wert x annehmen
- Beispiele:
  - $\pi = 3{,}14 + c, c \in \mathcal{A}(0{,}005)$ bedeutet, dass $\pi \approx 3{,}14$ ist
  - $2 \in \mathcal{A}(5)$, aber auch $4 \in \mathcal{A}(5)$
  - $sin(x) \in \mathcal{A}(1)$
- Man schreibt häufig (etwas ungenau):
  - $\pi = 3{,}14 + \mathcal{A}(0{,}005)$
  - $2 = \mathcal{A}(5)$ bzw. $4 = \mathcal{A}(5)$
  - $sin(x) = \mathcal{A}(1)$
- Das Gleichheitszeichen „=“ ist hier im Sinne von „ist“ zu verstehen und muss von links nach rechts gelesen werden
  - „Aristoteles ist ein Mensch“, aber ein Mensch ist nicht (notwendigerweise) Aristoteles

Die $\mathcal{A}$-Notation (Forts.)

- Die $\mathcal{A}$-Notation kann Konstanten wie Variablen oder Funktionen als Argument haben:

$$\begin{aligned} sin(x) &= \mathcal{A}(1) \\ x &= \mathcal{A}(1) \\ \mathcal{A}(x) &= x \cdot \mathcal{A}(1) \\ \mathcal{A}(x) + \mathcal{A}(y) &= \mathcal{A}(x+y) && \text{wenn } x \geq 0 \wedge y \geq 0 \\ (1+\mathcal{A}(t))^2 &= 1 + 3 \cdot \mathcal{A}(t) && \text{wenn } t = \mathcal{A}(1) \end{aligned}$$

Die $\mathcal{O}$-Notation

- Häufig ist allerdings nicht interessant, wie groß etwas wird, sondern **wie schnell** es groß wird ➡ **Wachstum**

Definition 8.1 (Wachstum)

*Zwei Funktionen $f(n)$ und $g(n)$ haben das gleiche Wachstumsverhalten, falls für genügend große n das Verhältnis der beiden nach oben und unten durch **Konstanten** beschränkt ist, d.h.*

$$\exists c, d, n_0 \in \mathbf{N}, \forall n \geq n_0, \quad c < \frac{f(n)}{g(n)} < d, \quad c < \frac{g(n)}{f(n)} < d$$

- Warum die Forderung nur für „genügend große“ n?
- ➡ Interessant sind Verfahren, die für große Probleminstanzen noch effizient sind
- Sprachregelung: „Sie skalieren gut.“

Beispiele

- $f_1(n) = n^2$ und $f_2(n) = 5 \cdot n^2 - 7 \cdot n$ haben das **gleiche** Wachstum:
  - Für alle $n > 2$ gilt: $\frac{1}{5} < \frac{(5n^2-7n)}{n^2} < 5$ und $\frac{1}{5} < \frac{n^2}{(5^2-7n)} < 5$
- $f_1(n) = n^3$ und $f_2(n) = n^2$ haben **nicht** das gleiche Wachstum
  - Für alle hinreichend große n gilt: $\frac{n^3}{n^2} = n > c$

Mengen

- Ähnlich, wie die $\mathcal{A}$-Notation eine Grenze angibt, definieren wir ein $\mathcal{O}$-Notation zur Beschreibung des Wachstums
- Informal: $\mathcal{O}(x)$ entspricht $c \cdot \mathcal{A}(x)$ mit einer unbekannten Konstante c
- Formal:

Definition 8.2 ($\mathcal{O}$-Notation)

$$\mathcal{O}(f(n)) = \{g(n) | \exists c > 0, \exists n_0 > 0, \ \ \forall n > n_0, \ g(n) \leq c \cdot f(n)\}$$

- Ähnlich wie bei $\mathcal{A}$-Notation schreibt man
  - $5n^2 - 7n = \mathcal{O}(n^2)$ statt $5n^2 - 7n \in \mathcal{O}(n^2)$
- Die Schreibweise mit dem „großen O“ (ursprünglich war der griechische Buchstabe Omikron gemeint) wird als **Landau-Notation** bezeichnet

Dominierung

- Die $\mathcal{O}$-Notation gibt an, welche Funktion das Wachstum **dominiert**

Definition 8.3 (Dominierung)

Bei zwei monotonen[2] Funktionen $f(n)$ und $g(n)$ dominiert $f(n)$ die Funktion $g(n)$, wenn

$$g(n) \in \mathcal{O}(f(n))$$

gilt.

