

Datenstrukturen und Algorithmen

- Kostenabschätzung und Vereinfachungen
- Induktion
- Suchalgorithmen
 - Binäre Suche
 - Interpolationssuche
 - Exponentielle Suche
 - Lineare Suche
- Sortieralgorithmen
 - Bubble Sort
 - Sortieren durch Auswahl
 - Sortieren durch Einfügen
 - Heapsort
 - Mergesort
 - Reines 2-Wege-Mergesort
 - Natürliche 2-Wege-Mergesort
 - Quicksort
 - Randomisiertes Quicksort
- Stack
- Queue
- Hashing
 - Universelles Hashing
 - Offenes Hashing
 - sondieren
 - Lineares Sondieren
 - Quadratisches Sondieren
 - Double Hashing
- Suchbäume
 - Natürliche Suchbäume
 - AVL-Baum
- Graphentheorie
 - Definitionen
 - Adjazenzmatrix
 - Reflexive-transitive-Hülle
 - Zusammenhangskomponente
 - Breiten-Tiefensuche
 - Topologische Sortierung
 - Kürzeste Wege (Dijkstra, Bellman-Ford, Floyd-Warshall, Johnson)
 - Minimaler Spannbaum (Prim)
- Dynamische Programmierung
- Amortisierte Analyse

Kostenabschätzung

- \mathcal{O} (=Obere Schranke) (=wächst höchstens)
- Θ (=Mittlere Schranke, wenn f in \mathcal{O} und Ω liegt) (=wächst genau)
- Ω (=Untere Schranke) (=wächst mindestens)
- n_0 (=ab welcher Zahl gilt Abschätzung)
- C (=verdoppelter Konstante)
- Addition $\underbrace{g_1 = \mathcal{O}(f) / g_2 = \mathcal{O}(f)}$

$$g_1 + g_2 = \mathcal{O}(f) / C = c_1 + c_2 / n = \max(n_0^1, n_0^2)$$

- Regel von L'Hôpital:

- $\frac{f}{g} = 0 \Rightarrow f = \mathcal{O}(g)$
- $\frac{f}{g} = c \Rightarrow f = \Theta(g)$
- $\frac{f}{g} = \infty \Rightarrow g = \mathcal{O}(f)$

Vereinfachungen

- $\mathcal{O}(5 \log_b(n^2)) = \mathcal{O}(\log_b(n^2)) = \mathcal{O}(2 \log_b(n)) = \mathcal{O}(\log_b(n)) = \mathcal{O}(\log(n))$
- 8^{2^n} = Konstant
- $\sqrt[n]{n} = n^{1/n}$
- $\log(2^n) = n \cdot \log(2) = n$
- $\log^4(2^n) = n$
- $\binom{n}{4} = n^4$
- $\log(n) < \sqrt{n}$
- $\log_x(n)^Y$ immer sehr klein
- $n \cdot \log(n) > n$
- $n! \geq 3^n$
- $j = j/3 \Rightarrow \log_3(j)$
- $i \cdot i \leq n \Rightarrow \sqrt{n}$

Induktion

- Induktionsanfang: Zeige, dass $n=1$ gilt
- Induktionshypothese: Wir nehmen an, Aussage sei für jedes n gültig
- Induktionsschritt: Wir zeigen, dass $n=n+1$ auch gilt

Binäre Suche

- Divide and Conquer
- Wir schauen in der Mitte, falls Suchelement kleiner ist, suchen wir links weiter, sonst rechts
- $n = 2^k$ Schritte $\Rightarrow O(\log(n))$
- Muss vor sortiert sein

Interpolationssuche

- Der statische Aufteilungsfaktor der Binären-Suche wird angepasst
- Man schätzt, wo Element sich etwa befindet
- Wenn man weiß, wo Element ist $O(\log/\log(n))$
- Schlechterster Fall $\Omega(n)$

Exponentielle Suche

- Länge des Arrays ist unbekannt
- Ich gehen in exponentiellen Schritten vom Anfang an her, weg
- Wenn Suchelement kleiner ist, habe ich Schraufce für Binäre Suche

Lineare Suche

- Wenn Array unsortiert ist
- Alles ein Mal ansehen $\Rightarrow O(n)$

- Suchen im unsortierten Array $O(n)$
- Suchen im sortierten Array $O(\log(n))$
Entscheidungsbaum

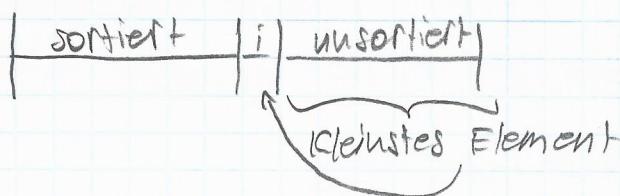
Sortieren

- In-situ / In-place (=Wenn man nur zusätzlich $\log(n)$ Zusatzspeicher benötigt)
- Stabil (=Wenn Reihenfolge gleicher Elemente beim Sortieren beibehalten werden)

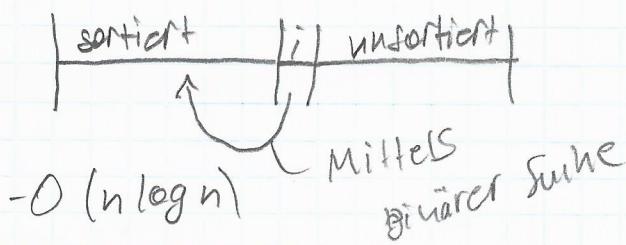
Bubblesort

3 7 5 1 4 3 1 5 4 7 1 3 4 5 7 ✓
 3 5 7 1 4 3 1 4 5 7
 3 5 1 7 4
 3 5 1 4 7
 - $\Theta(n^2)$

