Algorithmen und Datenstrukturen

by Davy

Einführung

Algorithmen sind systematische Anleitungen zum lösen von Problemen.

" Computer Science is the systematic study of algorithems and data structures, specifically

1) their formal properties
2) their mechanical and linguistic realization
3) their applications. "

Algorithmen vergleichen & bewerten

Praxis: tatsächliche Laufzeit
Theorie: Anzahl elementarer Operationen

Theorie ≠ Praxis
↑ hängt von zusätzlichen Faktoren ab

Idee: konstante Faktoren ignorieren und nur asymptotisches Wachstum in Abhängigkeit von n betrachten.

Zusätzlich muss auch darauf geachtet werden, dass ein Algorithmus nicht zu viel zusätzlichen Speicherplatz benötigt.

Algorithmen, die höchstens $O(\log n)$ zusätzlichen Speicherplatz benötigen, nennt man **in-place** oder **in-situ**.

Formal: asymptotische Notation **O-Notation**

Definition: Für $f: N \to \mathbb{R}^+ = \{y > 0\}$

$$O(f) := \{g: N \to \mathbb{R}^+ \mid \exists C > 0.\ \forall n \in \mathbb{N}.\ g(n) \leq C \cdot f(n)\}$$

in Worten

$O(f)$ ist die Menge aller Funktionen g mit der Eigenschaft, dass eine positive Konstante C existiert, so dass $g(n) \leq C \cdot f(n)$ für alle n gilt.

Berechnung : Theorem 1

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = x \qquad x = \begin{cases} 0 & f \in O(g),\ g \notin O(f) \\ \infty & g \in O(f),\ f \notin O(g) \\ c & g \in O(f),\ f \in O(g) \end{cases}$$

L'Hôspital's Rules

Falls $f(n) \to \infty$ ∧ $g(n) \to \infty$
oder $g(n) \to 0$ ∧ $f(n) \to \infty$
ist $\lim_{n\to\infty} \frac{f(n)}{g(n)} \equiv \lim_{n\to\infty} \frac{f'(n)}{g'(n)}$

Exponentielle Laufzeit $\hat{=}$ schlecht 2^n

Polynomielle Laufzeit $\hat{=}$ gut $n^4 + n^3$

Tricks:
- log Regeln
- umschreiben als $e^{\ln(x)}$ od. $2^{\log_2(x)}$
- komplizierte Terme substitutionieren.

Merke:
- $\log(n) << n^{0.00\ldots1}$
- $n^{100000} << 1.000\ldots1^n$
- $\log_2(n) \in O(\log_3(n))$ und $\log_3(n) \in O(\log_2(n))$

Für Divide and Conquer Algorithmen ist folgendes Theorem nützlich:

Master Theorem (U4): Let $a, C > 0$ and $b \geq 0$ be constants and $T: \mathbb{N} \mapsto \mathbb{R}^+$ a func. for all even $n \in \mathbb{N}$.

$$T(n) \leq a \cdot T\left(\frac{n}{2}\right) + C \cdot n^b \qquad (1)$$

Then for all $n = 2^k, k \in \mathbb{N}$,

- $b > \log_2 a$, $T(n) \leq O(n^b)$
- $b = \log_2 a$, $T(n) \leq O(n^{\log_2 a} \cdot \log n)$
- $b < \log_2 a$, $T(n) \leq O(n^{\log_2 a})$

If $T(n)$ is increasing, 2^k can be dropped. If (1) holds true for $=$, then we replace O with Θ.

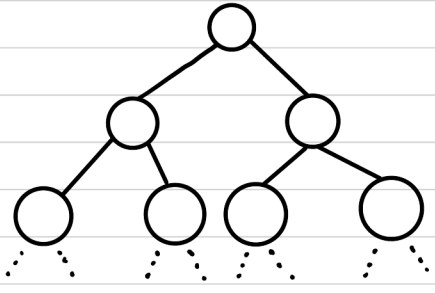
Big Θ und Ω

O beschreibt die obere Grenze für die Laufzeit.
Ω beschreibt die untere Grenze für die Laufzeit (schneller geht nicht)

Θ gibt die genau Laufzeit und existiert nur wenn $\Theta(f) = O(f) \cap \Omega(f)$.

Wenn wir ein Problem als einen Baum zeichnen können, ist es einfach ein Ω zu finden.

Bsp. Suchen in Array



Wir wissen, dass die Anzahl Blätter $\leq 2^h$ ist. Es gibt höchstens n Permutationen vom Array, d.h. n Anzahl Blätter.

$\Rightarrow$ worst case $\leftrightarrow$ höhe Baum h

$$n \leq 2^h \quad \Rightarrow \quad h \geq \log_2(n) \geq \Omega(\log n)$$

Höhe ist gleich einer Operation vom Algorithmus

Induktionsbeweis

Idee: Wir zeigen das etwas für 0 od. 1 geht. Danach nehmen wir an, es ginge auch für n, wenn wir nun folgern das es auch für n+1 gilt, haben wir unsere Annahme bewiesen.

