

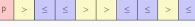
3.6. Quicksort

Pivotieren

- Wähle ein beliebiges Element als Pivot



- Teile A in zwei Teile



- Quicksort: Rekursion auf L und R



Wahl des Pivot

Maximum/Minimum (worst case) in $\mathcal{O}(n^2)$.

Best case (Pivot in der Mitte) $\omega(n)$.

Partition(A,l,r,p)

Input: Array A, welches den Pivot p in $A[l, \dots, r]$ mindestens einmal enthält.

Output: Array A partitioniert in $A[l, \dots, r]$ um p. Rückgabe der Position von p, während $l \leq r$ do

```

while A[l] < p do
    l ← l + 1
while A[r] > p do
    r ← r - 1
swap(A[l], A[r])
if A[l] = A[r] then
    l ← l + 1
return l-1

```

Quicksort(A,l,r)

Input: Array A der Länge n. $1 \leq l \leq r \leq n$.

Output: Array A, sortiert in $A[l, \dots, r]$.

if $l < r$ then

```

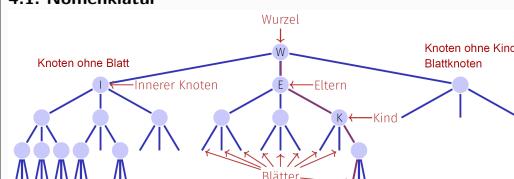
    Wähle Pivot p ∈ A[l, ..., r]
    k ← Partition(A, l, r, p)
    Quicksort(A, l, k - 1)
    Quicksort(A, k + 1, r)

```

Im Mittel benötigt randomisiertes Quicksort $\mathcal{O}(n \cdot \log(n))$ Vergleiche.

4. Natürliche Suchbäume

4.1. Nomenklatur



4.2. Binäre Suchbäume

Binärer Baum (nur zwei Nachfolgerknoten) mit Eigenschaften:

- Jeder Knoten v speichert einen Schlüssel
- Schlüssel im linken Teilbaum v.left kleiner als v.key
- Schlüssel im rechten Teilbaum v.right größer als v.key

4.2.1. Höhe eines Baumes

$$h(r) = \begin{cases} 0 & \text{falls } r = \text{null} \\ 1 + \max\{h(r \cdot \text{left}), h(r \cdot \text{right})\} & \text{sonst.} \end{cases}$$

Die Laufzeit der Suche ist somit im schlechtesten Fall $\mathcal{O}(h(T))$.

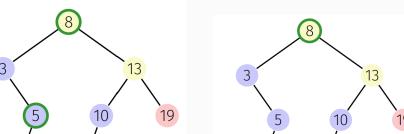
4.2.2. Operationen

Knoten entfernen

Mögliche Situationen: Knoten hat keine Kinder, Knoten hat ein Kind oder Knoten v hat zwei Kinder. Im letzten Fall: Der kleinste Schlüssel im rechten Teilbaum v.right ist der symmetrische Nachfolger von v → ersetze v durch seinen symmetrischen Nachfolger.

Auch möglich: ersetze v durch seinen symmetrischen Vorgänger
Implementation: der Teufel steckt im Detail!

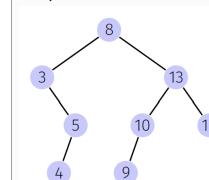
Grafik zu dem symmetrischen Vorgänger (links) oder Nachfolger (rechts)



4.3. Traversierungsarten

- Hauptreihenfolge (preorder):
 - v, dann $T_{left}(v)$, dann $T_{right}(v)$.
- Nebenreihenfolge (postorder):
 - $T_{left}(v)$, dann $T_{right}(v)$, dann v.
- Symmetrische Reihenfolge (inorder):
 - $T_{left}(v)$, dann v, dann $T_{right}(v)$.

Beispiel



- Hauptreihenfolge (preorder):
 - 8, 3, 5, 4, 10, 9, 19

- Nebenreihenfolge (postorder):
 - 4, 5, 3, 9, 10, 19, 13, 8

- Symmetrische Reihenfolge (inorder):
 - 3, 4, 5, 8, 9, 10, 13, 19

4.4. C++ - Implementation

4.4.1. Tree

```

class BST {
    Node* root;
public:
    BST();
    bool contains(int key) const;
    void insert(int key);
    void remove(int key);
    void print_preorder(std::ostream& out) const;
    void clear();
    ~BST();
};

// Constructor, creates an empty binary search tree
BST::BST(): root(nullptr) {}

// Returns true iff the BST contains the given key.
bool BST::contains(int key) const {
    return root == nullptr ? false : root->contains(key);
}

// Returns true iff the given key was inserted into the BST.
bool BST::insert(int key) {
    if (root == nullptr) {
        root = new Node(key);
    }
}

// POST: Returns true iff the tree contains search_key.
bool Node::contains(int search_key) const {
    const Node* curr = this;

    while (curr != nullptr) {
        if (curr->key == search_key) {
            return true;
        } else if (search_key < curr->key) {
            curr = curr->left;
        } else {
            assert(search_key > curr->key);
            curr = curr->right;
        }
    }
    return false;
}

// POST: Inserts new_key into the tree, if not already
// present. Returns true iff new_key was inserted.
// Maintains the binary search-tree invariant.
bool Node::insert(int new_key) {
    Node* curr = this;

    while (curr->key != new_key) {
        if (new_key < curr->key) {

```

```

            curr = curr->left;
        } else {
            curr = curr->right;
        }
    }
    curr = new Node(new_key);
}

// Returns true iff the given key was removed from the BST.
bool BST::remove(int key) {
    if (root == nullptr) {
        return false;
    } else if (root->key == key) {
        root = Node::symmetric_successor(root);
        return true;
    } else {
        curr = root->remove(key);
    }
    return false;
}

// Prints the preorder-traversal of the BST to out.
void BST::print_preorder(std::ostream& out) const {
    if (root != nullptr) {
        root->print_preorder(out);
        out << '\n';
    }
}

// Clears the BST, i.e. resets it to empty.
void BST::clear() {
    delete root;
    root = nullptr;
}

// Deconstructor
BST::~BST() {
    clear();
}

4.4.2. Node

struct Node {
    int key; // Key, i.e. value stored in this node
    Node* left; // Left child, i.e. root of left subtree
    Node* right; // Right child, i.e. root of right subtree

    Node(int key, Node* left, Node* right);
};

// Node constructor
Node::Node(int key, Node* left, Node* right):
    key(key), left(left), right(right) {}

// POST: Returns true iff the tree contains search_key.
bool Node::contains(int search_key) const {
    const Node* curr = this;

    while (curr != nullptr) {
        if (curr->key == search_key) {
            return true;
        } else if (search_key < curr->key) {
            curr = curr->left;
        } else {
            curr = curr->right;
        }
    }
    return false;
}

// Analogously, for curr's right child
remove_node = curr->right;
curr->right = symmetric_successor(curr->right);
} else if (remove_key < curr->key) {
// Descend left
curr = curr->left;
} else {
// Descend right
assert(remove_key > curr->key);
curr = curr->right;
}

if (remove_node != nullptr) {
// To prevent memory leaks, the removed node is freed
remove_node->left = nullptr;
remove_node->right = nullptr;
delete remove_node;
return true;
} else {
return false;
}

// POST: Prints the tree's keys in preorder traversal to out.
void Node::print_preorder(std::ostream& out) const {
    out << key << '\n';
    if (left != nullptr) left->print_preorder(out);
    if (right != nullptr) right->print_preorder(out);
}

// PRE: root != nullptr
// POST: Returns the symmetric successor of the given root
// node. If the symmetric successor is not a direct child of
// the given root, then the symmetric successor is also
// removed from its context, i.e. from the subtree of root
// where the symmetric successor was found.
Node* Node::symmetric_successor(Node* root) {

