Datenstrukturen und Algorithmen

Vorlesung 1: Algorithmische Komplexität

Joost-Pieter Katoen

Lehrstuhl für Informatik 2
Software Modeling and Verification Group

04. April 2023

Übersicht

Übersicht

Algorithmen

Algorithmus

Eine wohldefinierte Rechenvorschrift, um ein Problem durch ein Computerprogramm zu lösen.

Beispiel (Algorithmen)

Quicksort, Heapsort, Lineare und Binäre Suche, Graphalgorithmen.

Löst ein Rechenproblem, beschrieben durch

- die zu verarbeitenden Eingaben (Vorbedingung / precondition) und
- die erwartete Ausgabe (Nachbedingung / postcondition),

mithilfe einer Folge von Rechenschritten.

Beispiel Rechenproblem: Sortieren

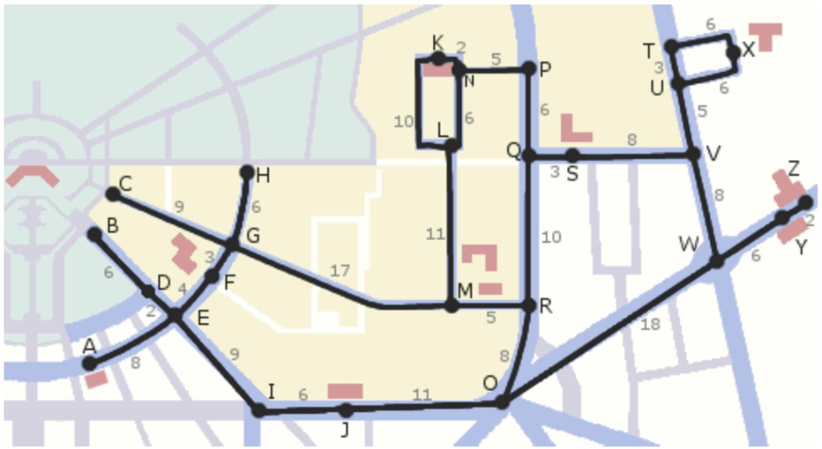
Beispiel

Eingabe: Eine Folge von n natürlichen Zahlen $\langle a_1, a_2, \ldots, a_n \rangle$ mit $a_i \in \mathbb{N}$.

Ausgabe: Eine Permutation (Umordnung) $\langle b_1, b_2, \ldots, b_n \rangle$ der Eingabefolge, sodass $b_1 \leqslant b_2 \leqslant \ldots \leqslant b_n$.

Andere Rechenprobleme: kürzester Weg

Andere Rechenprobleme: kürzester Weg



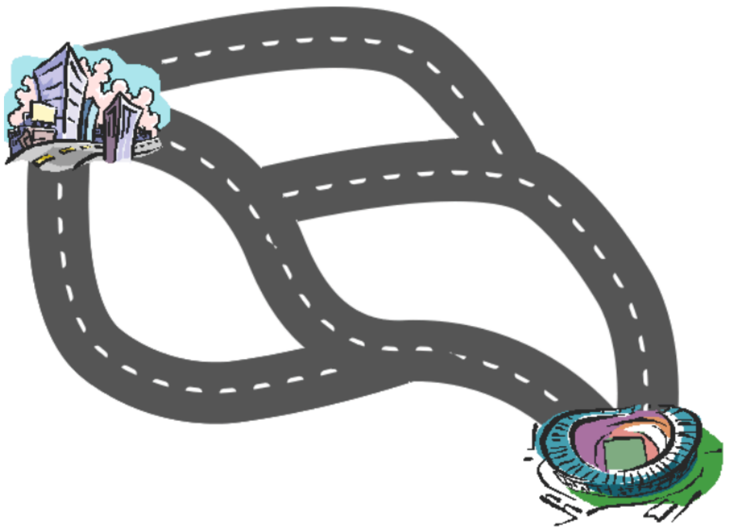
Andere Rechenprobleme: kürzester Weg

Beispiel (kürzester Weg)

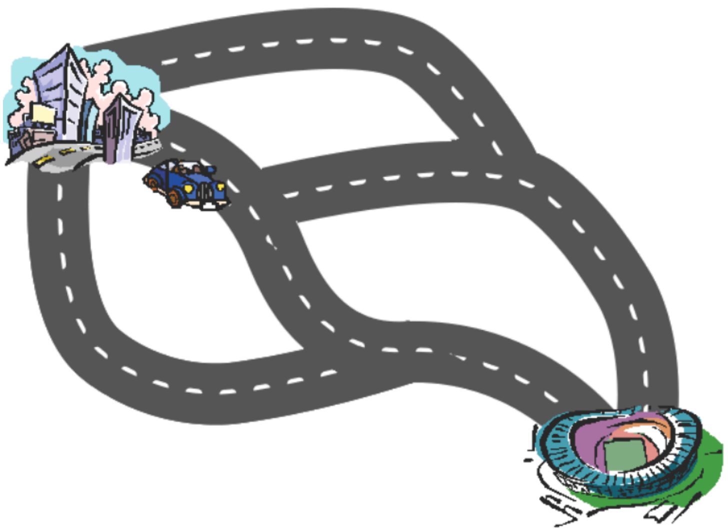
Eingabe:
1. Eine Straßenkarte, auf welcher der Abstand zwischen jedem Paar benachbarter Kreuzungen eingezeichnet ist,
2. eine Startkreuzung s und

Ausgabe: Ein kürzeste Weg von s nach z.