Induktionsvermutung

Induktionsanfang

Induktionshypothese

Induktionsschritt

Beispiel:

Induktionsvermutung: $\forall n \in \mathbb{N}, n > 0 : \sum_{i=0}^{n} i = \frac{n \cdot (n+1)}{2}$

Induktionsanfang: / **Base Case:** $n = 1$ $\quad 0+1 = \frac{1 \cdot (1+1)}{2}$ ist war

Induktionshypothese: / **Induction Hypothesis:** Wir nehmen an, es gelte $\sum_{i=0}^{n} i = \frac{n \cdot (n+1)}{2}$ für ein $n \in \mathbb{N} : n > 0$

$\left(\exists n \in \mathbb{N}, n > 0 : \sum_{i=0}^{n} i = \frac{n \cdot (n+1)}{2}\right)$

Induktionsschritt: / **Induction Step:**

$$\sum_{i=0}^{n+1} i = \sum_{i=0}^{n} i + (n+1) \overset{\text{I.V.}}{=} \frac{n \cdot (n+1)}{2} + (n+1)$$

(I.V. ← Induktionsvermutung I.H.)

$$= \frac{n \cdot (n+1) + 2(n+1)}{2} = \frac{n^2 + 3n + 2}{2} = \frac{(n+1) \cdot (n+2)}{2}$$

$$= \frac{(n+1) \cdot (n+1+1)}{2}$$ q.e.d.

Invariante

Eine Invariante wird benutzt um die Korrektheit eines Algorithmus zu zeigen. Sie funktioniert ähnlich wie Induktion und ist eine Aussage oder Zahl.

eine Bedingung

Die Invariante ist vor und nach dem Ausführen des Algorithmus wahr, sie ist unveränderlich, also invariant.

Ablauf Korrektheitsbeweis

1. Zeige Inv(0), d.h. Bedingung ist vor dem ersten Ausführen wahr.
2. Nimm an Inv(i) ist wahr und zeige, dass Inv(i+1) auch wahr sein muss. Fallunterscheidung kann vereinfacht werden „without loss of generality".
3. Schliesse, dass Inv(n), $n \in \mathbb{N}$ wahr sein muss und daher der Algorithmus wahr ist.

Evtl. muss noch gezeigt werden dass der Alg. terminiert.

Bsp. siehe U4.3

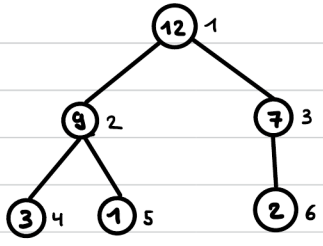
Datenstrukturen

Heap dt. Haufen

max/min Heap

Ein Heap erlaubt es uns ein max/min Element in $O(\log n)$ zu finden.

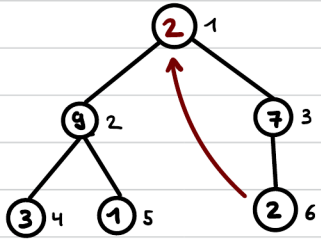
Ein Heap ist oft darstellbar durch einen binär Baum. Dabei werden die Daten in einem Array gespeichert, wobei die Kinder des Knoten mit Index i, an den Stellen $2 \cdot i$ und $2 \cdot i + 1$ zu finden sind.

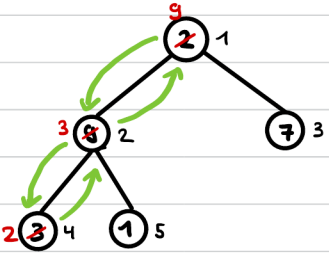


max Heap [12, 9, 7, 3, 1, 2]

Heap (wieder)herstellen

Wenn das erste Element aus dem Heap entfernt wird, rückt das letzte Element an seine Stelle. Nun wird die Heap Bedingung (z. Bsp. $i > 2i$) nicht mehr erfüllt. Um diese wieder herzustellen, wird das erste Element mit seinem grössten Kind verglichen und wenn nötig getauscht, danach wird dass selbe wiederholt bis die Heap-Bedingung erfüllt ist. Dies heisst versickern. Um ein Heap zu erstellen wird dieser Vorgang für jeden Knoten $n/2, \ldots, 1$ durchgeführt. $O(\log n)$



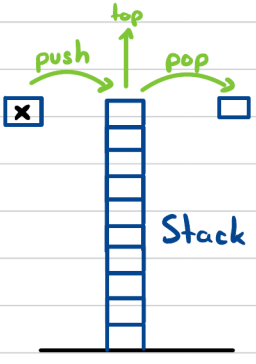


← alle nicht Blätter

Stack LiFo / FiLo

Wird mit einer LinkedList implementiert und besitzt drei Operationen:

- push (x) legt x auf den Stack
- pop () entfernt und liefert das oberste Element
- top () liefert das oberste Element



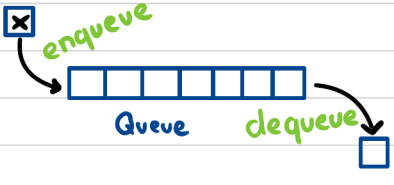
All diese Operationen laufen in $O(1)$

Queue LiLo / FiFo

doubly

Wird ebenfalls mit einer LinkedList implementiert und hat die Operationen:

- enqueue (x) fügt x hinten an
- dequeue liefert das vorderste Element

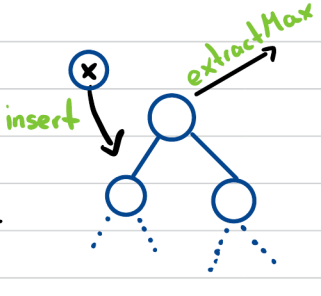


Auch hier laufen die Operationen in $O(1)$ ab.