```

```

assert(root != nullptr);

// If there's at most one child node, it must be the
// symmetric successor, and we're done right away.
if (root->left == nullptr) return root->right;
if (root->right == nullptr) return root->left;

// Otherwise, the symmetric successor must be the left-most
// element of root's right subtree. We use curr to descend
// down the tree, and parent will have parent->left == curr
// (if parent != nullptr). parent needed for removing the
// eventually found symmetric successor from its context.
Node* curr = root->right;
Node* parent = nullptr;

// Descend leftwards. After the loop, curr is the symmetric
// successor.
while (curr->left != nullptr) {
    parent = curr;
    curr = curr->left;
}

// Remove the symmetric successor from its context, if
// necessary.
if (parent != nullptr) {
    parent->left = curr->right;
    curr->right = root->right;
}

curr->left = root->left;
return curr;
}

// Deconstructor. All transitive children are recursively
// deleted.
Node::~Node() {
    delete left;
    delete right;
}

```

5. AVL Bäume

Ziel: Verhinderung der Degenerierung → garantiere, dass ein Baum mit n Knoten stets eine Höhe von $\mathcal{O}(\log(n))$.

5.1. AVL Bedingung

5.1.1. Balance eines Knotens

Die Balance eines Knotens v ist definiert als die Höhendifferenz seiner beiden Teilbäume $T_l(v)$ und $T_r(v)$:

$$\text{bal}(v) := h(T_r(v)) - h(T_l(v))$$

- **Augmentieren:** v .size Feld mit Balance

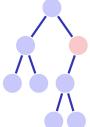
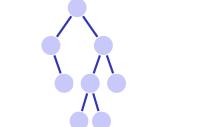
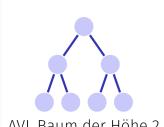
- **Baumhöhe:** Ein AVL Baum ist asymptotisch nicht mehr als 44% höher als ein perfekt balancierter Baum ($\lceil \log_2 n + 1 \rceil$).

5.1.2. AVL Bedingung

AVL Bedingung: für jeden Knoten v eines Baumes gilt:

$$\text{bal}(v) \in \{-1, 0, 1\}$$

Beispiele



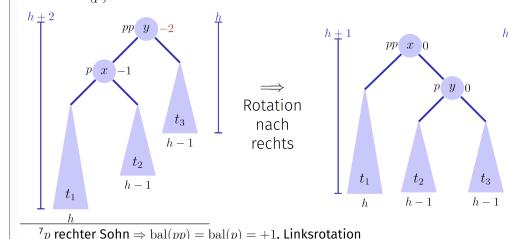
5.2. Einfügen

- Zuerst einfügen wie bei Suchbaum.
- Prüfe die Balance-Bedingung für alle Knoten aufsteigend von n zur Wurzel.
 $\text{upin}(p)$: Aufsteigend von p die Balance (Augmentation) anpassen, wobei gilt $\text{bal}(p) \in \{-1, 0, +1\}$.
- Problematischer Fall (rebalancieren!): p ist linker Sohn von pp , wobei $\text{bal}(pp)$ bereits vor dem Einfügen -1 ist (danach $+2$).

5.3. Rebalancieren: Rotationen

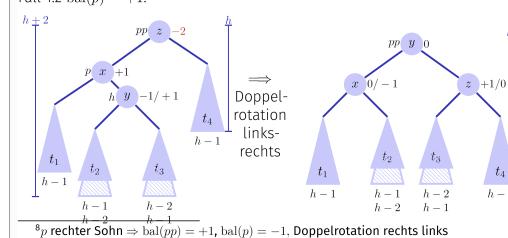
Fall 1: Rotation nach rechts

Fall 1.1 $\text{bal}(p) = -1$.⁷



Fall 2: Doppelrotation nach links-rechts

Fall 1.2 $\text{bal}(p) = +1$.⁸



5.4. Analyse

AVL-Bäume haben asymptotische Laufzeit von $\mathcal{O}(\log(n))$ (schlechtester Fall) für das Suchen, Einfügen und Löschen von Schlüsseln. Einfügen und Löschen ist verhältnismässig aufwändig und für kleine Probleme relativ langsam.

6. Heaps

6.1. [Max-]Heap

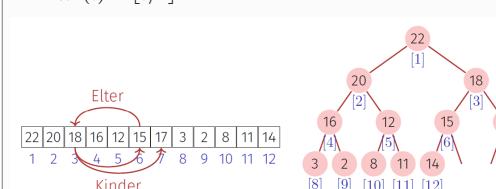
Datenstruktur optimiert zum schnellen Extrahieren von Minimum oder Maximum und Sortieren.

Binärer Baum mit folgenden Eigenschaften

1. Vollständig, bis auf die letzte Ebene
2. Lücken des Baumes in der letzten Ebene höchstens rechts.
3. **Heap-Bedingung:**
Max-(Min-)Heap: Schlüssel eines Kindes kleiner (grösser) als der des Elternknotens

Baum → Array:

1. Kinder (i) = $\{2i, 2i + 1\}$
2. Elter (i) = $\lfloor i/2 \rfloor$



Höhe eines Heaps: $H(n) = \lceil \log_2(n + 1) \rceil$

6.2. Heap bauen

- Jedes Blatt eines Heaps ist für sich schon ein korrekter Heap. → Induktion von unten!
- Aufrufe an Versickern: $n/2$. Also Anzahl Vergleiche und Bewegungen $v(n) \in \mathcal{O}(n \cdot \log(n))$.
- Versickerpfade sind aber im Mittel viel kürzer: $\mathcal{O}(n)$

6.3. Einfügen

- Füge neues Element an erste freie Stelle ein.
- Stelle Heap Eigenschaft wieder her: **Sukzessives Aufsteigen**
- Anzahl Operationen im worst case: $\mathcal{O}(\log(n))$

Aufsteigen(A,m)

Input: Array A mit mindestens m Elementen und Max-Heap-Struktur auf $A[1, \dots, m]$

Output: Array A mit Max-Heap-Struktur auf $A[1, \dots, m]$.

```

v ← A[m] // Wert
c ← m // derzeitiger Knoten (child)
p ← ⌊c/2⌋ // Elternknoten (parent)
while c > 1 and v > A[p] do
    A[c] ← A[p] // Wert Elternknoten → derzeitiger Knoten
    c ← p // Elternknoten → derzeitiger Knoten
    p ← ⌊c/2⌋
A[c] ← v // Wert → Wurzel des (Teil-)Baumes

```

6.4. Maximum entfernen

- Ersetze das Maximum durch das unterste rechte Element.
- Stelle Heap Eigenschaft wieder her: **Sukzessives Absinken** (in Richtung des grösseren Kindes / "Max. aufsteigen lassen")
- Anzahl Operationen im worst case: $\mathcal{O}(\log(n))$

Versickern(A,i,m)

Input: Array A mit Heapstruktur für die Kinder von i . Letztes Element m .