→ Wenn Array absteigend sortiert ist

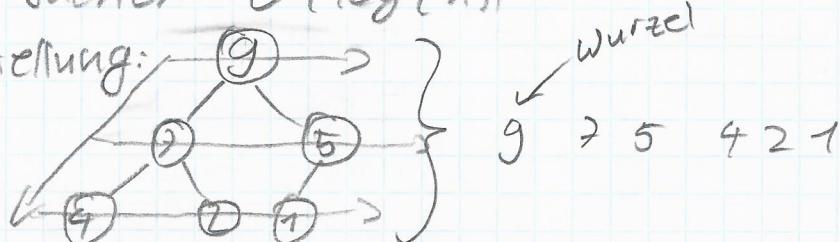
Sortieren durch Auswahl

- $\Theta(n^2)$, jedoch gehen wir nur ein Mal durch Array durch, somit weniger Schlüsselvergleiche

Sortieren durch Einfügen

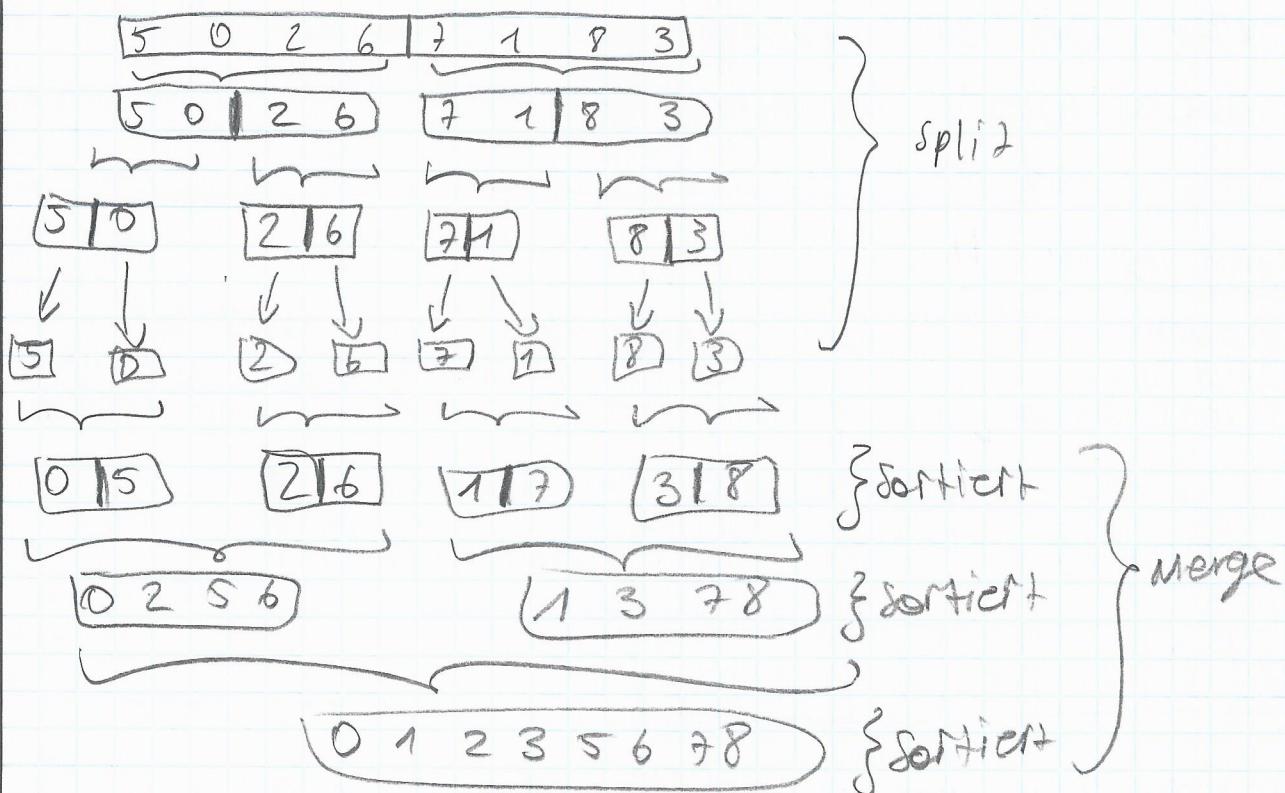
Heapsort

- Ist ein binärer Suchbaum
- Baum von links nach rechts gefüllt
- Höhe: $\log(n)$ ($n = \text{Anzahl Elemente}$)
- Anzahl Blätter: $\frac{n}{2}$
- Max-Heap: Wurzel hat max. Element, alle Kinder sind kleiner
- Min-Heap: Wurzel hat min. Element, alle Kinder sind größer
- Wenn Heap-Bedingung verletzt ist
 - Versickern ⚡
 - Hochblubbern ⚡
- Um Heap zu erstellen, setzen wir einfach alle zählt ein, und stellen dann Heap-Bedingung wieder her, für jedes kleine Heap (von unten nach oben)
 - ↪ 1x versickern max $O(\log(n))$ (=Restore-Heap-condition)
 - ↪ Das für alle Elemente $O(n \cdot \log(n))$
- Element suchen: $O(\log(n))$
- Arraydarstellung:

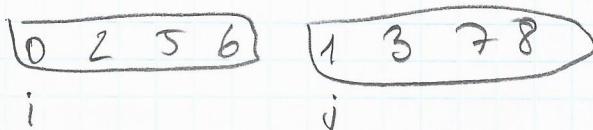


Merge-Sort

- wir berechnen kleine Teilbereiche und fügen dann zusammen



- Vergleiche: $\log(n)$
- Bewegungen: $\log(n)$
- Speicherplatz: n ($=$ Nicht gut)
- Verschmelzen zweier sortierten Teilfolgen
 - Mittels zwei Zeiger



↪ Wenn i kleiner ist, wird i in neues Array gespeichert, und eins nach rechts bewegt.