Priority Queue (für mehr siehe Heap)

Beruht auf einem Max-Heap mit Operationen:

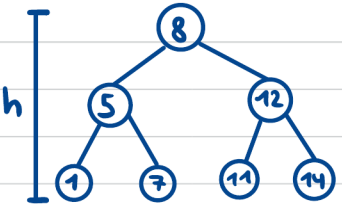
- insert (x) fügt x ein
- extractMax gibt max Element



Alle Operationen laufen in $O(\log n)$.

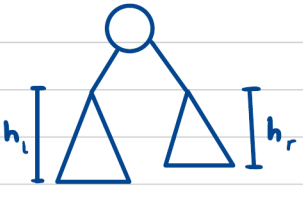
AVL-Tree

Ein AVL-Tree ist eine Art von Suchbaum. In einem Suchbaum laufen alle wichtigen Operationen (insert, remove, search) in $O(h)$. Das Spezielle am AVL-Tree ist, dass er immer Höhe $O(\log n)$ hat.



Ein perfekt balancierter Baum erfüllt die Bedingung, dass für jeden Knoten im Baum die Höhe der Unterbäume gleich ist. Wenn ein Baum diese Bedingung erfüllt, hat er eine Höhe von $\log n$. Da dies jedoch schwer ist, müssen AVL-Trees nur folgende Strukturbedingung erfüllen:

perfekt balanciert



$$|h_l - h_r| \leq 1 \quad \text{für alle Knoten}$$

Operationen:

- search - binäre Suche in $O(\log n)$
- remove - analog zu insert in $O(\log n)$
- insert - in $O(\log n)$ mehr Details auf der nächsten Seite

Insert

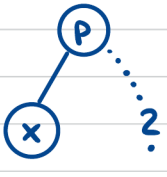
Besteht aus zwei Schritten:
- einfügen
- rebalancieren wenn nötig

1. Einfügen: findet immer an einem Blatt statt und benötigt $O(\log n)$ um das richtige Blatt zu finden.

2. Rebalancieren: Wir schauen uns nur das Einfügen links von einem Knoten p an. Rechts verläuft analog.

Um später zu rebalancieren merken wir uns in jedem Knoten p, $bal(p) = h_l - h_r$.

Nach dem Einfügen gibts nun drei Möglichkeiten:



1. $bal(p) = 1$ Linker Teilbaum höher
2. $bal(p) = 0$ Beide TB gleich hoch
3. $bal(p) = -1$ geht nicht, da wir nur links einfügen betrachten

in diesem Fall hat sich die Höhe verändert und wir müssen die Vorgänger betrachten um evtl. zu rebalancieren.

Wenn wir die Vorgänger betrachten gelten folgende Invarianten:

- $bal(p) \neq 0$
- Höhe von TB p ist gewachsen
- p hat Vorgänger

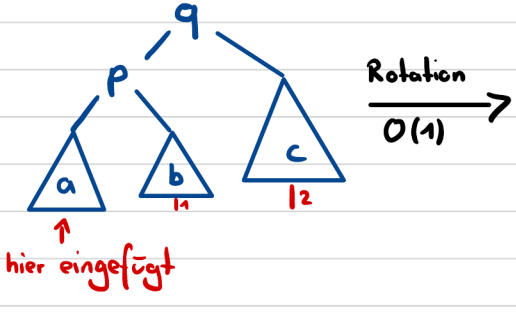
Wir betrachten den Fall, dass p links von Vorgänger q ist, der umgekehrte Fall verläuft analog.

Es gibt wieder drei Fälle:

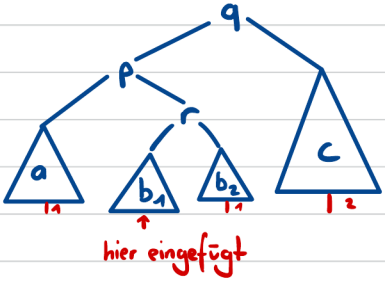
1. $bal(q) = -1 \rightarrow bal(q) = 0$ fertig
2. $bal(q) = 0 \rightarrow bal(q) = 1$ erneut Vorgänger betrachten
3. $bal(q) = 1 \rightarrow bal(q) = 2$ ↯ kein AVL-Tree → rebalancieren

$bal(p) = 1 / -1$

3a) $bal(p) = 1$

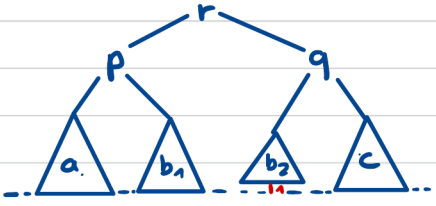


3b) bal(p) = -1



Doppel-rotation

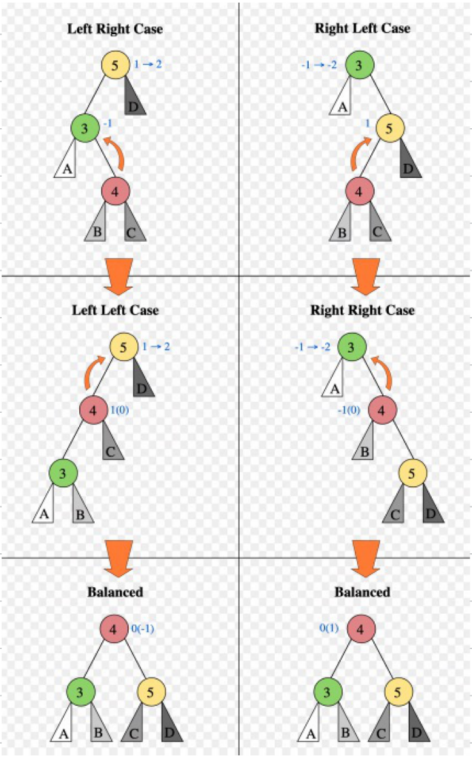
O(1)



bal(p) = 0
bal(r) = 0 ⇒ fertig
bal(q) = -1

Wir sehen, dass Insert O(log n) ist, Remove verläuft ähnlich in AVL-Trees und hat daher auch O(log n).