Output: Array A mit Heapstruktur für i mit letztem Element m .

```

while 2i ≤ m do
    j ← 2i; // j linkes Kind
    if j < m and A[j] < A[j + 1] then
        j ← j + 1; // j rechtes Kind mit grösserem Schlüssel
    if A[i] < A[j] then
        swap(A[i], A[j])
        i ← j; // weiter versickern
    else
        i ← m; // versickern beendet

```

6.5. Heap sortieren

Worst case: $\mathcal{O}(n \cdot \log(n))$

HeapSort(A,n)

```

Input: Array A der Länge n.
Output: A sortiert.
// Heap aufbauen
for i ← n/2 downto 1 do
    Versickere(A,i,n)
// Nun ist A ein Heap
for i ← n downto 2 do
    Vertausche(A[1],A[i])
    Versickere(A,1,i-1)
// Nun ist A sortiert.

```

7. Hashing

7.1. Motivation und Idee

Key/Value-Paare effizient abspeichern und finden z.B. für Implementation eines Dicts oder einer Datenbank.

```

Wörterbuch → fruits = {
    "banana": 2.95, "kiwi": 0.70,
    "pear": 4.20, "apple": 3.95
}

```

Idee

Direkter Zugriff (Array)

Probleme

1. Schlüssel müssen nichtnegative ganze Zahlen sein
2. Grosser Schlüsselbereich → grosses Array

7.2. Pre-Hashing: Lösung des ersten Problems

Prehashing: Bilde Schlüssel ab auf positive Ganzahlen mit einer Funktion $ph : \mathcal{K} \rightarrow \mathbb{N}$

Pre-Hashing: Beispiel String

Zuordnung Name $s = s_1 s_2 \dots s_{l_s}$ zu Schlüssel

$$ph(s) = (\sum_{i=0}^{l_s-1} s_i s_{l_s-i} \cdot b^i) \bmod 2^w$$

b , so dass verschiedene Namen möglichst verschiedene Schlüssel erhalten.
 w Wortgröße des Systems (z.B. 32 oder 64).

```

unsigned prehash(std::string s) {
    unsigned b = B;
    unsigned h = 0;

    for (unsigned i = 0; i < s.size(); ++i){
        h = h * b + s[i];
    }
    return h;
}

```

7.3. Hashing: Lösung des zweiten Problems

Reduziere das Schlüsseluniversum: Abbildung (Hash-Funktion) $h : \mathcal{K} \rightarrow \{0, \dots, m-1\}$ ($m \approx n$ = Anzahl Einträge in der Tabelle)

7.3.1. Nomenklatur

Hashfunktion h : Abbildung aus der Menge der Schlüssel \mathcal{K} auf die Indexmenge $\{0, 1, \dots, m-1\}$ eines Arrays (**Hashtabelle**)

Meist $|\mathcal{K}| \gg m$, Es gibt also $k_1, k_2 \in \mathcal{K}$ mit $h(k_1) = h(k_2)$ (**Kollision**). Eine Hashfunktion sollte die Menge der Schlüssel möglichst gleichmässig auf die Positionen der Hashtabelle verteilen.

7.3.2. Gebräuchliche Hashfunktion: Divisionsmethode

$$h(k) = k \bmod m$$

Ideal: m Primzahl, nicht zu nahe bei Potenzen von 2 oder 10
Aber oft: $m = 2^k - 1$ ($k \in \mathbb{N}$)

7.4. Konzept 1: Hashing mit Verkettung

Direkte Verkettung der Überläufer.

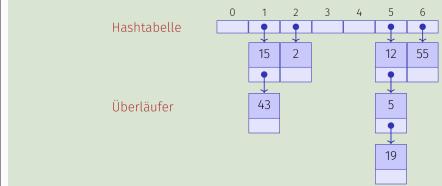
→ Resultiert im worst case in $\Theta(n^2)$ pro Operation

Beispiel

$$m = 7, \mathcal{K} = \{0, \dots, 500\}, h(k) = k \bmod m.$$

Schlüssel 12, 55, 5, 15, 2, 19, 43

Direkte Verkettung der Überläufer



Einfaches gleichmässiges Hashing

Starke Annahmen: Jeder beliebige Schlüssel wird mit gleicher Wahrscheinlichkeit (**Uniformität**) und unabhängig von den anderen Schlüsseln (**Unabhängigkeit**) auf einen der m verfügbaren Slots abgebildet.

Unter dieser Annahme ergibt sich die **erwartete Länge**:

$$\mathbb{E}(\text{Länge Kette } j) = \frac{n}{m} = \alpha, \alpha \text{ heisst der Belegungsfaktor oder Füllgrad.}$$

Daraus ergibt sich (bei einfacherm gleichmässigem Hashing) eine **erwartete Laufzeit (amortisiert)** von $\mathcal{O}(1)$ für Suchen, Einfügen, Lösen.

Vor- und Nachteile der Verkettung

- Belegungsfaktoren $\alpha > 1$ möglich; Entfernen von Schlüsseln einfach
- Speicherverbrauch der Verkettung

7.5. Konzept 2: Hashing mit offener Addressierung

- Speichere die Überläufer direkt in der Hashtabelle mit einer **Sondierungsfunktion** $s(k, j)$
- Tabellenposition des Schlüssels entlang der **Sondierungsfolge** $S(k)$

Technisches Detail zu **delete(k)**: Suche k in der Tabelle gemäß $S(k)$. Ersetze k durch den speziellen Schlüssel **removed**.

7.5.1. Lineares Sondieren

$$s(k, j) = h(k) + j \Rightarrow$$

$$S(k) = (h(k), h(k) + 1, \dots, h(k) + m - 1) \bmod m$$

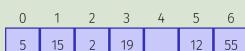
Problem → **Primäre Häufung**:

Ähnliche Hashadressen haben ähnliche Sondierungsfolgen → lange zusammenhängende belegte Bereiche.