- Natürliches 2-Wege Mergesort

↪ Man schaut Array an, und bestimmt bereits sortierte Teilstücke (= Rüns)

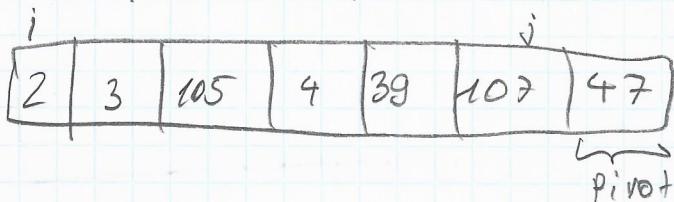
↪ Dadurch ist split unnötig

- Reines 2-Wege-Mergesort

- Man geht Array von links nach rechts durch
- Man speichert verschmelzte Teilfolgen über direkt wieder in Array

Quicksort

- Braucht nicht so viel Extraspeicher wie Mergesort
- Pivotelement unterteilt Array in links und rechts
 - Links (- kleiner als Pivotelement)
 - Rechts (- größer als Pivotelement)



1. Man schaut zuerst von links, bis wir größeres Element als Pivot haben
 2. Dann von rechts, bis wir kleineres Element als Pivotelement haben
 3. Beide Zeigerelemente vertauschen
 4. Das machen wir so lange, bis Zeiger $i=j$
 5. Rekursiver Aufruf
- Laufzeit, Durchschnitt: $n \cdot \log(n)$, dafür wenig Extrplatz
 - Laufzeit, wenn Pivot max/min Element ist: $\Theta(n^2)$
 - Randomisiertes Quicksort: wähle Pivot zufällig
 - Schlechtes Quicksort, wenn ein Teilstück erheblich größer ist, als anderes
 - Gutes Quicksort, wenn Pivot am Schluss immer in die Mitte getauscht wird

Analyse der Sortieralgorithmen

- Es geht nicht besser, als $O(n \cdot \log(n))$
 - Blatttiefe ist die Anzahl Schlüsselvergleiche, welche ich für ein Element im worst-case machen muss
- ↳ und dies für jedes Element

Stack

- Push (= Pusht Element auf Stack)
- Pop (= Poppt Element vom Stack und gibt es zurück)
- Top (= gibt oberstes Element aus, ohne es zu entfernen)
- $O(1)$

Queue

- Enqueue (= Fügt Element in Warteliste hinten an)
- Dequeue (= Nimmt Element von 1. Warteschlangenposition weg und gibt es aus)
- $O(1)$

Priority-Queue

- Insert ersetzt Enqueue, welche zusätzlich noch die Priorität angibt
- Extract-Max (= Entfernt Element mit max. Priorität)

Multistack

- Multipop(x) (= Poppt gleichzeitig x Elemente vom Stapel)

Hashing

- Wir wollen keine Kollisionen und Häufungen
- Offenes Hashverfahren
 - Wenn Kollision, benutzen wir eine Sondierungsfunktion
 - Somit entsteht keine Liste an einem Arrayplatz, wo sich die doppelten Elemente stapeln
- Universelles Hashing
 - Randomisiert aus Hashing-Datenbank auswählen
- Quadratisches Sondieren
 - Wenn man viele gleiche Elemente hat, führt dies zu sekundären Häufungen
$$\underbrace{1^2 / -1^2 / 2^2 / -2^2 / \dots}_{\text{Wird der eigentlichen Adresse abgezogen}} \quad (\lceil j/2 \rceil)^2 - (-1)^{j+1}$$
- Lineares Sondieren
 - Ich schaue immer beim linken Nachbar
- Double Hashing
 - Bei Kollision gibt es eine zweite Hashfunktion

Ausrichtung nach Suchhäufigkeit

- Frequency-Count: Die Elemente werden nach Abrufhäufigkeit angeordnet
 - Worst-Case: Immer letztes Element suchen
- Transpose: Je Aufruf wird Element mit Vorgänger getauscht
- Move-to-Front: Gesuchtes Element an Anfang der Liste

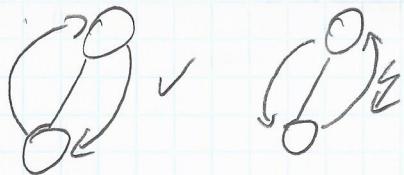
Natürlicher Suchbaum

- Rechtes Kind größer als Vater
- Linkes Kind kleiner als Vater
- ganz unten rechts (=max Wert)
- ganz unten links (=min Wert)
- in Order
 - linkes Kind
 - Elternteil
 - rechtes Kind
- PreOrder (Eltern immer zuerst angeben)
 - Elternteil
 - linkes Kind
 - Rechtes Kind
- PostOrder (Eltern am Schluss)
 - linkes Kind
 - Rechtes Kind
 - Elternteil
- Operationen
 - Suchen = Einfach durch Baum durchgehen
 - Einfügen = Durch Baum durchgehen, bis man an Blatt kommt
 - Löschen = . Knoten hat kein Kind => Einfach löschen
 - Knoten hat ein Kind => Kind wird neuer Knoten
 - Knoten hat zwei Kinder => symmetrischer Nachfolger suchen