⇒ Operationen in AVL-Trees sind in O(log n)

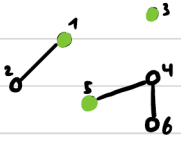


Union - find

Wenn wir mit Graphen arbeiten brauchen wir eine Datenstruktur, die effizient mit ZHK arbeitet. Dies brauchen wir z.Bsp. für den Kruskal Algorithmus.

Idee: Wir speichern für jeden Knoten zu welcher ZHK er gehört, indem wir einen Repräsentanten für die ZHK wählen.

Bsp.



Knoten v	1	2	3	4	5	6
rep[v]	1	1	3	5	5	5

● Repräsentant

Operationen:

make (V):	$O(\|V\|)$	Speicher: $\Theta(n)$
same (u,v):	$O(1)$	
union(u,v):	$\Theta(n)$	worst case
	$O(\log n)$	Ø immer die kleinere ZHK verändern

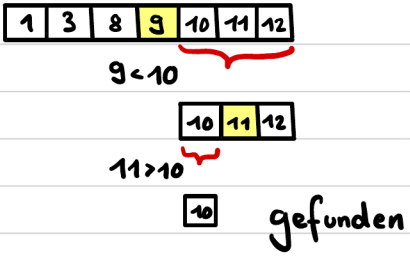
Algorithmen

Suchalgorithmen in sortierten Arrays.

Binary Search $O(\log n)$

Mittleres Element anschauen und anhand dessen rechts oder links weiter suchen.

Bsp. suche nach 10:



Wenn wir einen Verdacht haben in welchem Teil des Arrays sich das gesuchte Element befinden muss, kann Exponentielle Suche oder Interpolations Suche verwendet werden.

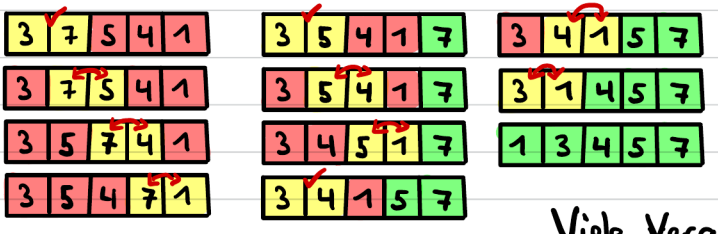
Suchalgorithmen in unsortierten Arrays.

In unsortierten Arrays müssen alle Elemente betrachtet werden. Dies nennt man Lineare Suche und hat eine Laufzeit von $\Omega(n)$.

Sortieralgorithmen bestmöglich $O(n \cdot \log n)$

Bubblesort $\Theta(n^2)$

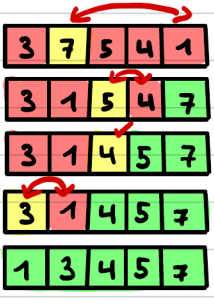
Bsp.



Viele Vergleiche!

SelectionSort $\Theta(n^2)$

Bsp.

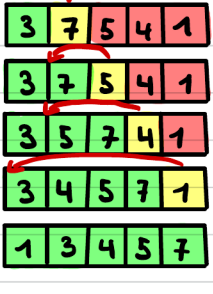


Wir nehmen das jeweils grösste, unsortierte Element und tauschen es an die letzte unsortierte Stelle.

Viele Vergleiche!

Insertion Sort $\Theta(n^2)$

Bsp.

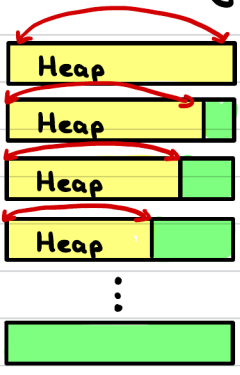


Es wird jeweils das nächste Element genommen und in den bereits sortierten Teil eingefügt.

Viele Verschiebungen!

HeapSort $O(n \cdot \log n)$

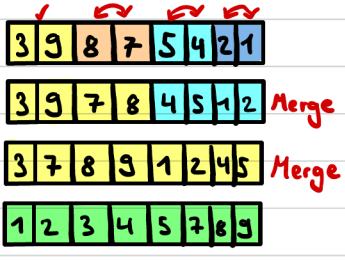
Bsp.



Funktioniert ähnlich wie SelectionSort, aber hier wird durch einen Heap das Array vorsortiert, wodurch das maximale, unsortierte Element immer an erster Stelle ist. (Heap wird jedes Mal neu erstellt.

MergeSort $\Theta(n \cdot \log n)$

Bsp.

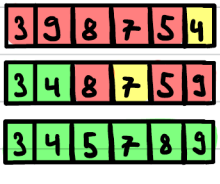


Divide and Conquer Algorithmus. Zuerst wird in Paaren sortiert, danach werden die Teilstücke „zusammengefügt"/gemerged.