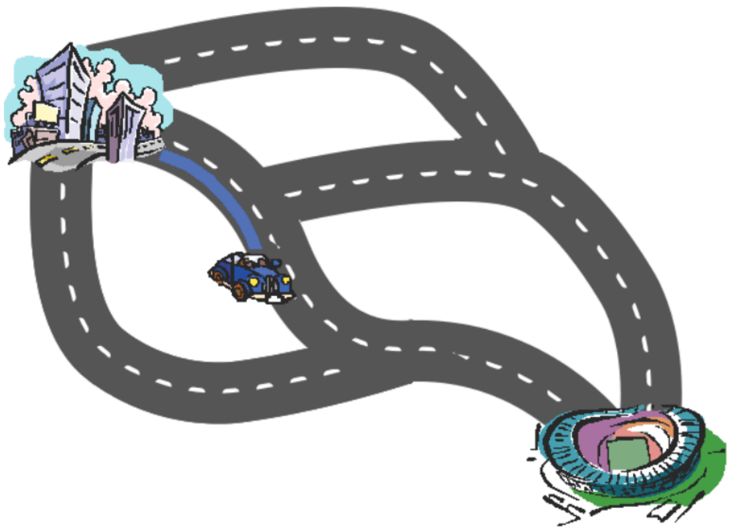
Andere Rechenprobleme: maximale Flüsse



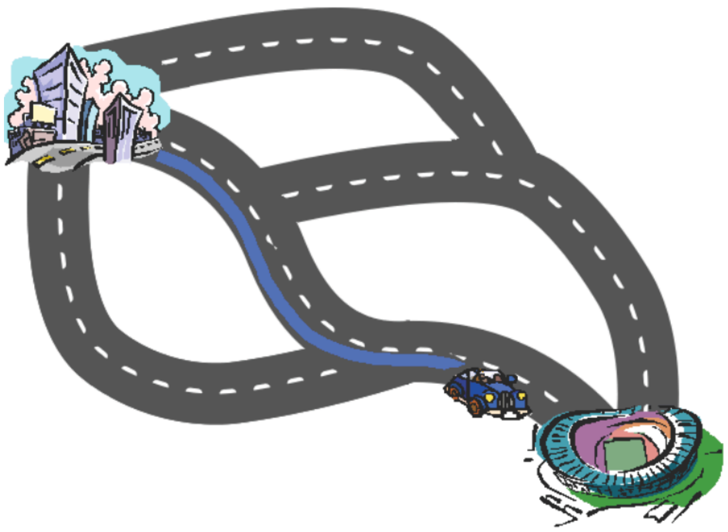
Andere Rechenprobleme: maximale Flüsse



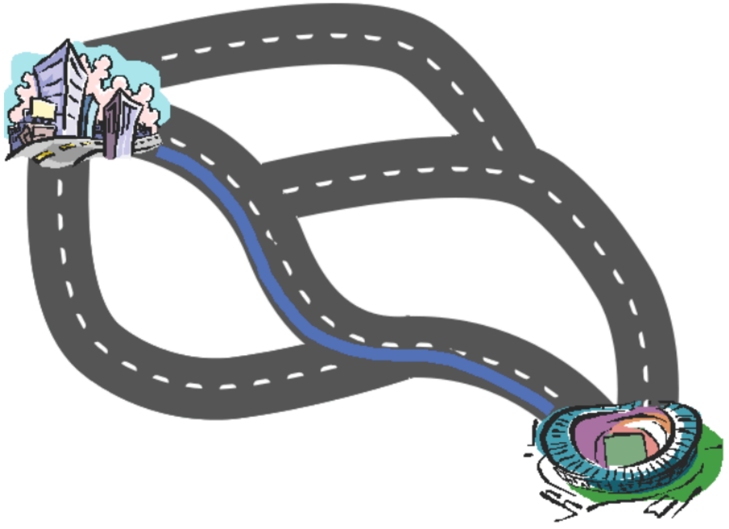
Andere Rechenprobleme: maximale Flüsse



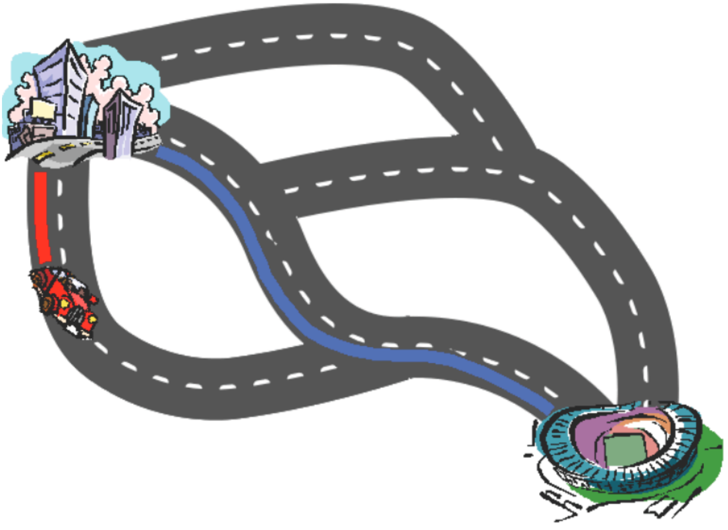
Andere Rechenprobleme: maximale Flüsse



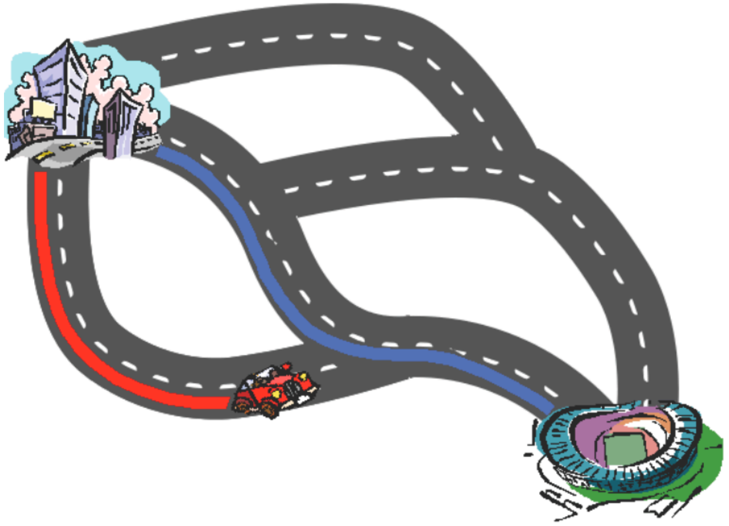
Andere Rechenprobleme: maximale Flüsse



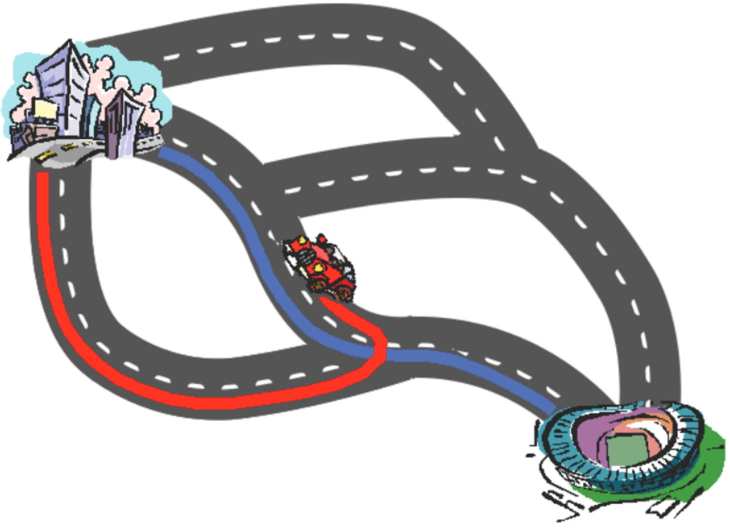
Andere Rechenprobleme: maximale Flüsse



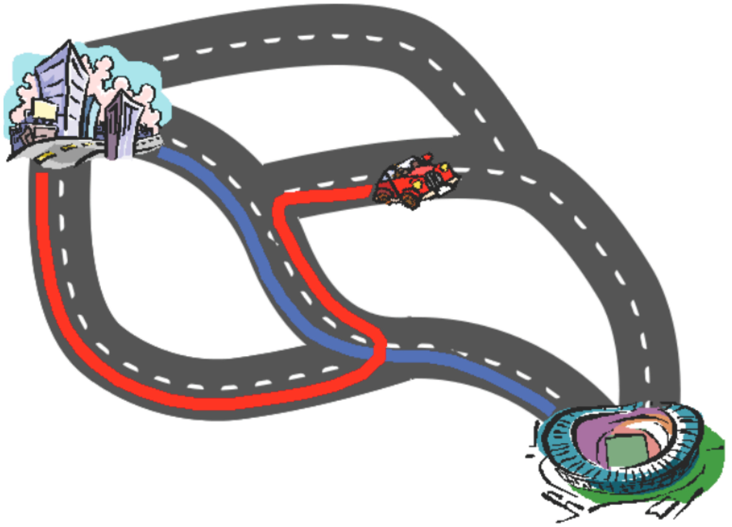