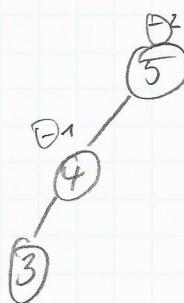
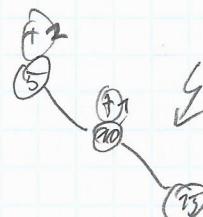


AVL-Baum

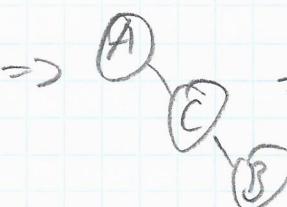
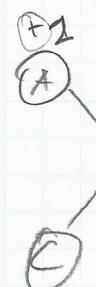
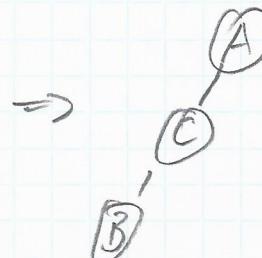
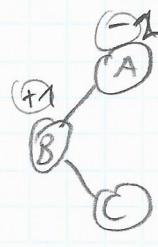
- Verhindert Denaturierung
- Höhe darf sich nur um maximal ± 1 unterscheiden
- Arten von Rotationen (links - / rechts +)



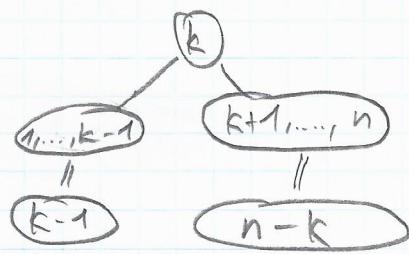
• Einfachrotation:



• Doppelrotation:



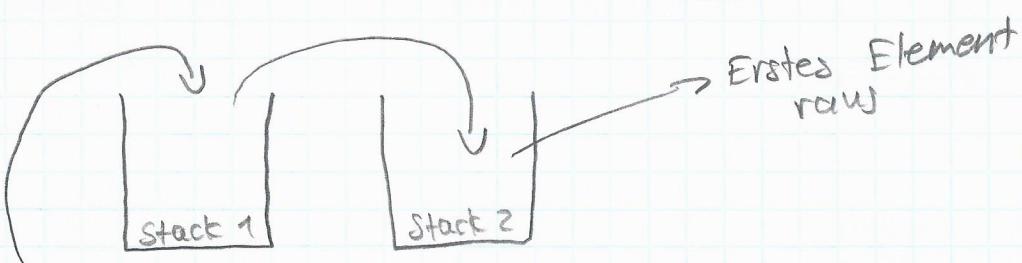
Anzahl verschiedener Suchbäume



} das n Mal für jeden Teilbaum

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

Quene mit Stack implementieren



Enqueue(x)

Dequeue()

→ Nur wenn Stack 2 leer ist, wird von Stack 1 alles übergepusht.

Graphentheorie

- Graph besteht aus: $G = (V, E)$

- Knoten ○ (= Vertex)

- Kanten — (= Edges)

- Eulerischer Zyklus: Jeder Knoten hat geraden Grad

- Jede Kante ein Mal besuchen, und am Schluss beim Anfangsknoten wieder landen

- Eulerkreis: Höchstens zwei ungerade Grade hat

- Wenn man alle Kanten besucht, aber am Schluss nicht beim Anfangsknoten landet

- Gerichteter Graph

- Jede Kante hat eine Richtung $\textcircled{a} \rightarrow \textcircled{b}$ (a, b)

- Ungerichteter Graph

- Kann auch als gerichteter Graph dargestellt werden
- $\textcircled{a} - \textcircled{b}$ (a, b), (b, a)

- Gewichteter Graph

- Ist ein gerichteter Graph

- Jede Kante hat zusätzlich noch ein Gewicht

- Multigraph

- Wenn zwei Knoten durch mehrere Kanten verbunden sind



- Sehr selten

- Vollständiger Graph

- Wenn jeder Knoten mit jedem anderen Knoten verbunden ist

- Bipartiter Graph

- Wenn es zwei Mengen gibt, welche ineinander keine Kanten haben



- Grad

- Ungerichtet: Alle Kanten die ankommen: $\deg^- = \deg^+$

- Gerichtet: -Eingangsgrad: \deg^-

- -Ausgangsgrad: \deg^+

- Schleifen: Wenn \textcircled{a} (= Zyklus der Länge 1)

- Zyklus: $\textcircled{a} \rightarrow \textcircled{a}$

- Zusammenhängend: Wenn jeder Knoten jeden anderen

Knoten erreichen kann

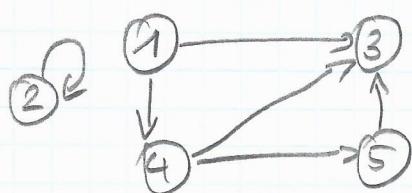
• Jeder zusammenhängende Graph hat mindestens $n-1$ Kanten

- Kreis:  (=Kreis der Länge 3)

Datenstrukturen Graphen

- Adjazenzmatrix

- Größe: $n \times n$
- Wenn Matrix symmetrisch \Leftrightarrow ungerichteter
- 1 = Kante / 0 = Keine Kante