QuickSort $O(n^2)$

in Realität am besten! mit ø $O(n \log n)$

Bsp.



Ein zufälliges Pivotelement p wird gewählt. Danach werden die Elemente < p auf der linken und die Elemente > p auf der rechten Seite angeordnet. Auf die beiden Seiten wird der Algorithmus rekursiv angewendet.

Randomisierter Quicksort ist am besten.

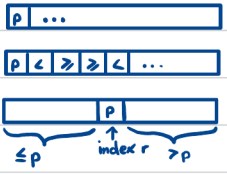
QuickSort funktioniert am besten wenn das gewählte Pivotelement der Median ist. Dies könnte man mit QuickSelect erreichen. Jedoch würde damit nur die Worst-Case Laufzeit auf $O(n \log n)$ gebracht und der Average Case bleibt gleich. Daher lohnt sich die zusätzliche Komplexität nicht.

QuickSelect

Das Ziel ist es in einem gegebenen Array der Grösse n, dass i-kleinste Element zu finden.

Idee: Pivotieren, rekursiv

- $O(1)$ 1) wähle Pivot, z.Bsp erstes
- $O(n)$ 2) zähle Elemente $\leq p \Rightarrow r$ viele
- $O(n)$ 3) teile A
  - 4) $i = r$: gefunden
    $i < r$: suche i-tes links
    $i > r$: suche $(i-r)$-tes rechts



Laufzeit: Worst-Case $\Theta(n^2)$
Average-Case $\Theta(n)$

Was ist ein gutes Pivotelement? - reduziert länge um Faktor $q < 1$.

Median der Mediane

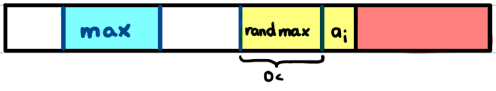
1) a) betrachte Array in 5-er Gruppen
   b) bestimme Median in jeder Gruppe
   c) A' = Array der Gruppen Mediane
   d) bestimme Median von $A' \Rightarrow p$

2)-4) gleich wie vorhin

Laufzeit: $O(n)$
Reduziert QuickSelect zu $O(n)$

Maximum Subarray Sum $\Theta(n)$ DP

Such die maximale, aneinanderliegende Summe in einem Array.



Der beste Algorithmus geht von links nach rechts durchs Array und sucht dabei ein randmax > 0, dann vergleicht er randmax + a_i > max, wenn true speichert er das neue Max, sonst wenn randmax + a_i > 0 geht er weiter zu a_{i+1} und wenn randmax + a_i < 0, beginnt er ein neues randmax bei a_{i+1}.

Dynamisches Programmieren DP

DP ist wie Induktion. Ein DP-Algorithmus besteht aus zwei wesentlich Komponenten:

Wird verwendet wenn es rekursive Strukturen gibt und immer die gleichen Probleme auftreten.

- Bottom-Up berechnung der Rekurenz:
  - ↳ evtl. auch top-down,
  - ↳ Resultate werden in Tabelle gespeichert

  Memoization

- Design der Rekurenz
  - ↳ evtl. Problem abwandeln

Die Berechnung der Fibonacci-Zahlen ist ein gutes Beispiel.

Längste aufsteigende Teilsumme $O(n \log n)$

Bsp. [4, 9, 8, 13, 10, 11, 7, 3, 16]

Länge	1	2	3	4	5	6
Endwert	~~4~~	~~9~~	~~13~~	11	16	
	3	~~8~~	10			
		7				

updates

Ergibt Länge, wenn Vorgänger gemerkt werden, können Elemente ausgelesen werden.

Längste gemeinsame Teilfolge (ähnlich: Min. Editierdistanz)

T I G E R
Z I E G E

Es gibt 4 mögliche Fälle:

1. $\overline{x}$ $LGT(n,m) = LGT(n-1,m)$
2. $\overline{y}$ $LGT(n,m) = LGT(n,m-1)$
3. $\frac{y}{x}$ $LGT(n,m) = LGT(n-1,m-1)$
4. $\frac{x}{x}$ $LGT(n,m) = LGT(n-1,m-1)+1$

Als Induktion: max aus mit base case $LGT(0,\cdot) = 0$ oder $LGT(\cdot,0) = 0$. top-down

bottom-up: berechnung durch füllen von Tabelle.

LGT	_	T	I	G	E	R
_	0	0	0	0	0	0
Z	0	0	0	0	0	0
I	0	0	1	1	1	1
E	0	0	1	1	2	2
G	0	0	1	2	2	2
E	0	0	1	2	3	3

→ Länge

↑ Zeile ist wichtig!

Lösung durch Rückverfolgen wo sich Zahl verändert

Shortest Edit Distance funktioniert äquivalent

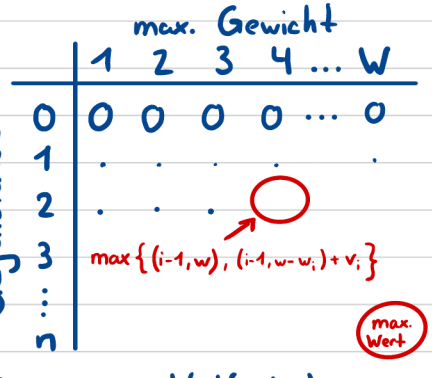
Knapsack

Hier mit dem Beispiel eines Einbrechers, der ein max. Gewicht W tragen kann und eine Auswahl zwischen n Gegenständen hat, wobei ein Gegenstand i den Wert w_i und das Gewicht w_i hat.