Andere Rechenprobleme: maximale Flüsse



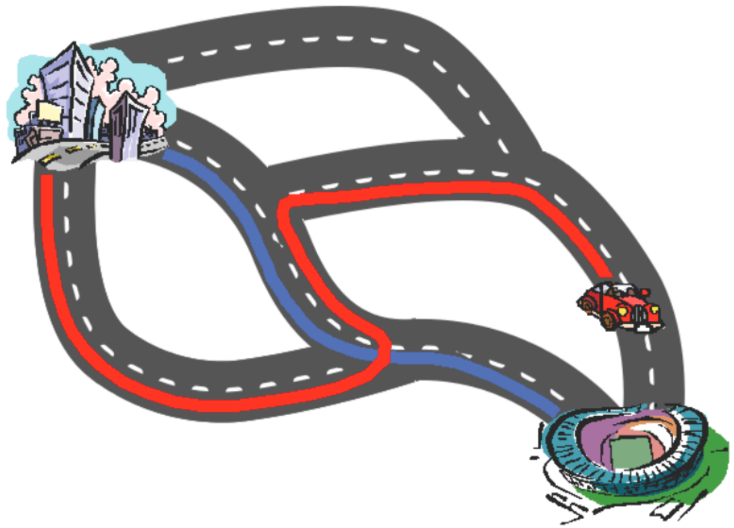
Andere Rechenprobleme: maximale Flüsse



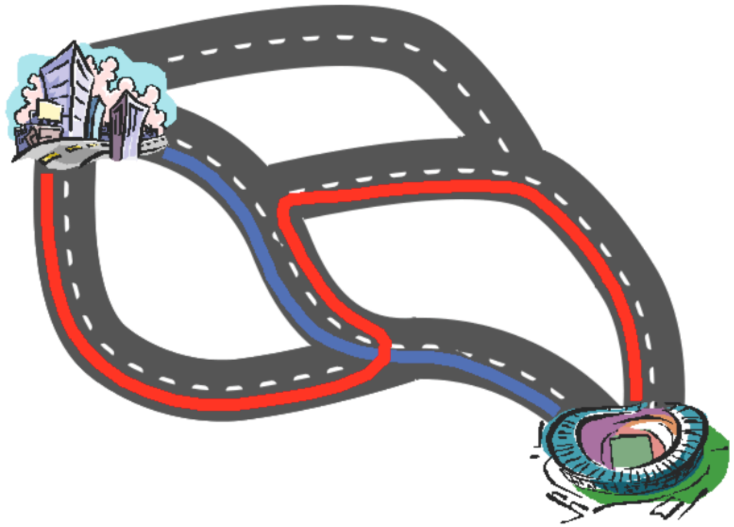
Andere Rechenprobleme: maximale Flüsse



Andere Rechenprobleme: maximale Flüsse



Andere Rechenprobleme: maximale Flüsse



Andere Rechenprobleme: maximale Flüsse

Beispiel (maximale Flüsse)

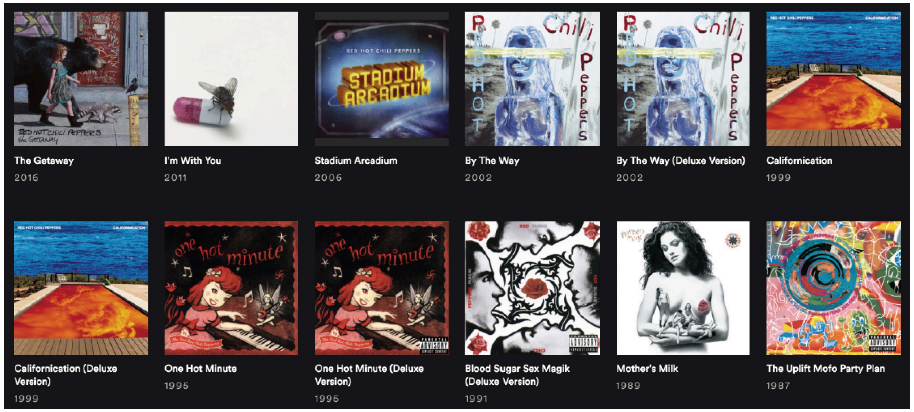
Eingabe:
1. Eine Straßenkarte, auf der die Kapazität der Straßen eingezeichnet ist,
2. eine Quelle und
3. eine Senke.

Ausgabe: Die maximale Rate, mit der Material (= Zuschauer) von der Quelle bis zur Senke (= Stadion) transportiert werden kann, ohne die Kapazitätsbeschränkungen der Straßen zu verletzen.

Andere Rechenprobleme: das CD-Brennproblem

Andere Rechenprobleme: das CD-Brennproblem

Betrachte einige Schallplatten von Red Hot Chili Peppers:



Wie bekommen wir eine Kompilation ihrer Songs auf einige CDs?

Andere Rechenprobleme: das CD-Brennproblem

Beispiel (CD-Brennproblem)

Eingabe:
1. $N \in \mathbb{N}$ Songs, Song i dauert $0 < n_i \leqslant 80$ Minuten,
2. $k \in \mathbb{N}$ CDs, jeweils mit Kapazität: 80 Minuten.