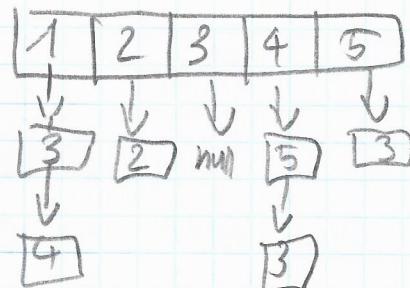
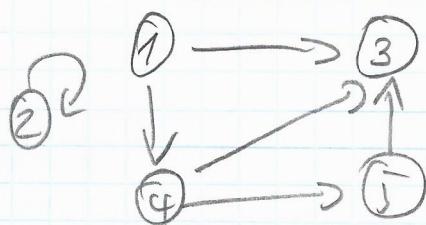


	1	2	3	4	5
1	0	0	1	1	0
2	0	1	0	0	0
3	0	0	0	0	0
4	0	0	1	0	1
5	0	0	1	0	0

schleife

- Adjazenzliste

- Verbraucht nicht so viel Speicherplatz: $|V| + |E|$



Wege von Adjazenzmatrix ablesen

- A^T = Wege der Länge T, wenn $A_{ij} = 1$ ist
- Wie finde ich kürzesten Weg von $i \rightarrow j$
↳ so lange A^T , bis $A_{ij} \neq 0$ ist, dann haben wir einen Weg
- Dreieck finden: A^3 und Diagonale zusammenzählen und dann durch 6 rechnen

Relationen von Graphen

- Reflexiv: Jeder Knoten hat Schleife
 - Diagonale = 1
- Symmetrisch: Ungerichtet
- Transitiv: $(u, v) \& (v, z) \Rightarrow (u, z)$

Reflexive-transitive-Hülle f. gerichtete Graphen

- Erreichbarkeitsgraph



Durchlaufen von Graphen

- Depth-First-Search (= Tiefensuche)
 - geht zuerst in die Tiefe
 - geht mit Stack

1. Wir werfen Anfangsknoten auf Stapel

2. While Stack ist nicht leer

 2.1. Pop oberster Knoten, und wenn Knoten noch nicht besucht, markiere als gesucht und push alle Nachbarn auf Stapel, die noch nicht besucht sind

- Breitensuche

- Wir suchen zuerst alle Knoten mit Distanz 1/2/3/...

- geht mit Queue

1. Wir werfen Anfangsknoten auf Queue

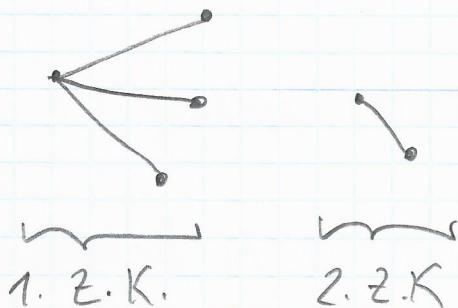
2. While Stack ist nicht leer

- 2.1. Dequeue vorderster Knoten und markiere ihn als besucht

- 2.2. Werfe alle seinen noch nicht besuchten Nachbarn hinten in die Queue

Zusammenhangskomponente für ungerichtete Graphen

- Eine Zusammenhangskomponente ist eine Äquivalenzklasse der transitiven Hülle



- Berechnung mittels Breiten-/Tiefensuche
 ↳ Jeder Neustart ist Zusammenhangskomponente

- Transitiv-Hülle für gerichtete Graphen

- Marshall-Algorithmus $O(n^3)$

- Transitiv-Hülle für ungerichtete Graphen

- Tiefen-/Breitensuche $O(|V| + |E|)$

Topologische Sortierung für gerichtete Graphen

- Man fängt bei Knoten ohne Eingangskanten an.
- Falls es das nicht gibt, ist Graph zyklisch
- Wir brauchen DAG (=Directed acyclic graph)
- Wenn man Knoten aufgeschrieben hat, denkt man sich dieser weg, und schaut dann, welcher Knoten wiederum keinen Eingangsknoten hat

Kürzester Weg in Graphen

- Mit Kantenkosten: 1

↳ Breitensuche löst Problem

$\rightarrow O(|E| + |V| \cdot \log |V|)$
Fibonacci-Heap

- Mit verschiedenen positiven Kantenkosten

- Dijkstras Algorithmus

- Alle Distanzen auf ∞ setzen, außer Startpunkt = 0

- Immer Entfernung aktualisieren, falls kleiner

- Mit verschiedenen positiven/negativen Kantenkosten

- Bellman - Ford Algorithmus

- Bei negativen Zyklen existiert kein kürzester Weg

- Startpunkt Kosten 0, alle anderen Kosten ∞

- Algorithmus erkennt negative Zyklen

- Wir relaxieren $|V|-1$ Mal + 1 Mal

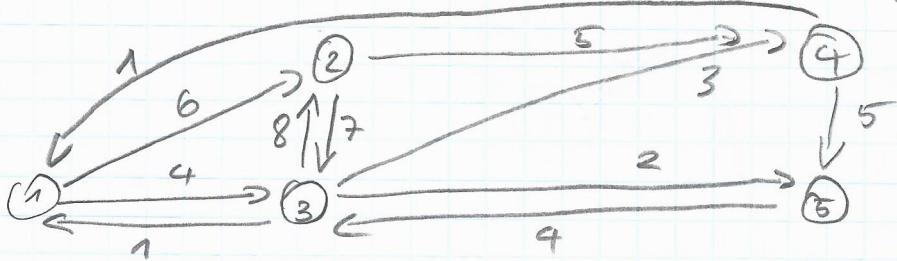
- Somit relaxieren wir alle Knoten $|V|-1$ Mal