- Schreibweise: $\mathrm{dom}(f(n), g(n))$ gibt die dominierende Funktion
- **Beispiele**:
  - $\mathrm{dom}(n^3, n^2) = n^3$
  - $\mathrm{dom}(2^n, n^k) = 2^n$ (für konstante $k > 1$)

[2] Der allgemeine Fall ist etwas komplizierter

Rechenregeln für $\mathcal{O}$-Notation

- Für Funktionen $f(n)$ (bzw. $g(n)$) mit der Eigenschaft $\exists n_0 > 0, \ \forall n > n_0, \ f(n) > 0$ gilt:

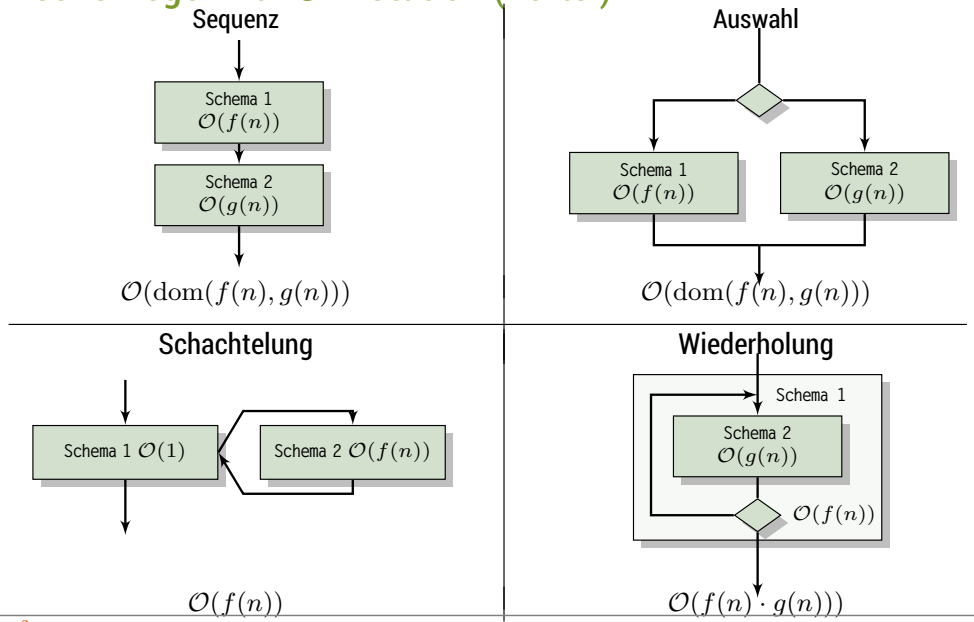
$$\mathcal{O}(f(n)) + \mathcal{O}(g(n)) = \mathcal{O}(f(n) + g(n))$$
$$\mathcal{O}(f(n)) \cdot \mathcal{O}(g(n)) = \mathcal{O}(f(n) \cdot g(n))$$
$$\mathcal{O}(f(n) + g(n)) = \mathcal{O}(\mathrm{dom}(f(n), g(n)))$$

- Beispiel: $\mathcal{O}(\sum_{i=1}^{n} i) = \mathcal{O}(\frac{n^2+n}{2}) = \mathcal{O}(n^2)$

Achtung!

Bei Anwendung in Induktionen ist aufgrund des „unüblichen“ Gleichheitszeichen Vorsicht geboten!

Rechenregeln für $\mathcal{O}$-Notation (Forts.)



Wichtige Klassen von Funktionen

	Sprechweise	Typische Algorithmen/Operationen
$\mathcal{O}(1)$	konstant	Addition, Vergleichsoperationen, rekursiver Aufruf, ...
$\mathcal{O}(\log n)$	logarithmisch	Suchen in einer sortierten Sequenz
$\mathcal{O}(n)$	linear	Bearbeiten jedes Elementes einer Menge
$\mathcal{O}(n \cdot \log n)$		gute Sortierverfahren
$\mathcal{O}(n \cdot \log^2 n)$		
	$\vdots$	
$\mathcal{O}(n^2)$	quadratisch	primitive Sortierverfahren
$\mathcal{O}(n^k), k \geq 2$	polynomiell	
	$\vdots$	
$\mathcal{O}(2^n)$		Ausprobieren von Kombinationen
$\mathcal{O}(k^n), k > 1$	exponentiell	

Eingabecodierung

- Die $\mathcal{O}(n)$-Notation hängt von der Größe der Eingabe ab
- Was ist das?
- ➡ Speicherplatz in Bits oder Worten
- Vorsicht bei der Codierung!