Beispiel

$$m = 7, \mathcal{K} = \{0, \dots, 500\}, h(k) = k \bmod m.$$

Schlüssel 12, 55, 5, 15, 2, 19



7.5.2. Quadratisches Sondieren

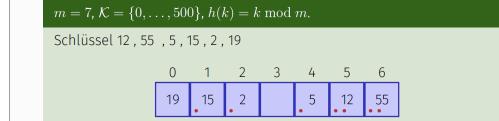
$$s(k, j) = h(k) + \lceil j/2 \rceil^2 (-1)^{j+1}$$

$$S(k) = (h(k), h(k) + 1, h(k) - 1, h(k) + 4, h(k) - 4, \dots) \bmod m$$

Problem → **Sekundäre Häufung**:

Synonyme k und k' (mit $h(k) = h(k')$) durchlaufen dieselbe Sondierungsfolge.

Beispiel



7.5.3. Double Hashing

Verwendung von zwei Hashfunktionen $h(k)$ und $h'(k)$ → Vermeidung primärer und sekundärer Häufungen.

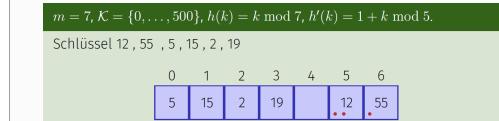
$$s(k, j) = h(k) + j \cdot h'(k)$$

$$S(k) = (h(k), h(k) + h'(k), h(k) + 2h'(k)b, \dots, h(k) + (m-1)h'(k)) \bmod m$$

Gleichmässiges Hashing

Starke Annahme: Die Sondierungssequenz $S(k)$ eines Schlüssels k ist mit gleicher Wahrscheinlichkeit eine der $m!$ vielen Permutationssequenzen von $\{0, 1, \dots, m-1\}$. → Füllgrad $\alpha = \frac{n}{m} < 1$, so hat die nächste Operation erwartete Laufzeitkosten von $\leq \frac{1}{1-\alpha}$

Beispiel



7.5. Konzept 2: Hashing mit offener Addressierung

- Speichere die Überläufer direkt in der Hashtabelle mit einer **Sondierungsfunktion** $s(k, j)$
- Tabellenposition des Schlüssels entlang der **Sondierungsfolge** $S(k)$

Technisches Detail zu **delete(k)**: Suche k in der Tabelle gemäß $S(k)$. Ersetze k durch den speziellen Schlüssel **removed**.

7.5.1. Lineares Sondieren

$$s(k, j) = h(k) + j \Rightarrow$$

$$S(k) = (h(k), h(k) + 1, \dots, h(k) + m - 1) \bmod m$$

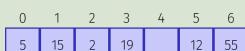
Problem → **Primäre Häufung**:

Ähnliche Hashadressen haben ähnliche Sondierungsfolgen → lange zusammenhängende belegte Bereiche.

Beispiel

$$m = 7, \mathcal{K} = \{0, \dots, 500\}, h(k) = k \bmod m.$$

Schlüssel 12, 55, 5, 15, 2, 19

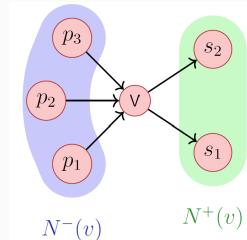


8. Graphen

8.1. Terminologie

Ein Graph $G = (V, E)$ besteht aus der Menge von Knoten $V = \{v_1, \dots, v_n\}$ und der Menge von Kanten E .

- Gerichteter Graph:** $E \subseteq V \times V = \{(u, v) : u \in V, v \in V\}$
- $w \in V$ heisst **adjazent** zu $v \in V$, falls $(v, w) \in E$
 - Vorgänger eines Knotens v : $N^-(v) := \{u \in V | (u, v) \in E\}$
 - Nachfolger eines Knotens v : $N^+(v) := \{u \in V | (v, u) \in E\}$
 - Eingangsgrad: $\deg^-(v) := |N^-(v)|$
 - Ausgangsgrad: $\deg^+(v) := |N^+(v)|$



Ungerichteter Graph: $E \subseteq \{(u, v) : u \in V, v \in V, u \neq v\}$

- $w \in V$ heisst **adjazent** zu $v \in V$, falls $\{v, w\} \in E$
- Nachbarschaft: $N(v) := \{w \in V | \{v, w\} \in E\}$
- Grad: $\deg(v) := |N(v)|$ (Schleifen zählen 2)

Vollständiger Graph: Ungerichteter Graph mit $E = \{(u, v) : u \in V, v \in V, u \neq v\}$

Biparter Graph: Graph, bei dem V so in disjunkte U und W aufgeteilt werden kann, dass alle $e \in E$ einen Knoten in U und einen in W haben

Wege:

- **Weg / Path:** Sequenz von Knoten $p = \langle v_1, v_2, \dots, v_k \rangle$ so dass für jedes $i \in \{1, \dots, k\}$ eine Kante von v_i nach v_{i+1} existiert
- **Pfad / einfacher Pfad / simple path:** Weg der keinen Knoten mehrfach verwendet
- **Länge des Weges:** Anzahl enthaltene Kanten k
- **Gewicht des Weges** (in gewichteten Graphen): $\sum_{i=1}^k c((v_i, v_{i+1}))$ (bzw. $\sum_{i=1}^k c(\{v_i, v_{i+1}\})$)

Zusammenhang:

- Ungerichteter Graph heisst **zusammenhängend**, wenn für jedes Paar $v, w \in V$ ein verbindender Weg existiert.
- Gerichteter Graph heisst **stark zusammenhängend**, wenn für jedes Paar $v, w \in V$ ein verbindender Weg existiert.
- Gerichteter Graph heisst **schwach zusammenhängend**, wenn der entsprechende ungerichtete Graph zusammenhängend ist.

Zyklen:

- **Zyklus:** Weg (und nicht einfacher Pfad!) $\langle v_1, \dots, v_{k+1} \rangle$ mit $v_1 = v_{k+1}$
- **Einfacher Zyklus:** Zyklus, aber Knoten kommen nicht mehrfach vor (außer s und t)
- **Kreis:** Zyklus mit paarweise verschiedenen v_1, \dots, v_k , welcher keine Kante mehrfach verwendet
- **Kreisfrei (azyklisch):** Graph ohne jegliche Kreise.