Ausgabe: k CDs gefüllt mit einer Auswahl der N Songs, sodass
1. die Songs in chronologische Reihenfolge vorkommen und
2. die totale Dauer der (verschiedenen) ausgewählten Songs maximiert wird,

wobei ein Song komplett auf eine CD gebrannt werden soll.

Algorithmen

Kernpunkte

- Korrektheit: Bei jeder Eingabeinstanz stoppt der Algorithmus mit der korrekten Ausgabe?
- Eleganz
- Effizienz: Wieviel Zeit und Speicherplatz wird benötigt?

Effiziente Algorithmen verwenden effektive Datenstrukturen.

Datenstrukturen

Datenstruktur

Ein mathematisches Objekt zur Speicherung von Daten.

Man spricht von einer Struktur, da die Daten in einer bestimmten Art und Weise angeordnet und verknüpft werden, um den Zugriff auf sie und ihre Verwaltung geeignet und effizient zu ermöglichen.

Beispiele (Datenstrukturen)

Array, Baum, Kellerspeicher (stack), Liste,
Warteschlange (queue), Heap, Hashtabelle ...

Effizienz von Algorithmen – Kriterien

Wichtige Kriterien sind (für eine bestimmte Eingabe):

- die benötigte Zeit, Zeitkomplexität
- der benötigte Platz. Platzkomplexität

Zeitkomplexität ≠ Platzkomplexität ≠ Komplexität des Algorithmus

Ziel

Beurteilung der Effizienz von Algorithmen unabhängig von

- verwendetem Computer, Programmiersprache, Fähigkeiten des Programmierers usw.

Effizienz von Algorithmen – Elementare Operation

Die Analyse hängt von der Wahl der elementaren Operationen ab, etwa:

- „Vergleich zweier Zahlen“ beim *Sortieren* eines Arrays von Zahlen.
- „Multiplikation zweier Fließkommazahlen“ bei *Matrixmultiplikation*.

Elementare Operationen

- Anzahl der elementaren Operationen sollte eine gute Abschätzung für die Anzahl der Gesamtoperationen sein.
- Anzahl der elementaren Operationen bildet die Basis zur Bestimmung der Wachstumsrate der Zeitkomplexität bei immer längeren Eingaben.

Effizienz von Algorithmen – Beispiele

Technologie führt nur zu Verbesserung um einen konstanten Faktor:

Beispiel

Selbst ein Supercomputer kann einen „schlechten“ Algorithmus nicht retten: Für genügend große Eingaben gewinnt *immer* der schnellere Algorithmus auf dem langsameren Computer.

Beispiel

Typische Laufzeiten (bis auf einen konstanten Faktor) für Eingabelänge n:

1	konstant	$n \cdot \log n$	
$\log n$	logarithmisch	n^2	quadratisch
n	linear	2^n	exponentiell

Zeitkomplexität in der Praxis I

Beispiel (Tatsächliche Laufzeiten)

	Komplexität				
Länge n	$33n$	$46n \log n$	$13n^2$	$3{,}4n^3$	2^n
10	0,00033 s	0,0015 s	0,0013 s	0,0034 s	0,001 s
10^2	0,0033 s	0,03 s	0,13 s	3,4 s	$4{\cdot}10^{16}$ y
10^3	0,033 s	0,45 s	13 s	0,94 h	
10^4	0,33 s	6,1 s	1300 s	39 d	
10^5	3,3 s	1,3 m	1,5 d	108 y	

Benötigte Zeit (s = Sekunde, h = Stunde, d = Tag, y = Jahr)

- Der Einfluss großer konstanter Faktoren nimmt mit wachsendem n ab.

Zeitkomplexität in der Praxis II

Beispiel (Größte lösbare Eingabelänge)

	Komplexität				
Verfügbare Zeit	$33n$	$46n \log n$	$13n^2$	$3{,}4n^3$	2^n
1 s	30 000	2000	280	67	20
1 m	1 800 000	82 000	2170	260	26
1 h	108 000 000	1 180 800	16 818	1009	32

Größte lösbare Eingabelänge

- Eine 60-fach längere Eingabe lässt sich nicht durch um den Faktor 60 längere Zeit (oder höhere Geschwindigkeit) bewältigen.

Schnellere Computer...

Sei N die größte Eingabelänge, die in fester Zeit gelöst werden kann.

Frage

Wie verhält sich N, wenn wir einen K-mal schnelleren Rechner verwenden?

#Operationen benötigt für Eingabe der Länge n	Größte lösbare Eingabelänge
$\log n$	N^K
n	$K \cdot N$
n^2	$\sqrt{K} \cdot N$
2^n	$N + \log K$

Übersicht

Idee

Wir betrachten einen gegebenen Algorithmus A.

Worst-Case Laufzeit

Die Worst-Case Laufzeit von A ist die von A maximal benötigte Anzahl elementarer Operationen auf einer *beliebigen* Eingabe der Länge n.