Primfaktorenzerlegung

- **Gegeben**: $x \in \mathbf{N}$
- **Gesucht**: $\{p_i\}, \prod_i p_i = x$

Als schwieriges Problem bekannt ➡ Grundlage von RSA

Eingabecodierung (Forts.)

- Trivialer Algorithmus für Primfaktorenzerlegung:

Require: $n \in \mathbf{N}$
Ensure: $\mathbf{P} = \{p_i\}, x = \prod_i p_i$

```
1: P ← ∅                                   ▷ P is multiset
2: for y ∈ [2, ⌊√x⌋] do
3:     while x mod y = 0 do
4:         P ← P ∪ {y}
5:         x ← x/y
6:     end while
7: end for
```

- **Binäre Kodierung** von x: Laufzeit exponentiell (bezüglich der Länge der Eingabe)
- **Unäre Kodierung** von x (x Einsen als Eingabe) Laufzeit linear

Merke:

Wird bei der $\mathcal{O}$-Notation die Größe von Zahlen als Eingabegröße betrachtet, wird stets die „dichte“ binäre Kodierung angenommen.

Weitere asymptotische Maße

Neben der $\mathcal{O}$-Notation, die angibt, wie eine Funktion „höchstens“ wächst, gibt es noch weitere Maße:

- $\Omega(f(n)) = \{g(n) | \exists c > 0, \exists n_0 > 0, \forall n > n_0, g(n) \geq c \cdot f(n)\}$
  ➡ „mindestens“
- $\Theta(f(n)) = \mathcal{O}(f(n)) \cap \Omega(f(n))$
  ➡ „genau“
- $o(f(n)) = \{g(n) | \forall c > 0, \exists n_0 > 0, \forall n > n_0, g(n) \leq c \cdot f(n)\}$
  ➡ „weniger“
- $\omega(f(n)) = \{g(n) | \forall c > 0, \exists n_0 > 0, \forall n > n_0, g(n) \geq c \cdot f(n)\}$
  ➡ „mehr“
- **Beachte:** Es gibt kein $\theta(f(n))$, da $o(f(n)) \cap \omega(f(n)) = \emptyset$ gilt

Rechenzeiten

- Wie lange ist die minimale Rechenzeit, wenn ein einzelner Rechenschritt $1 \mu sec$ dauert

Komplexität	$n = 10^1$	$n = 10^2$	$n = 10^3$	$n = 10^4$
$\Theta(\log_2 n)$	$4 \cdot 10^{-6} sec$	$7 \cdot 10^{-6} sec$	$10^{-5} sec$	$1{,}4 \cdot 10^{-5} sec$
$\Theta(n)$	$10^{-5} sec$	$10^{-4} sec$	$10^{-3} sec$	$10^{-2} sec$
$\Theta(n \log_2 n)$	$4 \cdot 10^{-5} sec$	$7 \cdot 10^{-4} sec$	$10^{-2} sec$	$0{,}14\ sec$
$\Theta(n^2)$	$10^{-4} sec$	$10^{-2} sec$	$1\ sec$	$1{,}8\ min$
$\Theta(2^n)$	$10^{-3} sec$	$0{,}4 \cdot 10^{17}$ Jahre	$> 10^{80}$ Jahre	$> 10^{80}$ Jahre
$\Theta(n!)$	$3\ sec$	$> 10^{80}$ Jahre	$> 10^{80}$ Jahre	$> 10^{80}$ Jahre

Beispiel: Minimumsuche

Algorithm SEEKMINIMUM
Require: Sequenz $a_1, \ldots, a_n$
Ensure: $p = \min(a_1, \ldots, a_n)$

```
1: p ← a_1                                   ▷ O(1)
2: for i ∈ {2, ..., n} do                    ▷ O(Σ_{i=2}^{n} (1 + T(L)))
3:     if a_i < p then                       ▷ O(1)
4:         p ← a_i                           ▷ O(1)
5:     end if
6: end for
```

- Folglich gilt:

$$\mathcal{O}\left(1 + \left(\sum_{i=2}^{n}(1+1)\right)\right) = \mathcal{O}(n)$$