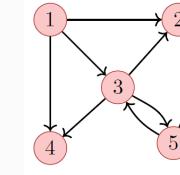
Beobachtungen

- Allgemein: $0 \leq |E| \in \mathcal{O}(|V|^2)$
- Zusammenhängender Graph: $|E| \in \Omega(|V|)$
- Vollständiger Graph: $|E| = \frac{|V| \cdot (|V|-1)}{2}$ (ungerichtet)
- Maximal $|E| = |V|^2$ (gerichtet)
- Maximal $|E| = \frac{|V| \cdot (|V|+1)}{2}$ (ungerichtet)

8.2. Repräsentation von Graphen

8.2.1. Adjazenzmatrix

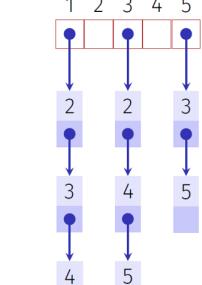
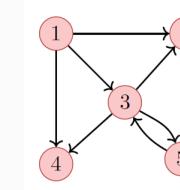
Graph $G = (V, E)$ mit Knotenmenge v_1, \dots, v_n gespeichert als Adjazenzmatrix $A_G = (a_{ij})_{1 \leq i, j \leq n}$ mit Einträgen aus $\{0, 1\}$. $a_{ij} = 1$ genau dann wenn Kante von v_i nach v_j . Speicherbedarf $\Theta(|V|^2)$. A_G ist symmetrisch, wenn G ungerichtet.



$$\begin{pmatrix} 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 \end{pmatrix}$$

8.2.2. Adjazenzliste

Viele Graphen $G = (V, E)$ mit Knotenmenge v_1, \dots, v_n haben deutlich weniger als n^2 Kanten. Repräsentation mit Adjazenzliste: Array $A[1], \dots, A[n]$, $A[i]$ enthält vertaktete Liste aller Knoten in $N^+(v_i)$. Speicherbedarf $\Theta(|V| + |E|)$.

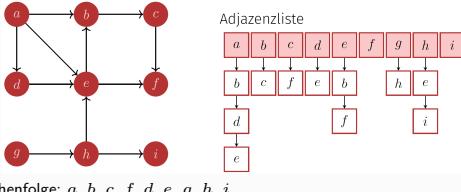


8.2.3. Laufzeiten einfacher Operationen

Operation	Matrix	Liste
Nachbarn/Nachfolger von $v \in V$ finden	$\Theta(n)$	$\Theta(\deg^+ v)$
$v \in V$ ohne Nachbar/Nachfolger finden	$\Theta(n^2)$	$\Theta(n)$
$(u, v) \in E$?	$\Theta(1)$	$\Theta(\deg^+ v)$
Kante einfügen	$\Theta(1)$	$\Theta(1)$
Kante löschen	$\Theta(1)$	$\Theta(\deg^+ v)$

8.3. Graphen Traversieren: Tiefensuche

Verfolge zuerst Pfad in die Tiefe, bis nichts mehr besucht werden kann.



Tiefensuche ab Knoten v : DFS-Visit(G, v)

Laufzeit (ohne Rekursion): $\Theta(\deg^+(v))$

Tiefensuche für alle Knoten: DFS-Visit(G)

Laufzeit: $\Theta(|V| + \sum_{v \in V} (\deg^+(v) + 1)) = \Theta(|V| + |E|)$

DFS-Visit(G, v)

DFS-Visit(G)

Input: Graph $G = (V, E)$

foreach $v \in V$ do

$v.color \leftarrow \text{white}$

foreach $v \in V$ do

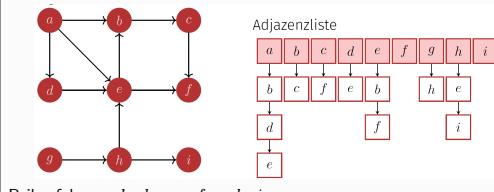
if $v.color = \text{white}$ then

DFS-Visit(G, v)

$v.color \leftarrow \text{black}$

8.4. Graphen Traversieren: Breitensuche

Verfolge zuerst Pfad in die Breite, gehe dann in die Tiefe.



BFS-Visit(G, v)

BFS-Visit(G)

Input: Graph $G = (V, E)$

Queue $Q \leftarrow \emptyset$

$v.color \leftarrow \text{grey}$

enqueue(Q, v)

while $Q \neq \emptyset$ do

$w \leftarrow \text{dequeue}(Q)$

foreach $c \in N^+(w)$ do

if $c.color = \text{white}$ then

$c.color \leftarrow \text{grey}$

enqueue(Q, c)

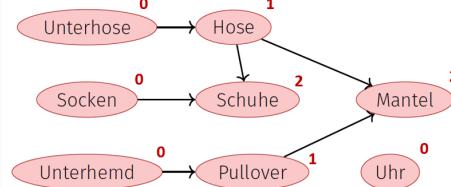
$w.color \leftarrow \text{black}$

Extraplatz: $\mathcal{O}(|V|)$

Laufzeit: $\Theta(|V| + |E|)$

8.5. Topologische Sortierung

Ein gerichteter Graph $G = (V, E)$ besitzt genau dann eine topologische Sortierung, wenn er kreisfrei ist.



Augmentiere den Eingangsgrad. Abarbeitung nur wenn Eingangsgrad 0 ist. Eingangsgrad verringern entspricht Knotenentfernen.

Topological-Sort(G)

Input: Graph $G = (V, E)$.

Output: Topologische Sortierung ord

Stack $S \leftarrow \emptyset$

foreach $v \in V$ do $A[v] \leftarrow 0$

foreach $(v, w) \in E$ do $A[w] \leftarrow A[w] + 1$ // Eingangsgrade berechnen

foreach $v \in V$ with $A[v] = 0$ do push(S, v) // Merke Nodes mit Eingangsgrad 0

$i \leftarrow 1$

while $S \neq \emptyset$ do

$v \leftarrow \text{pop}(S)$; $\text{ord}[v] \leftarrow i$; $i \leftarrow i + 1$ // Wähle Knoten mit Eingangsgrad 0

foreach $(v, w) \in E$ do // Verringere Eingangsgrad der Nachfolger

$A[w] \leftarrow A[w] - 1$

if $A[w] = 0$ then push(S, w)

if $i = |V| + 1$ then return ord else return "Cycle Detected"

Analysis

- Sei $G = (V, E)$ ein gerichteter, kreisfreier Graph. Der Algorithmus Topological-Sort berechnet in Zeit $\Theta(|V| + |E|)$ eine topologische Sortierung ord für G .
- Sei $G = (V, E)$ ein gerichteter, nicht-kreisfreier Graph. Der Algorithmus Topological-Sort terminiert in Zeit $\Theta(|V| + |E|)$ und detektiert den Zyklus.

8.6. Kürzeste Wege

Notation

$\delta(u, v) =$ Gewicht eines kürzesten Weges von u nach v

$$\delta(u, v) = \begin{cases} \infty & \text{kein Weg von } u \text{ nach } v \\ \min\{c(p) : u \xrightarrow{p} v\} & \text{sonst} \end{cases}$$

Beobachtungen

- Einfachster Fall: Kantengewicht 1 → Breitensuche
- Es gibt Situationen, in denen kein kürzester Weg existiert: negative Zyklen könnten auftreten.
- Es kann exponentiell viele Wege geben → alle Wege probieren ist ineffizient
- Ein kürzester Weg von s nach v (ohne weitere Einschränkungen) kann nicht länger sein als ein kürzester Weg von s nach v , der u enthalten muss.
- $\delta(s, v) \leq \delta(s, u) + \delta(u, v)$
- Optimale Substruktur: Teilstücke von kürzesten Pfaden sind kürzeste Pfade (Kürzester Pfad ⇒ kürzeste Subpfade)
- Kürzeste Wege enthalten keine Zyklen

8.6.1. Allgemeiner Algorithmus (Relaxier-Algorithmus)

Gesucht: Kürzeste Wege von einem Startknoten s aus.