Greedy: Die naheliegende Lösung ist den Wert pro Gewicht auszurechnen und dann in absteigender Reihenfolge die Gegenstände einzupacken bis W erreicht ist.
⇒ Naiver Algorithmus, Lösung beliebig viel schlechter

DP: Ein DP Algorithmus ist für dieses Problem optimal. Wir verwenden eine Tabelle maxWert mit Dimensionen 0...W und 0...n. Jeder Eintrag wird aus zwei Vorherigen berechnet. Dies sind die beiden Fälle:

- i ist nicht teil der Lösung → $maxWert(i-1, w)$
- i ist teil der Lösung → $maxWert(i-1, w-w_i) + v_i$



das selbe geht auch mit V anstelle von W und maxValue

Der Base Case ist hier $maxWert(0, w) = 0$. Durch Rückverfolgen kann die optimale Auswahl an Gegenständen berechnet werden. Die Laufzeit ist $\Theta(nW)$, da so viele Tabelleneinträge berechnet werden müssen.

Polynomialzeitapproximation

Unser Algorithmus benötigt aktuell eine Pseudopolynomiale Zeit. Um dies zu verbessern kann ein Approximationsalgorithmus verwendet werden. Dies geschieht jedoch auf Kosten von Genauigkeit. für mehr: Skript 66.

Graphentheorie

Es gibt viele Informatik / Algorithmik Probleme die als Graphen interpretiert werden können (Bsp. Navigation).

Definitionen

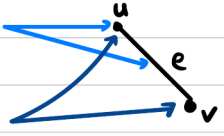
Graph: $G(V, E)$
- Knotenmenge V (vertices)
- Kantenmenge E (edges)

wobei jede Kante ein ungeordnetes Paar Knoten ist.

Knotengrad: Anzahl Nachbaren / anliegende Kanten $\deg(u)$

Inzident: incidental

Adjazent: adjacent



Weg: Folge von benachbarten Knoten (walk).

Pfad: Weg ohne wiederholte Knoten.

Zyklus: Weg mit $v_0 = v_l$, $l \geq 2$ (closed walk).

Kreis: Zyklus ohne wiederholte Knoten, $l \geq 3$ (cycle).

Eulerweg: Alle Kanten genau einmal benutzt.

Hamiltonpfad: Alle Knoten genau einmal besucht.

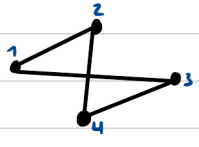
Eulerzyklus: Zyklus mit allen Kanten genau einmal benutzt.

Um einen Graphen auszudrücken gibt es verschiedene Datenstrukturen mit jeweils unterschiedlichen Vor- und Nachteilen

Adjazenzmatrix

$V \times V$ Matrix mit Einträgen
0 wenn zwei Knoten nicht adjazent sind und 1 wenn sie es sind.

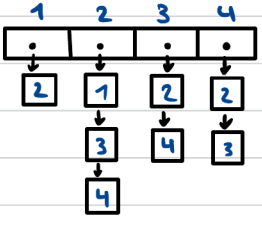
	1	2	3	4
1	0	1	1	0
2	1	0	0	1
3	1	0	0	1
4	0	1	1	0

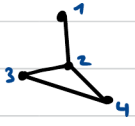


$u,v \in E$: $O(1)$ alle inzidenten Kanten: $O(|V|)$ Eulerweg: $O(|V| \cdot |E|)$

Adjazenzliste

Ein Array von Listen, wobei jeder Knoten eine Liste mit ihm adjazenten Knoten hat.

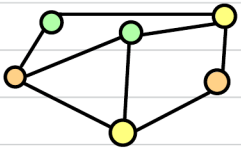




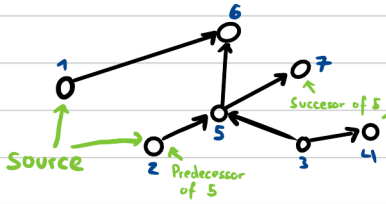
$u,v \in E$: $O(1 + \min \deg(u), \deg(v))$ alle inzidenten Kanten: $O(1 + \deg(u))$ Eulerweg: $O(|V| + |E|)$

Kompleter Graph: jeder Knoten ist direkt mit jedem anderen Knoten verbunden.

Graph Färbung: jeder Knoten wird so gefärbt, dass zwei adjazente Knoten nie die gleiche Farbe haben.



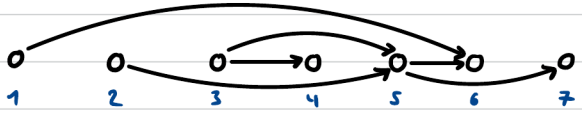
Gerichtete Graphen



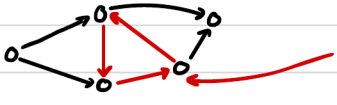
d.h. sind geordnete Paare

In einem gerichteten Graphen haben alle Kanten eine Richtung, es gibt eine Abhängigkeit

Eine topologische Sortierung ist eine Reihenfolge der Knoten, die alle Abhängigkeiten erfüllt.



Sobald ein Graph einen gerichteten Zyklus enthält, gibt es keine topologische Sortierung.