\hookrightarrow Wenn sich dann noch etwas verändert, negativer Zyklus

$\rightarrow O(|V| \cdot |E|)$

- Floyd-Warshall dynamische Programmierung

- Wir brauchen Tableau mit Kantengewichte



	1	2	3	4	5
$k=1$	0	6	4	0	0
$k=2$	2	0	0	7	5
$k=3$	3	1	0	3	2
$k=4$	4	1	0	7	15
$k=5$	5	0	1	4	0

$\uparrow \uparrow \uparrow \uparrow \uparrow$
 $k=1 \ k=2 \ k=3 \ k=4 \ k=5$

$k=0$ (Wir zählen k hoch bis $|V|$)

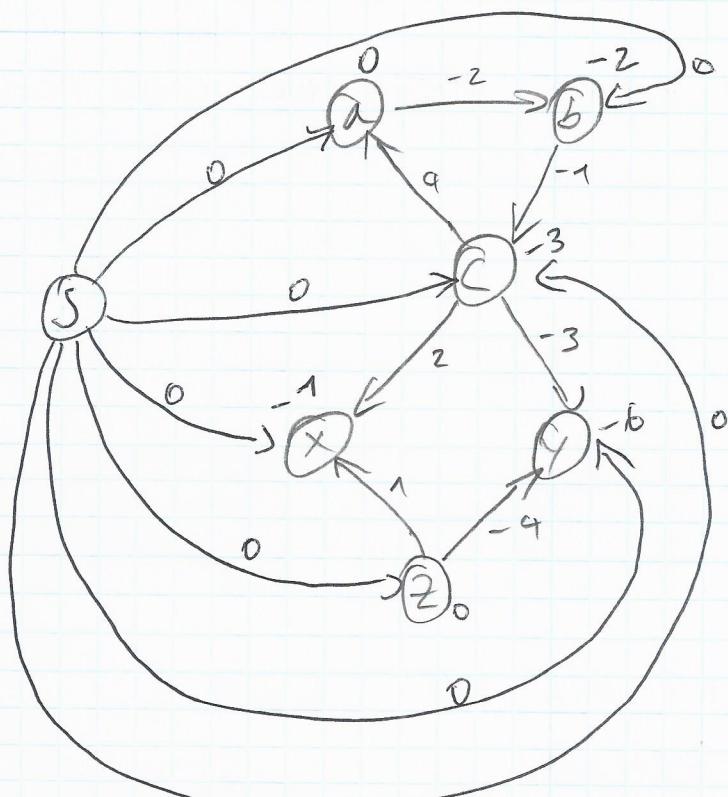
$k=1 \quad k \in \{1, \dots, M\}$ (\hookrightarrow i.d.F 5)

$$\text{dist}[i][j] \leftarrow \text{dist}[i][j] + \text{dist}[k][j]$$

- Wir schauen immer $\min(m_{ij}, m_{ik} + m_{kj})$

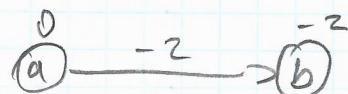
- Johnson

- Wir eliminieren die negativen Gewichte und benutzen dann Dijkstra



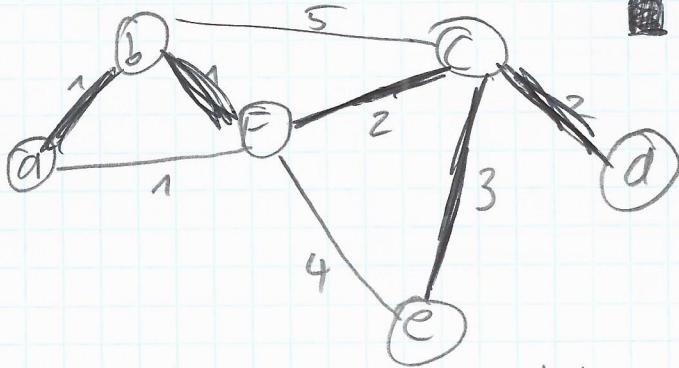
Entfernung von negativen Gewichten - Formel:

Neu: Mitte + Ende - Anfang



$$\text{Neu: } -2 + 0 - (-2) = 0$$

Minimaler Spannbaum

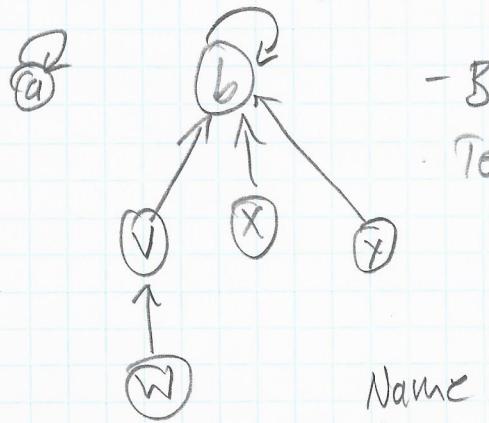


Minimaler Spannbaum

- Kruskal - Greedy - Algorithmus

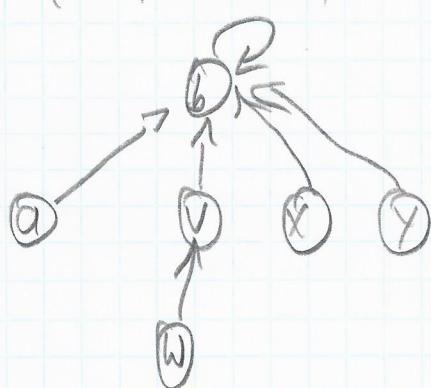
- Nehme immer die billigste Kante, die keinen Kreis verursacht

