

Informatik II

1. Algorithmen und Datenstrukturen

1.1. Terminologie

Algorithmus: Wohldefinierte Berechnungsvorschrift, welche aus Eingabedaten (input/Probleminstanz) Ausgabedaten (output) berechnet.

Datenstrukturen: Eine Datenstruktur organisiert Daten so in einem Computer, dass man sie (in den darauf operierenden Algorithmen) effizient nutzen kann.

Effizienz: Die Effizienz eines Algorithmus ist seine Sparsamkeit bezüglich der Ressourcen, Zeit und Speicherplatz, die er zur Lösung eines festgelegten Problems beansprucht.

1.2. Effizienz von Algorithmen

Asymptotische Laufzeiten

- **Obere Schranke:** $\mathcal{O}(g) = \{f : \mathbb{N} \rightarrow \mathbb{R} \mid \exists c > 0, \exists n_0 \in \mathbb{N} : \forall n \geq n_0 : 0 \leq f(n) \leq c \cdot g(n)\}$
- **Untere Schranke:** $\Omega(g) = \{f : \mathbb{N} \rightarrow \mathbb{R} \mid \exists c > 0, \exists n_0 \in \mathbb{N} : \forall n \geq n_0 : 0 \leq c \cdot g(n) \leq f(n)\}$
- **Scharfe Schranke:** $\Theta(g) := \Omega(g) \cap \mathcal{O}(g)$
 $\Theta(f) = \{g : \mathbb{N} \rightarrow \mathbb{R} \mid \exists c > 0, \exists n_0 \in \mathbb{N} : \forall n \geq n_0 : 0 \leq \frac{1}{c} \cdot f(n) \leq g(n) \leq c \cdot f(n)\}$

Theorem

Seien $f, g : \mathbb{N} \rightarrow \mathbb{R}^+$ zwei Funktionen. Dann gilt.

1. $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0 \Rightarrow f \in \mathcal{O}(g), \mathcal{O}(f) \subseteq \mathcal{O}(g)$
2. $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = C > 0$ (C konstant) $\Rightarrow f \in \Theta(g)$
3. $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} \rightarrow \infty \Rightarrow g \in \mathcal{O}(f), \mathcal{O}(g) \not\subseteq \mathcal{O}(f)$

Beispiel aufsteigende Laufzeiten:

$2^{16}, \log(n^4), \log^8(n), \sqrt{n}, n \log n, \binom{n}{3}, n^5 + n, \frac{2^n}{n^2}, n!, n^n$

1.3. Analyse mit Rekurrenz und Teleskopie

```
void g(int n) {
    if (n>1) {
        g(n/2);
        g(n/2);
    }
    else {
        f();
    }
}
```

Rekurrenz ($n = 2^i$)

$$T(n) = \begin{cases} 2T(n/2) & n > 1 \\ 1 & n = 1 \end{cases}$$

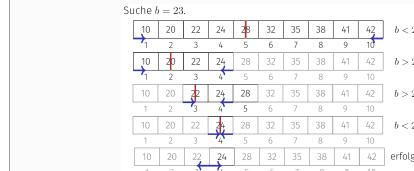
Teleskopieren

$$T(n) = 2 \cdot T(n/2) = 2 \cdot (2 \cdot T(n/4)) = 2^i \cdot T(n/2^i) = n \cdot T(n/n) \in \Theta(n)$$

2. Suchen

2.1. Divide and Conquer

Gegeben: Sortiertes Array A mit n Elementen und einen Schlüssel b
Gesucht: Index k mit $A[k] = b$
Lösung: Zeiger und Halbierung des Arrays



2.2. Binärer Suchalgorithmus: BSearch(A,l,r,b)

Input: Sortiertes Array A von n Schlüsseln. Schlüssel b . Bereichsgrenzen $1 \leq l, r \leq n$ mit $l \leq r$ oder $l = r + 1$.
Output: Index $m \in [l, \dots, r+1]$, so dass $A[i] \leq b$ für alle $l \leq i < m$ und $A[i] \geq b$ für alle $m < i \leq r$.