- Gewicht des kürzesten bisher gefundenen Pfades
 - Zu Beginn: $d_s[v] = \infty$ für alle Knoten $v \in V$
 - Ziel: $d_s[v] = \delta(s, v)$ für alle $v \in V$
- Vorgänger eines Knotens: u Beginn $\pi_s[v]$ undefined für jeden Knoten $v \in V$

Algorithmus

- Initialisiere d_s und π_s : $d_s[v] = \infty$, $\pi_s[v] = \text{null}$ für alle $v \in V$
- Setze $d_s[s] \leftarrow 0$
- Wähle eine Kante $(u, v) \in E$:

Relaxiere (u, v) :

```
if  $d_s[v] > d_s[u] + c(u, v)$  then
   $d_s[v] \leftarrow d_s[u] + c(u, v)$ 
   $\pi_s[v] \leftarrow u$ 
```

- Wiederhole 3 bis nichts mehr relaxiert werden kann (bis $d_s[v] \leq d_s[u] + c(u, v) \quad \forall (u, v) \in E$)

Dijkstra(G, s)

Input: Positiv gewichteter Graph $G = (V, E, c)$, Startpunkt $s \in V$

Output: Minimale Gewichte d der kürzesten Pfade und Vorgängerknoten für jeden Knoten.

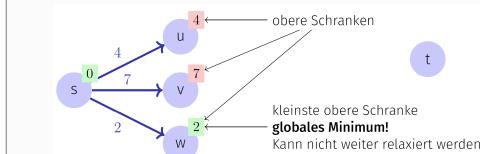
```
foreach  $u \in V$  do
   $d_s[u] \leftarrow \infty$ ;  $\pi_s[u] \leftarrow \text{null}$ 
 $d_s[s] \leftarrow 0$ ;  $R \leftarrow \{s\}$ 
while  $R \neq \emptyset$  do
   $u \leftarrow \text{ExtractMin}(R)$ 
  foreach  $v \in N^+(u)$  do
    if  $d_s[u] + c(u, v) < d_s[v]$  then
       $d_s[v] \leftarrow d_s[u] + c(u, v)$ 
       $\pi_s[v] \leftarrow u$ 
    if  $v \in R$  then
      DecreaseKey( $R, v$ )
    else
       $R \leftarrow R \cup \{v\}$ 
```

DecreaseKey (Aufsteigen im MinHeap), Position im Heap: Speichern am Knoten, Hashtabelle oder Lazy Deletion

Laufzeit

- $|V| \times \text{ExtractMin}: \mathcal{O}(|V| \log |V|)$
- $|E| \times \text{Insert oder DecreaseKey}: \mathcal{O}(|E| \log |V|)$
- $1 \times \text{Init}: \mathcal{O}(|V|)$
- Insgesamt: $\mathcal{O}(|E| \log |V|)$

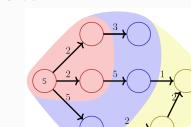
Beispiel Dijkstra



Grundidee

Menge V aller Knoten wird unterteilt in

- die Menge M von Knoten, für die schon ein kürzester Weg von s bekannt ist
- die Menge $R = \cup_{v \in M} N^+(v) \setminus M$ von Knoten, für die kein kürzester Weg bekannt ist, die jedoch von M direkt erreichbar sind.
- die Menge $U = V \setminus (M \cup R)$ von Knoten die noch nicht berücksichtigt wurden.



Betrachte alle Nachbarn der Menge M und füge den Knoten mit dem kürzesten Weg zu s der Menge M hinzu.

8.7. Minimale Spannbäume

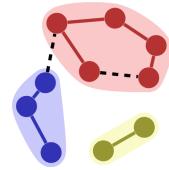
Problem

- Gegeben: Ungerichteter, zusammenhängender, gewichteter Graph $G = (V, E, c)$
 - Gesucht: Minimaler Spannbaum $T = (V, E') : \text{zusammenhängender, zyklusfreier Teilgraph } E' \subset E, \text{ so dass } \sum_{e \in E'} c(e) \text{ minimal.}$
- Greedy (gierige) Verfahren berechnen eine Lösung schrittweise, indem lokal beste Lösungen gewählt werden.

8.7.1. Union-Find Kruskal Algorithmus

Zur Implementation

Gegeben eine Menge von Mengen $i \equiv A_i \subset V$. Zur Identifikation von Schnitten und Kreisen: Zugehörigkeit der beiden Endpunkte einer Kante zu einer der Mengen.



Allgemeines Problem: Partition (Menge von Teilmengen) benötigt einen abstrakter Datentyp (**Union-Find**) mit folgenden Operationen:

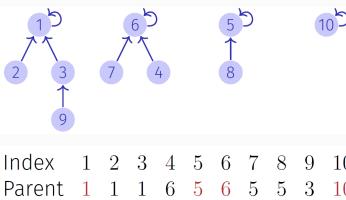
- Make-Set((i)): hinzufügen einer neuen Menge i
 $(p[i] \leftarrow i; \text{return } i)$
- Find (e): Name i der Menge, welche e enthält
(while($p[i] \neq 0$) do $i \leftarrow p[i]$; return i)
- Union(i, j): Vereinigung der Mengen mit Namen i und j
($p[j] = i$, wobei i und j die Wurzeln (Namen) sind.)

Laufzeitoptimierungen:

- a) Immer kleineren Baum an grösseren hängen
- b) Bei Find Knoten immer an den Parent hängen

Implementation von Union-Find

Idee: Baum für jede Teilmenge in der Partition, z.B. 1, 2, 3, 9, 7, 6, 4, 5, 8, 10, wobei die Baumwurzeln → Namen (Stellvertreter) der Mengen ist.