- Da Schleifendurchlauf fest, auch:

$$\Omega\left(1 + \left(\sum_{i=2}^{n}(1+1)\right)\right) = \Omega(n)$$

- ... und damit SEEKMINIMUM $\in \Theta(n)$

Beispiel: Bubblesort

Algorithm BSORT
Require: $e_1, \ldots, e_n$
Ensure: $\forall i \in \{1, n-1\}, e_i \leq e_{i+1}$

```
1: repeat
2:     changed ← false;
3:     for i ← 1, ..., n − 1 do
4:         if e_i > e_{i+1} then
5:             SWAP(e_i, e_{i+1})
6:             changed ← true
7:         end if
8:     end for
9: until changed = false
```

- Wieviel Schleifendurchläufe bei `repeat-until`-Schleife?
- Immer Worst-Case-Fall betrachten:
  - Nach einem `for`-Schleifen-Durchlauf ist **mindestens** ein Element mehr geordnet
  - Worst-Case: Es ist **nur** ein Element mehr geordnet
  - Folgerung: $\mathcal{O}(n)$
- **Gesamt**:
  $\mathcal{O}\left(\mathcal{O}(n) \cdot \sum_{i=1}^{n-1}(\mathcal{O}(1))\right) = \mathcal{O}(n^2)$

Rekursion

- Betrachten rekursiven Algorithmus: Fakultät

Require: $n \in \mathbf{N}$
Ensure: $n!$
1: **procedure** FAK(n)
2: **if** $n = 1$ **then** ▷ $\mathcal{O}(1)$
3: **return** 1 ▷ $\mathcal{O}(1)$
4: **else**
5: **return** $n \cdot \text{FAK}(n-1)$ ▷ $\mathcal{O}(1 + \ldots?)$
6: **end if**
7: **end procedure**

- $T(n)$: Laufzeit von Fakultät
- $T(1) = \mathcal{O}(1)$
- $T(n) = T(n-1) + \mathcal{O}(1)$
  ➡ $T(n) = \mathcal{O}(n)$
- Mitunter ist die Komplexitätsbestimmung bei Rekursion wesentlich schwieriger ➡ Lösung von **Rekursionsgleichungen** notwendig

Durchschnittliche Laufzeit

- Laufzeiten müssen gewichtet werden
- Einfachster Ansatz: **Gleichverteilung** (uniforme Verteilung, alle Probleminstanzen $I \in \mathbf{I}_n$ gleichwahrscheinlich)

$$t(n) = \frac{1}{|\mathbf{I}_n|} \sum_{I \in \mathbf{I}_n} T(I)$$

- Tatsächliche Eingabeverteilung kann in der Praxis aber stark von uniformer Verteilung abweichen
- Dann Berücksichtigung der partiellen Wahrscheinlichkeiten

$$t(n) = \sum_{I \in \mathbf{I}_n} p_I \cdot T(I)$$

- Dabei muss $\sum_{I \in \mathbf{I}_n} p_I = 1$ gelten
- Probleme: Partielle Wahrscheinlichkeiten schwer zu bestimmen und zu berechnen

8.4 Besser Sortieren

- Unsere bisherigen Sortieralgorithmen (BUBBLESORT und INSERTIONSORT) habe eine Komplexität in $\mathcal{O}(n^2)$
- Wir suchen nach besseren Verfahren
- **Idee**: Teile und herrsche
- Beispiele:
  - QUICKSORT (C.A.R. Hoare, 1960)
  - MERGESORT (J.v. Neumann, 1945)
- Betrachten ersteres

Quicksort

Algorithm QUICKSORT
Require: $array = \{e_1, e_2, \ldots, e_n\}$
Ensure: $\forall i \in \{1, n-1\}, e_i \leq e_{i+1}$

1: **procedure** QSORT($array$)
2: **if** $|array| \leq 1$ **then**
3: **return** $array$
4: **end if**
5: select and remove a pivot value $pivot$ from $array$
6: $less \leftarrow [\,]$
7: $greater \leftarrow [\,]$;
8: **for** $e \in array$ **do**
9: **if** $e \leq pivot$ **then**
10: append e to $less$
11: **else**
12: append e to $greater$
13: **end if**
14: **end for**
15: **return** concatenate(QSORT($less$), $pivot$, QSORT($greater$))
16: **end procedure**

Quicksort (Forts.)