Best-Case Laufzeit

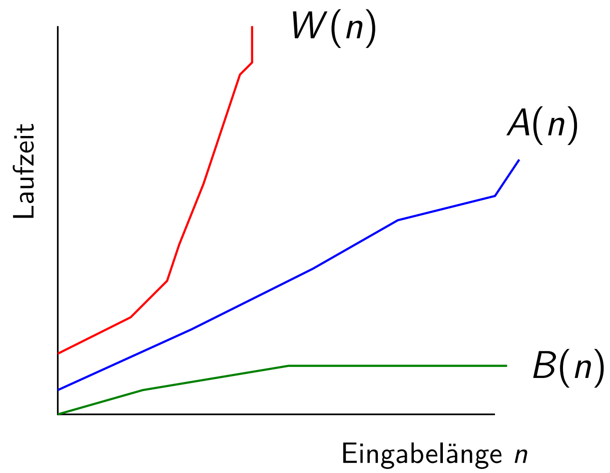
Die Best-Case Laufzeit von A ist die von A minimal benötigte Anzahl elementarer Operationen auf einer *beliebigen* Eingabe der Länge n.

Average-Case Laufzeit

Die Average-Case Laufzeit von A ist die von A durchschnittlich benötigte Anzahl elementarer Operationen auf einer *beliebigen* Eingabe der Länge n.

Alle drei sind Funktionen: Laufzeit in Abhängigkeit von der Eingabelänge!

Beispiel



Formale Definition (I)

Einige hilfreiche Begriffe

$$D_n = \text{Menge aller Eingaben der Länge } n$$
$$t(I) = \text{für Eingabe } I \text{ benötigte Anzahl elementarer Operationen}$$
$$\Pr(I) = \text{Wahrscheinlichkeit, dass Eingabe } I \text{ auftritt}$$

Woher kennen wir:

$t(I)$? – Durch Analyse des fraglichen Algorithmus.

$\Pr(I)$? – Erfahrung, Vermutung (z. B. „alle Eingaben treten mit gleicher Wahrscheinlichkeit auf“).

Formale Definition (II)

Worst-Case Laufzeit

Die Worst-Case Laufzeit von A ist die von A maximal benötigte Anzahl elementarer Operationen auf einer *beliebigen* Eingabe der Länge n:

$$W(n) = \max\{\, t(I) \mid I \in D_n \,\}.$$

Best-Case Laufzeit

Die Best-Case Laufzeit von A ist die von A minimal benötigte Anzahl elementarer Operationen auf einer *beliebigen* Eingabe der Länge n:

$$B(n) = \min\{\, t(I) \mid I \in D_n \,\}.$$

Formale Definition (II)

Average-Case Laufzeit

Die Average-Case Laufzeit von A ist die von A durchschnittlich benötigte Anzahl elementarer Operationen auf einer *beliebigen* Eingabe der Länge n:

$$A(n) = \sum_{I \in D_n} \Pr(I) \cdot t(I)$$

Lineare Suche

Rechenproblem

Eingabe: Array `E` mit `n` Einträgen sowie das gesuchte Element `K`.

Ausgabe: Ist `K` in `E` enthalten?

```
def linSearch(e, K):
  for i in range(len(e)):
    if e[i] == K:
      return True # Oder: return i
    return False # Nicht gefunden
```


Lineare Suche – Analyse

Elementare Operation

Vergleich einer ganzen Zahl `K` mit Element `E[index]`.

Menge aller Eingaben

D_n ist die Menge aller Permutationen von n ganzen Zahlen, die ursprünglich aus einer Menge $N > n$ ganzer Zahlen ausgewählt wurden.

Zeitkomplexität

- $W(n) = n$, da n Vergleiche notwendig sind, falls `K` nicht in `E` vorkommt (oder wenn `K == E[n-1]`).
- $B(n) = 1$, da ein Vergleich ausreicht, wenn `K` gleich `E[0]` ist.
- $A(n) \approx \frac{1}{2}n$?, da im Schnitt `K` mit etwa der Hälfte der Elemente im Array `E` verglichen werden muss? – Nein.

Lineare Suche – Average-Case-Analyse (I)

Zwei Szenarien

1. `K` kommt nicht in `E` vor.
2. `K` kommt in `E` vor.

Zwei Definitionen

1. Sei $A_{K \notin E}(n)$ die Average-Case-Laufzeit für den Fall "`K` nicht in `E`".
2. Sei $A_{K \in E}(n)$ die Average-Case-Laufzeit für den Fall "`K` in `E`".

$$A(n) = \Pr\{\texttt{K in E}\} \cdot A_{K \in E}(n) + \Pr\{\texttt{K nicht in E}\} \cdot A_{K \notin E}(n)$$

Der Fall "K in E"