- Union - find - Datenstruktur



- B ist Repräsentant dieser Teilmenge

Name der neuen Untermenge, also B
union(a, find(y))



- zusammenhängender Graph mit $|V|$ Knoten

- Spannbaum davon besitzt

- $[|V|]$ -Knoten

- $[|V|-1]$ -Kanten

Längste aufsteigende Teilfolge

- Wir haben Folge: 3 2 4 6 5 7 1

- Tabelle: n lang

• 1: Kleinste Teilfolge der Länge 1

1 2 3 4 5 6 7 (= 3 2 4 6 5 7 1)

(3)

2 (4) 6

1 (5) (7)

1 2 3 4 5 6 7 8 9 (= 2 9 3 18 5 46 7)

(2) 9

1 (3) 8

5

(4) (6) (7)

Längste gemeinsame Teilfolge

- Editierdistanz: Wie kommt man von Liebe nach Leder

	L	F	Fr	Fo	Frc	Fisch
L	0	1	2	3	4	5
F	1	0	1	2	3	4
Fr	2	1	1	2	3	4
Fo	3	2	2	2	3	4
Frc	4	3	3	2	3	4
Fisch	5	4	4	3	2	3
Fisch	6	5	5	4	3	2

Möglichkeiten, Wechselgeld zurückzugeben

		Mögliche Beträge				
		0	5	10	15	20
Münzen	0	1	0	0	0	0
	5	1	1	1	1	1
	10	1	1	2	2	3
	20	1	1	2	2	4
	50	1	1	2	2	4

Man hat hier folgende Münzen zur Verfügung: 0, 5, 10, 20, 50

1 Möglichkeit, nichts zurückzugeben

Subset-Sum

- Menge von Zahlen: 2, 3, 7, 8, 10
- Gibt es Kombination, um Untergruppe zu erstellen i.d.R 11?

	0	1	2	3	4	5	6	7	8	9	10	11
2	T	F	T	F	F	F	F	F	F	F	F	F
3	T	F	T	T	F*	T**	F	F	F	F	F	F
7	T	F***	T	T	F	T	F	T	T	T	F	
8	T	F	T	T	F	T	F	T	T	F	T	T
10	T											

→ Hälften ist möglich!

$$\frac{1 \cdot 8}{2 \cdot 11 - 8} = 3 \quad \{ 11$$

$$* 4 - 3 = 1$$

→ Wir schauen bei 1 nach \Rightarrow False

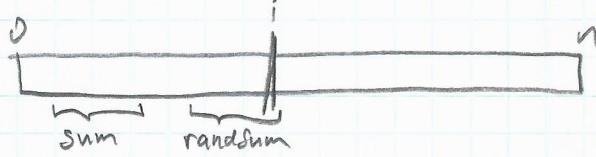
$$** 5 - 3 = 2$$

→ Wir schauen bei 2 nach \Rightarrow True

$$*** 1 - 3 = -2$$

→ Negativ, wir übernehmen von oben

Maximum Subarray Sum



- Sum = Speichert maximale Summe
- randSum = Summe mit a_i
 - Wenn randSum > sum \Rightarrow sum = randSum
 - Wenn randSum < 0 \Rightarrow RandSum = 0

$$\begin{matrix} \text{sum} \\ \text{randsum} \end{matrix} \quad ? = 0$$

Für $i=0$ bis n

$$\text{randsum} = \text{randsum} + a_i$$

Wenn randSum > sum \Rightarrow sum = randSum

Wenn randSum < 0 \Rightarrow randSum = 0

Gebe sum aus

Rucksack-Problem

- Maximales Gewicht: S
- Reihe von Gegenständen: a_1, a_2, \dots, a_n
 - Gewicht und Wert
- Wert muss maximiert werden

Man darf nur 1. Gegenstand verwenden	1	2	3	4	...	S
	0	0	0	0	0	0

1. & 2. Gegenstand	1	0	max.	<ul style="list-style-type: none"> • Entweder nehme ich i. Gegenstand • Aktuelles Gewicht - a_i (Gewicht) 		
	2	0		Value a_i + dort nachsehen Value		
alle Gegenstände	3	0		<ul style="list-style-type: none"> • Ich nehme Gegenstand nicht: 	0	
	⋮	⋮				

Pseudopolynomiel

- $O(n \cdot w)$

Es gibt
n verschiedene

→ Dies ist eigentlich nur eine Zahl
(= Binärdarstellung $\log(w)$)

↳ wir betrachten aber $1, 2, 3, \dots, w$

⇒ Wenn ich Zahl w habe, aber ich w verschiedene Werte durchgehen muss, ist es pseudopolynomiel

Greedy-Algorithmen

- Greedy stopft das Maximum rein, bis es nicht mehr geht.
- Es wird im Kleinen jeweils die beste Entscheidung getroffen, ohne Rücksicht auf Konsequenzen für den gesamten Suchverlauf