- Durch den eingebauten Listentyp lässt sich QUICKSORT gut in Python implementieren

```
# qsort -- QuickSort algorithm

def qsort(list):
  if list == []:
    return []
  else:
    pivot = list[0]
    less=[]
    greater=[]
    for x in list[1:]:
        if x<pivot: less.append(x)
        else: greater.append(x)
    return qsort(less) + [pivot] + qsort(greater)
```


Komplexität

- Die Rekursionstiefe ist bei QUICKSORT nicht fest
- Wie sieht der Worst-Case aus?
- Pivot-Element ist immer kleinstes oder größtes Element der Liste ➡ Rekursionstiefe ist $n - 1$
- Bei jedem Rekursionsaufruf müssen dann $i = |array| - 1$ Elemente betrachtet werden, was $n - Rekursionstiefe$ entspricht
- $\Theta\left(\sum_{i=1}^{n-1}(i)\right) = \Theta(\frac{n(n-1)}{2}) = \Theta(n^2)$
- Der Worst-Case ist also **nicht** besser als z.B. bei BUBBLESORT
- **Aber**: Es lohnt hier die Betrachtung des Average-Case

Average Case

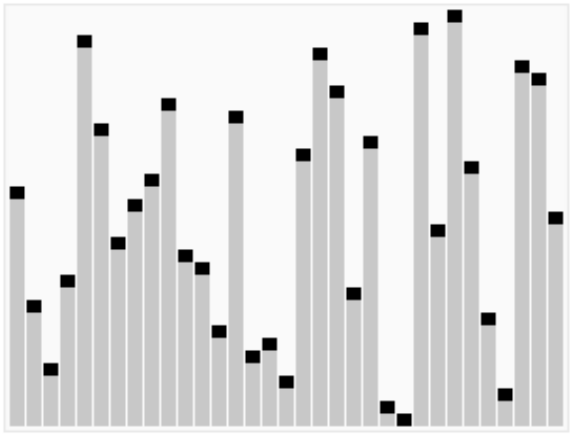
- Bilden Mittelwert über alle Permutationen
- Bei n Elementen gibt es $n!$ Permutationen
- Nur bei wenigen ist ein beliebig ausgewähltes Pivot-Element ungünstig
- Im Mittel gilt: QUICKSORT ist in $\mathcal{O}(n \cdot \log n)$
- Auf Herleitung/Beweis wird hier verzichtet

Speicherkomplexität

- Neben der Laufzeit ist der Speicherverbrauch eine kritische Ressource
- Es werden die **gleichen Komplexitätsmaße** benutzt, wie bei der Zeit
- Beispiel QUICKSORT:
  - Bei jeder Rekursion werden neue Arrays angelegt
  - Der Speicherbedarf ist in jeder Rekursionstiefe n
  - Rekursionstiefe ist maximal $n - 1$
- Speicherkomplexität von QUICKSORT ist in $\mathcal{O}(n^2)$
- **Aber**: Durch clevere Implementation kann QUICKSORT *in place* arbeiten
  - Gut geeignet für C
  - ➡ Dann Speicherkomplexität in $\mathcal{O}(n)$

Speicherkomplexität (Forts.)

- Ablauf von Quicksort mit *in-place*-Implementation



Erkenntnisse

- Die Komplexität kann ein Problem praktisch unlösbar machen
- Komplexität ist vor allem dann wichtig, wenn eine Lösung skalieren soll
- Durch geschickte Wahl eines Algorithmus kann unter Umständen die Komplexität reduziert werden
- Gleiches gilt für die Auswahl der benutzten Datenstrukturen
- (Viel) mehr davon in VL „**Algorithmen und Datenstrukturen**“

Aufgaben

Aufgabe 8.1

Beim Streichhölzchenspiel wird eine Anzahl von Streichhölzern auf den Tisch gelegt. Dann nehmen die beiden Spieler im Wechsel eine Anzahl von Streichhölzern weg, mindestens eines und maximal fünf. Der Spieler, der das letzte Steichholz nimmt (nehmen muss) hat verloren.

a) Entwickeln Sie eine Strategie (Algorithmus), der Ihre Gewinnchance vergrößert oder sogar garaniert!

b) Unter welchen Bedingungen (Anzahl Hölzer, erster oder zweiter Spieler) können Sie einen Sieg garantieren?

c) Wie ändert sich der Algorithmus, wenn der Spieler mit dem letzen Hölzchen gewinnt?

Aufgaben (Forts.)

Aufgabe 8.2

Eine Anzahl von Wassergläsern stehen auf einem Tisch, manche davon verkehrt herum (Beispiel siehe Abbildung). Es sollen alle Gläser „richtig“ herum hingestellt werden, wobei stets **zwei** Gläser auf einmal gedreht werden müssen (nie ein einzelnes Glas).

Unter welchen Startkonfigurationen gibt es eine Lösung?