Gerichteter Zyklus

Depth-First-Search DFS

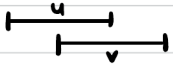
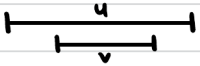
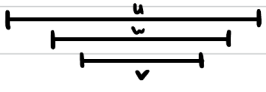
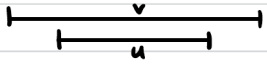
DFS ist ein Algorithmus, der zuerst in der „Tiefe" eines Graphen sucht. Er wird u.a. dazu verwendet eine topologische Sortierung zu erstellen.

Die Laufzeit beträgt $O(|V| + |E|)$

```
visit (u)
    mark u
    for successor v, unmarked
        visit (v)
    add u to topo. sorting
```

```
dfs (G)
    for u₀ ∈ V, unmarked
        visit (u₀)
```

Wir können die Kanten anhand des Intervalls indem sie aufgerufen werden klassifizieren. Kante $(u,v) \in E$

I_n vs. I_v	Klassifizierung
	nicht möglich, aufruf auf v muss zuerst enden
	forward oder DFS-Baum
	forward
	back
	cross
	nicht möglich (v noch nicht markiert als u betrachtet wird)

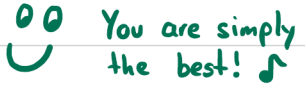
Auf ungerichteten Graphen ist „back" = „forward" Kante und es existieren keine „cross" Kanten.

Breadth-First-Search BFS

BFS oder Breitensuche ist das Gegenstück zu DFS. Zuerst wird in der Breite gesucht und danach erst geht man einen Schritt in die Tiefe des Baumes. So entsteht ein breadth-first tree.

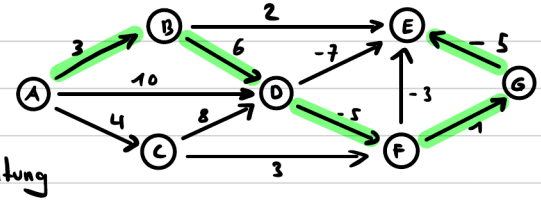
```
bfs (s)
    Queue S ← {s}   ⚠ Knoten können mehrmals in S vorkommen
    while S ≠ ∅
        u ← dequeue S
        if u unmarked
            mark u
            for (u,v) ∈ E, v unmarked
                enqueue S,v
```

Die Laufzeit beträgt dabei $O(|V| + |E|)$



Kürzester Weg in gewichteten Graphen

Wir suchen den kürzesten Weg zwischen zwei Knoten in einem Graph. Dabei hat aber nun jede Kante $e \in E$ eine Gewichtung $c(e)$. Um dieses Problem zu betrachten brauchen wir zuerst ein paar Definitionen.



Distanz

Die Distanz d zwischen zwei Knoten u,v entspricht folgendem:

$$d(u,v) = \min \{ c(w) \mid u \xrightarrow{w} v \}$$

$\Rightarrow$ falls $\overbrace{v_0, \dots, v_i}^{w}$ günstig, dann auch $v_0, \dots, v_{i-1}$ günstig.

In azyklischen Graphen ist die kürzeste Distanz ein DP-Problem mit folgender Rekurrenz: $d(s,v) = \min_{u,v \in E} d(s,u) + c(u,v)$.
Dies kann in $O(|V| + |E|)$ berechnet werden.

Dijkstra

Dijkstra funktioniert mit der Idee, dass die kürzeste Distanz zwischen zwei Punkten s, v über v^x gehen. D.h. wir können zuerst die kürzeste Distanz s, v^x berechnen.

Um die Laufzeit klein zu halten, verwenden wir eine Priority Queue (hier min-Heap). Dadurch erhalten wir eine Laufzeit von $O(|E| + |V| \cdot \log |V|)$.

```
dijkstra(G, v)
    H ← make heap (v)
    S ← ∅
    d[s] ← 0, d[v] ← ∞ for v ∈ V\{s}
    H.decrease_priority (s, 0)
    while S ≠ V
        v^x ← H.extract_min()
        S ← S ∪ {v^x}
        for (v^x, v) ∈ E, v ∉ S:
            d[v] ← min(d[v], d[v^x] + c(v^x, v))
            H.decrease_priority (v, d[v])
    return d
```

Der Algorithmus bekommt jedoch Probleme sobald Kanten mit negativer Gewichtung auftreten.

Hey girl are you Dijkstra?
- because I want you to show me the shortest path to your heart ♥! :-)

Bellman - Ford

Bellman-Ford funktioniert im Gegensatz zu Dijkstra nicht auf ungerichteten Graphen, aber dafür mit negativen Kanten.

In jedem Schritt werden die Knoten „relaxed".

Dies ergibt eine Laufzeit von $O(|V| \cdot |E|)$.

Fall negative Zyklen auftreten, wird ein Error geworfen.

```
bellmanFord(G, s)      // s is source
  for v in V
      distance[v] = ∞
      predecessor[v] = null
  distance[s] = 0
  repeat |V|-1 times
      for (u,v) in E with weight w
          if distance[u] + w < distance[v]
              distance[v] = distance[u] + w
              predecessor[v] = u
  for (u,v) in E with weight w
      if distance[u] + w < distance[v]
          error "Graph has negative cycles"
  return distance, predecessor
```

Vergleich

	Laufzeit	gerichtet/ ungerichtet	negative Kanten	negative Zyklen
DP für DAG	$O(\|V\|+\|E\|)$	gerichtet	ja	nein
Dijkstra	$O((\|E\|+\|V\|)\cdot \log\|V\|)$	beide	nein	nein
Bellman Ford	$O(\|E\| \cdot \|V\|)$	gerichtet	ja	jein (kann sie erkennen)

Minimals Spannbaum MST

Ein MST eines Graphen erfüllt die folgenden Eigenschaften:

- enthält alle $V \in G$
- enthält keine Zyklen

Aus einem MST kann durch entfernen von Kanten ein Spannwald gebildet werden. Teile eines solchen Waldes welche nicht verbunden sind, nennt man Zusammenhangs Komponenten ZHK.