Algorithmus

```

Input: Gewichteter Graph  $G = (V, E, c)$ 
Output: Minimaler Spannbaum mit Kanten  $A$ .
Sortiere Kanten nach Gewicht  $c(e_1) \leq \dots \leq c(e_m)$ 
 $A \leftarrow \emptyset$ 
for  $k = 1$  to  $|V|$  do
    MakeSet( $k$ )
for  $k = 1$  to  $m$  do
     $(u, v) \leftarrow e_k$ 
    if Find( $u$ )  $\neq$  Find( $v$ ) then
        Union(Find( $u$ ), Find( $v$ ))
         $A \leftarrow A \cup e_k$ 
    else
        // konzeptuell:  $R \leftarrow R \cup e_k$ 
return  $(V, A, c)$ 

```

Laufzeit des Kruskal Algorithmus

- Sortieren der Kanten: $\Theta(|E| \log |E|) = \Theta(|E| \log |V|)$
- Initialisieren der Union-Find Datenstruktur $\Theta(|V|)$
- $|E| \times \text{Union}(\text{Find}(x), \text{Find}(y)) : \mathcal{O}(|E| \log |V|)$
- Insgesamt: $\Theta(|E| \log |V|)$

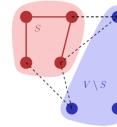
8.7.2. Algorithmus von Jarník, Prim, Dijkstra

Idee: Starte mit einem $v \in V$ und lasse von dort unter Verwendung der Auswahlregel einen Spannbaum wachsen:

```

 $A \leftarrow \emptyset$ 
 $S \leftarrow \{v\}$ 
for  $i \leftarrow 1$  to  $|V|$  do
    Wähle billigste  $(u, v)$  mit  $u \in S, v \notin S$ 
     $A \leftarrow A \cup \{(u, v)\}$ 
     $S \leftarrow S \cup \{v\}$  // (Färbung)

```



Bemerkungen

- Man braucht keine Union-Find Datenstruktur (Färbung reicht aus)
- Vorgehensweise:
 - Immer Knoten mit kleinstem Gewicht zur Menge S hinzufügen
 - Wenn der Knoten noch nicht in S ist → MST ist zyklusfrei

Laufzeit insgesamt: $\mathcal{O}(|E| \log |V|)$

9. Dynamische Programmierung

9.1. Idee

- Aufteilen eines komplexen Problems in eine vernünftige Anzahl kleinerer Teilprobleme
- Die Lösung der Teilprobleme wird zur Lösung des komplexeren Problems verwendet
- Identische Teilprobleme werden nur einmal gerechnet

→ Wir tauschen Laufzeit gegen Speicherplatz

9.2. Dynamic Programming vs. Divide-And-Conquer

- **Optimale Substruktur:** In beiden Fällen ist das Ursprungsproblem (einfacher) lösbar, indem Lösungen von Teilproblemen herangezogen werden können.
- Bei Divide-And-Conquer Algorithmen sind **Teilprobleme unabhängig**: deren Lösungen werden im Algorithmus nur einmal benötigt.
Beim DP sind Teilprobleme nicht unabhängig. Das Problem hat **überlappende Teilprobleme**, welche im Algorithmus mehrfach gebraucht werden.
- Identische Teilprobleme werden nur einmal gerechnet d.h. **keine zirkulären Abhängigkeiten zwischen Teilproblemen**

9.3. Memoization

Memoization (sic) Abspeichern von Zwischenergebnissen.

- Bevor ein Teilproblem gelöst wird, wird Existenz eines entsprechenden Zwischenergebnis geprüft
- Existiert ein gespeichertes Zwischenergebnis bereits, so wird dieses verwendet.
- Andernfalls wird der Algorithmus ausgeführt und das Ergebnis wird entsprechend gespeichert

Beispiel Fibonacci

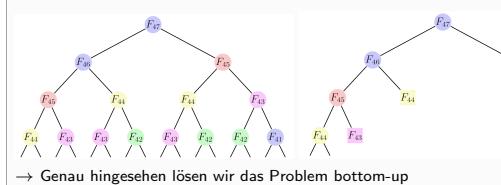
Input: $n \geq 0$

Output: n -te Fibonacci Zahl

```

if  $n \leq 2$  then
     $f \leftarrow 1$ 
else if  $\exists \text{memo}[n]$  then
     $f \leftarrow \text{memo}[n]$ 
else
     $f \leftarrow \text{FibonacciMemoization}(n - 1) + \text{FibonacciMemoization}(n - 2)$ 
     $\text{memo}[n] \leftarrow f$ 
return  $f$ 

```



9.7. Die Editierdistanz / Levenshteinabstand

Aufgabenstellung

Gesucht: Günstigste zeichenweise Transformation $A_n \rightarrow B_m$ mit Kosten

Operation	Levenshtein	LGT ²⁴	allgemein
c einfügen	1	1	ins(c)
c löschen	1	1	del(c)
Ersetzen $c \rightarrow c'$	$\mathbb{1}(c \neq c')$	$\infty \cdot \mathbb{1}(c \neq c')$	repl(c, c')

Beispiel

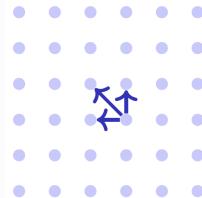
T I G E R T I _ G E R T → Z +E -R
Z I E G E Z I E G E _ Z → T -E +R

Wie findet man den DP Algorithmus?

- Genaue Formulierung der gesuchten Lösung:
 $E(n, m) = \text{minimale Anzahl Editieroperationen (ED Kosten) für } a_{1..n} \rightarrow b_{1..m}$
- Definiere Teilprobleme (und bestimme deren Anzahl):
Teilprobleme $E(i, j) = \text{ED von } a_{1..i}, b_{1..j}$ (Anz. $n \cdot m$)
- Raten / Aufzählen (und bestimme die Laufzeit für das Raten):
 $a_{1..i} \rightarrow a_{1..i-1}$ (löschen) $a_{1..i} \rightarrow a_{1..i} b_j$ (einfügen)
 $a_{1..i} \rightarrow a_{1..i-1} b_j$ (ersetzen)
- Rekursion: verbinde die Teilprobleme:

$$E(i, j) = \min \begin{cases} \text{del}(a_i) + E(i-1, j) \\ \text{ins}(b_j) + E(i, j-1) \\ \text{repl}(a_i, b_j) + E(i-1, j-1) \end{cases}$$

- Memoisieren / Tabellieren. Bestimme die Abhängigkeiten der Teilprobleme:



Berechnung von links oben nach rechts unten. Zeilen- oder Spaltenweise.

- Lösung des Problems: Lösung steht in $E(n, m)$

Bottom-Up Beschreibung

Dimension der Tabelle? Bedeutung der Einträge?

- Tabelle $E[0..n][0..m]$: $E[i, j] = \text{Minimaler Editierabstand der Zeichenketten } (a_1, \dots, a_i) \text{ und } (b_1, \dots, b_j)$

Berechnung eines Eintrags

- $E[0..i] \leftarrow i \quad \forall 0 \leq i \leq m, E[j..0] \leftarrow i \quad \forall 0 \leq j \leq n$. Berechnung von $E[i, j]$ sonst mit $E[i, j] = \min\{\text{del}(a_i) + E(i-1, j), \text{ins}(b_j) + E(i, j-1), \text{repl}(a_i, b_j) + E(i-1, j-1)\}$

Berechnungsreihenfolge

- Abhängigkeiten berücksichtigen: z.B. Zeilen aufsteigend und innerhalb von Zeilen Spalten aufsteigend.

Rekonstruktion einer Lösung?