- Nehme an, dass alle Elemente in E unterschiedlich sind.
- Damit ist die Wahrscheinlichkeit für K == E[i] gleich $\frac{1}{n}$.
- Die Anzahl benötigter Vergleiche im Fall K == E[i] ist $i+1$.
- Damit ergibt sich:

$$\begin{aligned} A_{K\in E}(n) &= \sum_{i=0}^{n-1} \Pr\{\texttt{K == E[i]} \mid \texttt{K in E}\} \cdot t(\texttt{K == E[i]}) \\ &= \sum_{i=0}^{n-1} \left(\frac{1}{n}\right) \cdot (i+1) \\ &= \left(\frac{1}{n}\right) \cdot \sum_{i=0}^{n-1} (i+1) \\ &= \left(\frac{1}{n}\right) \cdot \frac{n(n+1)}{2} \\ &= \frac{n+1}{2}. \end{aligned}$$

Herleitung

$$A(n) = \Pr\{\texttt{K in E}\} \cdot A_{K\in E}(n) + \Pr\{\texttt{K nicht in E}\} \cdot A_{K\notin E}(n)$$

$$\Big|\ A_{K\in E}(n) = \frac{n+1}{2}$$

$$= \Pr\{\texttt{K in E}\} \cdot \frac{n+1}{2} + \Pr\{\texttt{K nicht in E}\} \cdot A_{K\notin E}(n)$$

$$|\ \Pr\{\text{nicht } B\} = 1 - \Pr\{B\}$$

$$= \Pr\{\texttt{K in E}\} \cdot \frac{n+1}{2} + (1 - \Pr\{\texttt{K in E}\}) \cdot A_{K\notin E}(n)$$

$$|\ A_{K\notin E}(n) = n$$

$$= \Pr\{\texttt{K in E}\} \cdot \frac{n+1}{2} + (1 - \Pr\{\texttt{K in E}\}) \cdot n$$

$$= n \cdot \left(1 - \frac{1}{2}\Pr\{\texttt{K in E}\}\right) + \frac{1}{2}\Pr\{\texttt{K in E}\}$$

Lineare Suche – Average-Case-Analyse

Endergebnis

Die Average-Case-Zeitkomplexität der linearen Suche ist:

$$A(n) \;=\; n \cdot \left(1 - \frac{1}{2}\Pr\{\texttt{K in E}\}\right) + \frac{1}{2}\Pr\{\texttt{K in E}\}$$

Beispiel

Wenn $\Pr\{\texttt{K in E}\}$

- $= 1$, dann $A(n) = \frac{n+1}{2}$, d. h. etwa 50% von E ist überprüft.
- $= 0$, dann $A(n) = n = W(n)$, d. h. E wird komplett überprüft.
- $= \frac{1}{2}$, dann $A(n) = \frac{3 \cdot n}{4} + \frac{1}{4}$, d. h. etwa 75% von E wird überprüft.

Übersicht

Übersicht (Teil I)

1. Algorithmische Komplexität
2. Asymptotische Effizienz
3. Elementare Datenstrukturen
4. Suchen
5. Rekursionsgleichungen
6. Sortieren: in-situ, Mergesort, Heapsort, Quicksort
7. Binäre Suchbäume
8. Rot-Schwarz-Bäume
9. Hashing

Präsenzübung (07. Juni 2023)

Übersicht (Teil II)

1. Elementare Graphenalgorithmen
2. Minimale Spannbäume
3. Kürzeste-Pfade-Algorithmen
4. Maximale Flüsse
5. Dynamische Programmierung
6. Algorithmische Geometrie

Klausur (27. Juli 2023)

Literatur

Die Vorlesung orientiert sich im Wesentlichen an diesem Buch:

Thomas H. Cormen, Charles E. Leiserson, Ronald Rivest, Clifford Stein:

Algorithmen - Eine Einführung

R. Oldenbourg Verlag , 4. Auflage (2013).

Vorlesungen

- Die Vorlesungen finden im SoSe 2023 alle in Präsenz statt
- Keine virtuell Beteiligung an die Vorlesungen möglich
- DSAL Vorlesungen sind als Video (Aufnahmen VideoAG 2015) verfügbar:

  `https://video.fsmpi.rwth-aachen.de/15ss-dsal`

Vorlesungsfolien

- Für jede Vorlesung gibt es aktuelle Vorlesungsfolien
  - die können ggf. leicht abweichen von den Folien in der Vorlesung
  - es gilt: der Inhalt der aktuellen Folien ist gültig
- Die Folien werden an den Vorlesungstagen (Di.+Do.) bereit gestellt
  wobei Feiertage (z.B. Himmelfahrt, Fronleichnam) ausfallen
- 20 Vorlesungstermine:

April	4.4, 6.4, 11.4, 13.4, 18.4, 20.4, 27.4
Mai	4.5, 9.5, 11.5, 16.5, 17.5, 23.5, 25.5,
Juni	6.6, 13.6, 20.6, 22.6, 29.6,
Juli	4.7

Frontalübung

Frontalübung: Mi. 12:30–14:00

Erste Frontalübung: Mi. 13. April

Inhalt: Extra Erklärungen, extra Beispiele, extra Stoff, Q&A

Übungsbetrieb

Übungsgruppen

- etwa 25 Übungsgruppen: verschiedene Uhrzeiten Mo.–Fr.
- Assistenten:
  Daniel Cloerkes, Ira Fesefeldt,
  Hannah Mertens und Tobias Winkler.