Boruvka

```
boruvka (G)
  F = ∅        //sichere Kanten
  while  F ≠ V
      (S1, S2, ..., Sk) = ZHK von F
      (e1, e2, ..., ek) = minimale Kanten an (S1, S2, ... Sk)
      F = F ∪ {e1, e2, ..., ek}
```

Laufzeit pro Heration: $O(|V| + |E|)$

Anzahl Herationen: $\leq \log_2 |V|$

Totale Laufzeit: $O(\log|V| \cdot (|E| + |V|))$

Prim

Prim funktioniert ähnlich zu Dijkstra bis auf den markierten Teil. Das Prinzip ist den MST ausgehend vom Startknoten s aufzubauen. Der Min-Heap ist wichtig um schnell die minimale Kante an S zu finden.

Die Laufzeit ist gleich wie bei Dijkstra und Boruvka: $O(\log|V| \cdot (|V|+|E|))$

```
prim (G, s)
    h = min-heap (V, ∞)
    S = ∅
    d[s] = 0,  d[v] = ∞  for v ∈ V \ {s}
    h.decreaseKey (s, 0)
    while h ≠ ∅
        v = h.extractMin
        S = S ∪ {v}
        for (v, u) ∈ E, u ∉ S
            d[u] = min (d[u], w(v, u))
            h.decreaseKey (u, d[u])
```

Kruskal

Kruskal basiert darauf die Kanten nach Gewicht zu sortieren und dann der Reihe nach sichere Kanten zum MST hinzuzufügen.

```
kruskal (G)
    F = ∅
    for (v,u) ∈ E, aufsteigend sortiert
        if (v,u) in verschiedenen ZHK
            F = F ∪ {v,u}
```

Zusammen mit der Union-find Datenstruktur erhalten wir eine Laufzeit von $O(|E| \cdot \log|E| + |E| \cdot \log|V|)$.

Wir wollen nun Graphen-Algorithmen die nicht den kürzesten Weg von einem Knoten, sondern von allen berechnen. Wir können dies tun, indem wir die bereits bekannten Algorithmen n-mal anwenden, oder mit neuen Algorithmen.

Graph	Algorithmus	Laufzeit
$G=(V,E)$	n x Breitensuche	$O(mn+n^2)$
$G=(V,E,c)$ $c: E \to \mathbb{R}^+$	n x Dijkstra	$O(mn+n^2 \log n)$
$G=(V,E,c)$ $c: E \to \mathbb{R}$	n x Bellman-Ford	$O(mn^2)$
	Floyd-Warshall	$O(n^3)$
	Johnson	$O(mn+n^2 \log n)$

Floyd-Warshall

Idee: nummerierte Knoten $1 \dots n$

$d_{u,v}^{\,i}$ = kosten des günstigsten Weges $u \to v$ mit Zwischenknoten $\leq i$.

$= \min \{ d_{uv}^{\,i-1}, d_{ui}^{\,i-1} + d_{iv}^{\,i-1} \}$

Dies resultiert in einer 3D-DP-Tabelle:

Initialisierung:

$d_{uu}^{\,0} = 0$

$d_{uv}^{\,0} = 1 \Rightarrow (u,v) \in E$

$d_{uv}^{\,0} = \infty$

Laufzeit: $O(n^3)$ Speicherplatz: $O(n^2)$ mit inplace

Bemerkung: Es gibt zwei verschiedene Variationen

- $d_{uv} = d_{uv} \vee (d_{ui} \wedge d_{iv})$ ⇒ transitive Hülle
- $d_{uv} = d_{uv} + (d_{ui} \cdot d_{iv})$ ⇒ Matrixmultiplikation

Bedeutung der Matrixmultiplikation für Graphen:

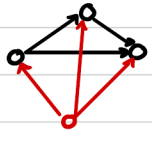
A = Adjazenzmatrix von G

$(i,j) \in A^k$ = anzahl Wege von i nach j mit Länge k!

Johnson

Idee: wir machen alle Kantengewichte positiv und führen dann $n \times$ Dijkstra aus.

Um die Gewichte positive zu machen führen wir einen neuen Knoten mit Distanz 0 zu allen Knoten ein.



$$c^*(u,v) = c(u,v) + h(u) - h(v)$$

↑ neue Kantengewichte

Die Höhe $h(u)$ ist der kürzeste Weg vom neuen Knoten nach u.

⇒ kürzeste Wege bleiben am kürzesten
⇒ Zyklenkosten bleiben gleich

Analyse:

Neuer Knoten:	$O(n)$
h-Werte, Bellman-Ford:	$O(mn)$
n-Mal Dijkstra:	$O(mn + n^2 \log n)$
Laufzeit:	$O(mn + n^2 \log n)$

⇒ besser, aber nur bei dünn besetzten Graphen