- Beginne bei $j = m, i = n$. Falls $E[i, j] = \text{repl}(a_i, b_j) + E(i-1, j-1)$ gilt, gib $a_i \rightarrow b_j$ aus und fahre fort mit $(j, i) \leftarrow (j-1, i-1)$; sonst, falls $E[i, j] = \text{del}(a_i) + E(i-1, j)$ gib $\text{del}(a_i)$ aus fahre fort mit $j \leftarrow j-1$; sonst, falls $E[i, j] = \text{ins}(b_j) + E(i, j-1)$, gib $\text{ins}(b_j)$ aus und fahre fort mit $i \leftarrow i-1$. Terminiere für $i = 0$ und $j = 0$.

Analyse

Anzahl Tabelleneinträge: $(m+1) \cdot (n+1)$

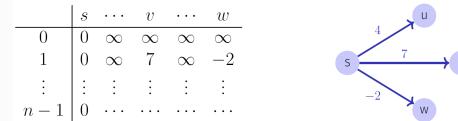
Laufzeit: $\mathcal{O}(m \cdot n)$

9.8. Kürzeste Wege: DP Ansatz (Bellman)

Induktion über Anzahl Kanten $d_s[i, v]$: Kürzeste Weglänge von s nach v über maximal i Kanten.

$$d_s[i, v] = \min \begin{cases} d_s[i-1, v] \\ \min_{(u,v) \in E} (d_s[i-1, u] + c(u, v)) \end{cases}$$

$$d_s[0, s] = 0, \underbrace{d_s[0, v] = \infty}_{\text{Zyklus}} \forall v \neq s$$



Algorithmus: Iteriere über letzte Zeile bis die Relaxationsstufen keine Änderung mehr ergeben, maximal aber $n-1$ mal. Wenn dann noch Änderungen, dann gibt es keinen kürzesten Pfad.

Bellmann-Ford(G,s)

Input: Graph $G = (V, E, c)$, Startpunkt $s \in V$

Output: Wenn Rückgabe true, Minimale Gewichte d der kürzesten Pfade zu jedem Knoten, sonst kein kürzester Pfad.

```
foreach u ∈ V do
    d[u] ← ∞; πs[u] ← null
d[s] ← 0;
for i ← 1 to |V| do
    f ← false
    foreach (u, v) ∈ E do
        f ← f ∨ Relax(u, v)
    if f = false then return true
return false;
```

Analysse

Laufzeit: $\mathcal{O}(|V| \cdot |E|)$

Speicherplatz: $\mathcal{O}(|V|^2) \rightsquigarrow$ eigentlich sogar $\mathcal{O}(|V|)$, da nur immer die letzte Zeile abgespeichert werden muss.

10. Code Beispiele

10.1. Beispiel: Levenshtein-Algorithmus

```
def Levenshtein(x, y):
    # D[m,n] = distance between x and y
    # D[i,j] = distance between strings x[1..i] and y[1..j]
    m = len(x)
    n = len(y)
    D = [[0 for i in range(m+1)] for j in range(n+1)]
    for j in range(0, m+1):
        D[0][j] = j
    for i in range(1, n+1):
        D[i][0] = i
        for j in range(1, m+1):
            # D[i,j] = min(
            # D[i-1,j-1] + 1
            # D[i-1,j-1] + d(x[i],y[j])
            # D[i-1,j] + 1
            # D[i,j-1] + 1 )
            q = D[i-1][j-1]
            if x[i-1] != y[j-1]:
                q += 1
            q = min(q, D[i-1][j-1]+1)
            q = min(q, D[i-1][j]+1)
            q = min(q, D[i][j-1]+1)
            D[i][j] = q
    return D[n][m]
```

10.2. Beispiel: Längste gemeinsame Teilfolge

```
import Data
import time

# compute longest ascending sequence for a point of the matrix
```

```
def LASR(A,L,y,x):
    if L[y][x] > 0:
        return L[y][x]
    maxLength = 0
    if x>0 and A[y][x] < A[y][x-1]:
        maxLength = max(maxLength, LASR(A,L,y,x-1))
    if y>0 and A[y][x] < A[y-1][x]:
        maxLength = max(maxLength, LASR(A,L,y-1,x))
    if y<len(A)-1 and A[y][x] < A[y+1][x]:
        maxLength = max(maxLength, LASR(A,L,y+1,x))
    if x>len(A[y])-1 and A[y][x] < A[y][x+1]:
        maxLength = max(maxLength, LASR(A,L,y,x+1))
    L[y][x] = maxLength + 1;
    return maxLength
```

```
root = self.a[root]
while self.a[i] != root:
    next = self.a[i]
    self.a[i] = root
    i = next
return root

# recursive version
def find_recursive(self,i):
    if self.a[i] == i:
        return i
    self.a[i] = self.find(self.a[i])
    return self.a[i]
```

10.4. Beispiel: Sliding Window

```
def main():
    text = input()

    map = {'a':0, 'b':0, 'c':0}
    bestl = -1
    bestr = len(text)
    l=0
    r=1
    num=0
    while r < len(text):
        if num == 3 and bestr-bestl > r-l:
            print("matrix a")
            Data.print_matrix(A)
            print("path lengths matrix")
            Data.print_matrix(L)

        print("maximum length",m)
        print("time:", stop-start, "s")
        r+=1
        num=0
        if num == 3 and bestr-bestl > r-l:
            bestl = l
            bestr = r
        if num >= 3:
            x = text[l]
            if x in map:
                xc = map[x]
                xc -= 1
                map[x] = xc
                if xc == 0:
                    num -= 1
            l += 1
        if r < len(text):
            x = text[r]
            if x in map:
                xc = map[x]
                xc += 1
                map[x] = xc
                if xc == 1:
                    num += 1
        if bestl == -1:
            print(text,"does not contain a,b AND c.")
        else:
            print("contains a,b,c between",bestl,"and",bestr)
```

10.5. Beispiel: Palindrome Checker

```
def isPalindrome(word):
    for i in range(0, len(word)//2):
        if word[i] != word[-1-i]:
            return False
    return True

def main():
    again, word = True, input("Enter a word: ")
    while again:
        if isPalindrome(word):
            cprint(word + ' is a palindrome!')
        else:
            cprint(word + ' is not a palindrome')
        word = input("Enter a word (or just <ENTER> to stop): ")
        again = len(word) > 0
```

11. Anhang

11.1. Nützliche Formeln für asymptotische Laufzeiten

Gauss'sche Summenformel

$$\sum_{i=0}^n i = \frac{n \cdot (n+1)}{2} \in \theta(n^2)$$

Binomialkoeffizient

$$\binom{n}{k} = \frac{n!}{k! \cdot (n-k)!} = \frac{\overbrace{n \cdot (n-1) \cdots (n-k)}^{k\text{-Faktoren}}!}{k! \cdot (n-k)!} \in \theta(n^k)$$

Spezielle Summen

$$\sum_{i=0}^{10n} \log n^n \in \theta(10 \cdot n \cdot \log n^n) \in \theta(n^2 \cdot \log n)$$