Anmeldung für die Übungsgruppen

Anmeldung zum Übungsbetrieb über RWTHonline bis:

Montag, 10. April 2023, 23:59 Uhr (Aachener Zeit).

Übungsbetrieb (1)

- Übungsblätter (Hausaufgaben), die wöchentlich ausgegeben werden
- Ausgabe spätestens Mittwochs (bis 23:59 Uhr) über RWTHmoodle
- Bearbeitung (und Abgabe) in Gruppen
  von je drei Studierende aus dem selben Tutorium[1]
- Abgabe der Hausaufgaben:
  - Frist: Mittwochs um 23:59 Uhr
- Jedes Übungsblatt hat $\geqslant 1$ klar identifizierte "Klausur"-Aufgabe
  - Aufgaben die vom Niveau her einer Klausuraufgabe entsprechen

[1] Suche nach Abgabepartner/inn/en über Forum im Moodle Lernraum ab 11.04.

Übungsbetrieb (2)

Format Hausaufgaben:

- Elektronisch als pdf-Datei
- Oben mit Übungsgruppe, Name+Matrikelnummer beschriften
- Einreichung über RWTHmoodle

Musterlösungen:

- Werden in den Tutorien vorgerechnet
- Werden als pdf-Datei zur Verfügung gestellt

Benotung Hausaufgaben:

- Hausaufgaben werden korrigiert, sind keine Klausurzulassung

Übungsbetrieb (3)

Bearbeitung der Übungsblätter wird dringend empfohlen!

Tipp: ohne selbst zu üben, keine Chance die Klausur zu bestehen

Übungsbetrieb (4)

Wichtige Termine

Anmeldung:	10. April 2023
Abgabe 0, Übungsblatt	12. April 2023
1.Übungsblattl:	12. April 2023
Abgabe 1. Übungszettel:	19. April 2023
2.Übungsblatt:	19. April 2023
Abgabe 2. Übungsblatt:	26. April 2023
	......
Frontalübung:	Mittwoch, 12:30–14:00 ab 12. April 2023
Präsenzübung:	Mittwoch, 7. Juni 2020 (abends)

Das 0. Übungsblatt wird **selbtständig** bearbeitet
und wird **nicht** korrigiert. Es wird also **nicht** eingereicht.

Python Modul

Umfang DSAL:	7+1 Kredits (ECTS)
Neu ab SoSe 2023:	DSAL enthält einen Selbstlernmodul Python
Umgang Python Modul:	1 ECTS
Prüfung:	integrales Bestandteil der Präsenzübung
Vorbereitung:	einige Aufgaben in den Übungsblätter
Vorlesung:	alle Algorithmen werden in Python beschrieben

Prüfung

Die Prüfung ist eine schriftliche Klausur von 120 Minuten.

Zulassungskriterium Klausur

Mindestens 50% der in der Präsenzübung (PÜ) erreichbaren Punkte

Bonusregelung

1. Bei $\geqslant$ 70% der Gesamtpunkte über alle Hausaufgaben bis der PÜ: 10 Punkte Bonus in der Präsenzübung
2. Wenn ausserdem $\geqslant$ 70% der Gesamtpunkte über alle Hausaufgaben ab PÜ bis Klausur:
   - Klausur bestanden? Eine Notenstufe besser (außer bei 1.0)
   - Klausur nicht bestanden? 10 Punkte Bonus[2]

[2] Beim erzielen von 50–59 Punkte (nicht bestanden), werden 60 Punkte angerechnet (bestanden mit Note 4,0).

Wichtige Termine

Präsenzübung:	Mittwoch, 7. Juni 2023 (90 Minuten)
Klausur:	Donnerstag, 27. Juli 2023 (120 Minuten)
Wiederholung PÜ:	Donnerstag, 27. Juli 2023
Wiederholungsklausur:	Mittwoch, 13. September 2023

Anmeldung zur Prüfung über RWTHonline

Zusammenfassung Anmeldungen

Vorlesung: Keine Anmeldung erforderlich.

Übungsgruppen:

Anmeldung zum Übungsbetrieb über RWTHonline
Frist: bis Mo., 10. April 2023 um 23:59 Uhr.

Präsenzübung:

Anmeldung zur Präsenzübung über RWTHmoodle
Frist: zwischen 11. April und 24. Mai um 23:59 Uhr

Klausur:

Anmeldung zur Prüfung über RWTHonline
Frist: ab Mo., 15. Mai, bis 1. Juli um 23:59 Uhr

Fragen

1. Erster Ansprechpartner ist der Tutor Ihrer Übungsgruppe
2. Diskussionsforum über RWTHmoodle
3. E-Mail: `dsal@lists.rwth-aachen.de`
4. Bitte keine Email direkt an mich oder die Assistenten
5. Zur Not einen Zoom-Termin mit mir vereinbaren

Nächste Vorlesung

Nächste Vorlesung

(Grün)Donnerstag 6. April, 16:30, H.01