Strassen Matrixmultiplikation

a	b	
c	d	
e	f	$ea + eb +$ $fc + fd$
g	h	$ga + gb +$ $hc + hd$

- Normale Matrixmultiplikation: n^3

- Strassen Matrixmultiplikation: $n^{2.83}$

Multiplikation von Karatsuba / Ofman

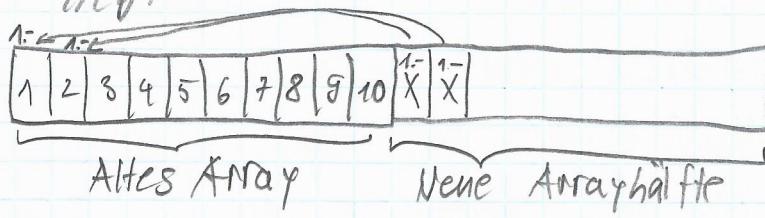
$$\begin{array}{r} ab \\ 62 \\ \hline \end{array} \cdot \begin{array}{r} cd \\ 37 \\ \hline \end{array}$$

$$\begin{array}{r}
 14 & (=b \cdot d) \\
 14 & (=b \cdot d) \\
 16 & (= (a-b) \cdot (d-c)) \\
 18 & (=a \cdot c) \\
 18 & (=a \cdot c) \\
 \hline
 2294
 \end{array}$$

Amortisierte Analyse

- Beispiel Hashing

- Wenn Array voll ist, erstellen wir doppelt so grosses Array und kopieren alle Elemente rüber
- Bankkonto: Guthaben für Rübergangskopieren
- Jedes Element zahlt 3.- CHF ein
 - Für das erste Einfügen (wird sofort verbraucht)
 - Für das Kopieren in das neue Array
 - Für das Kopieren eines anderen Elements, welches eine Arraylänge weiter vorne liegt



Dynamische Programmierung: Aufsteigende Sequenzen

9	27	42	41	48
35	39	8	3	5
12	48	2	38	4
15	47	28	28	6
19	1	25	33	10

- Man fängt mit kleinstem Element in Matrix an
- Man schreibt immer die längste Teilfolge in DP-Tabelle
- Man sucht bei Nachbarn die längste Teilfolge, wenn Nachbar kleiner ist
- Für Backtracking: Man speichert auch noch, welcher Nachbar man genommen hat.

Dynamische Programmierung: Windräder (Abstände)

- Man muss optimalen Platz für Windräder finden, wo Abstand mind. D ist, und es versch. Gewinne gibt
- N verschiedene Plätze mit n versch. Gewinnen
- $1 \times N$ - Tableau
 - Ich schaue immer, dass für das i . Windrad alle anderen Windräder der Reihe maximal ist
 - Um i zu berechnen, schaue ich mir alle Windräder mit Abstand $\geq D$ an, und wähle Windrad mit max. Gewinn aus
 - Dieses Windrad hat bereits auch schon andere Windräder mit max. Gewinn genährt
- Wenn ich fertig bin, gehe ich durch Tableau durch, und suche max. Gewinn
- Backtracking
 - maxGewinn - Aktueller Gewinn Windrad
 - ↳ Ergebnis muss irgendwo im Array vorkommen
 - ↳ Ich mache das selbe immer wieder, und merke, welche Windräder abgezogen wurden
 - ↳ Wenn Differenz = 0, dann bin ich fertig

Maximum-Subarray

a_1, a_2, \dots, a_n

Alg. soll max. Summe berechnen

Alg 1: Alles ausprobieren $\Theta(n^3)$

Alg 2: Vorberechnung Präfixsumme

Alg 3: Divide and Conquer

Divide and Conquer

- Einteilung in kleinere Probleme
- Liste wird so weit wie möglich geteilt
- Paare werden analysiert
↳ und neu geordnet
- Teillisten werden wieder zu Liste kombiniert
 $\Theta(n \log(n))$

Induktionsalg.

- Was muss ich tun $i \rightarrow i+1$

Binäre Suche

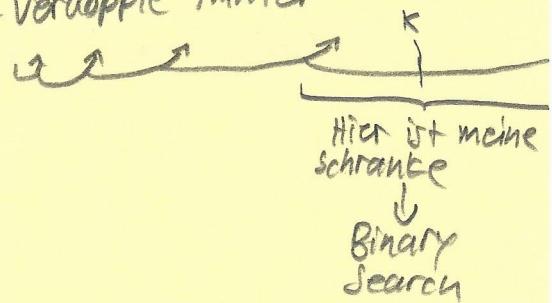
- Nur für geordnetes Array
 - Wir schauen, was in Mitte ist
 - Wenn Element kleiner \rightarrow links
 - Wenn Element größer \rightarrow Rechts
 - Wieder Mitte von Links/Rechts
↳ Bis Element gefunden wurde
 $\Theta(\log(n))$
- $\lfloor \quad \rfloor$ (=Floor)

Interpolationssuche

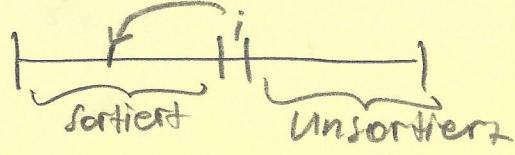
- Ich schätze, wo mein Element ca. ist
- Danach \rightarrow Binary search

Exponential Search

- Länge des Arrays unbekannt
- Ich fange bei an an
- Verdopple immer



Insertion sort

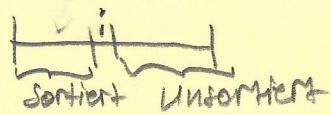


- Man sucht mit Binary Search, wo das i. Element im bereits sortierten Teil des Arrays hingehört

Selection sort

- induktiv

↳ Einmal das Array durchratteln und dann fertig sein



- Ich nehme das kleinste Element aus unsortiert und setze es in i ein

Bubble Sort

- überprüfen, ob Element größer oder kleiner und vertauscht dann

6 1 2 3 4 5 (=unsorted)
 ↗
 1 6 2 3 4 5
 1 2 6 3 4 5
 1 2 3 6 4 5
 1 2 3 4 6 5
 1 2 3 4 5 6 (=sorted)