$m \leftarrow \lfloor (l+r)/2 \rfloor$

if $l > r$ **then** // erfolglose Suche

return l

else if $b = A[m]$ **then** // gefunden

return m

else if $b < A[m]$ **then** // Element liegt links

return BSearch($A, l, m-1, b$)

else // $b > A[m]$: Element liegt rechts

return BSearch($A, m+1, r, b$)

3. Sortieren

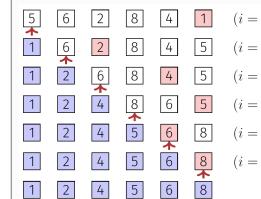
3.1. Laufzeiten von Sortier-Algorithmen

Algorithmus	Vergleiche		Vertauschungen	
	average	worst	average	worst
Bubble Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
Auswahl	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$	$\Theta(n)$
Einfügen	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n^2)$	$\Theta(n^2)$
Quicksort	$\Theta(n \log n)$	$\Theta(n^2)$	$\Theta(n \log n)$	$\Theta(n \log n)^*$
Mergesort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$

3.2. Bubble-Sort

```
void bubbleSort(std::vector<T> vec) {
    n = vec.size();
    for (unsigned int i = 0; i < n; ++i) {
        for (unsigned int j = 0; j < n - i - 1; ++j) {
            if (vec[j] > vec[j+1]) {
                std::swap(vec[j], vec[j+1]);
            }
        }
    }
}
```

3.3. Sortieren durch Auswahl



Auswahl des kleinsten Elements durch Suche im unsortierten Teil $A[:n]$ des Arrays.

Tausche kleinstes Element an das erste Element des unsortierten Teiles.

Unsortierter Teil wird ein Element kleiner ($i \rightarrow i+1$). Wiederhole bis alles sortiert.

Selection Sort

Input: Array $A = (A[1], \dots, A[n])$, $n \geq 0$.

Output: Sortiertes Array A

```
for i ← 1 to n - 1 do
    p ← i
    for j ← i + 1 to n do
        if A[j] < A[p] then
            p ← j;
    swap(A[i], A[p])
```

Analyse

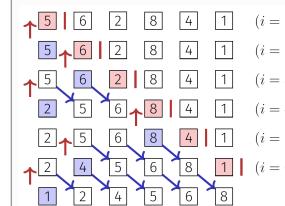
Anzahl Vergleiche im schlechtesten Fall:

$$\Theta(n^2)$$

Anzahl Vertauschungen im schlechtesten Fall:

$$n - 1 = \Theta(n)$$

3.4. Sortieren durch Einfügen



Iteratives Vorgehen:
 $i = 1 \rightarrow n$

Einfügeposition für Element i bestimmen.
Element i einfügen, ggfs. Verschiebung nötig.

Selection Sort

Input: Array $A = (A[1], \dots, A[n])$, $n \geq 0$.

Output: Sortiertes Array A

```
for i ← 2 to n do
    x ← A[i]
    p ← BinarySearch(A, 1, i - 1, x); // Kleinstes p ∈ [1, i] mit A[p] ≥ x
    for j ← i - 1 downto p do
        A[j + 1] ← A[j]
    A[p] ← x
```

Analyse

Nachteil: Im schlechtesten Fall viele Elementverschiebungen. / Vorteil: Der Suchbereich (Einfügebereich) ist bereits sortiert → binäre Suche möglich.

Anzahl Vergleiche im schlechtesten Fall:

$$\Theta(n \log(n))$$

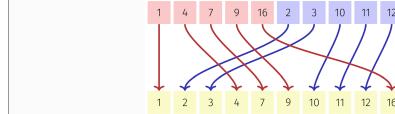
Anzahl Vertauschungen im schlechtesten Fall:

$$\Theta(n^2)$$

3.5. Mergesort

3.5.1. Merge

Minimum von A kann mit 2 Vergleichen ermittelt werden.



Merge(A,l,m,r)

Input: Array A der Länge n , Indizes $1 \leq l \leq m \leq r \leq n$. $A[l, \dots, m], A[m+1, \dots, r]$ sortiert

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

$B \leftarrow \text{new Array}(r - l + 1)$

$i \leftarrow l; j \leftarrow m + 1; k \leftarrow 1$

while $i \leq m$ **do**

if $A[i] \leq A[j]$ **then** $B[k] \leftarrow A[i]; i \leftarrow i + 1$
 else $B[k] \leftarrow A[j]; j \leftarrow j + 1$
 $k \leftarrow k + 1$

while $j \leq r$ **do** $B[k] \leftarrow A[j]; j \leftarrow j + 1; k \leftarrow k + 1$

for $k \leftarrow l$ **to** r **do** $A[k] \leftarrow B[k - l + 1]$

3.5.2. Mergesort



Mergesort(A,l,r) → rekursive Variante

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

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

if $l < r$ **then**

```
m ← ⌊(l+r)/2⌋ // Mittlere Position
Mergesort(A, l, m) // Sortiere vordere Hälfte
Mergesort(A, m+1, r) // Sortiere hintere Hälfte
Merge(A, l, m, r) // Verschmelzen der Teilstücke
```

Analyse: Laufzeit $\Theta(n \log(n))$; zusätzlicher Speicherbedarf: $\Theta(n)$

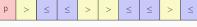
3.6. Quicksort

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

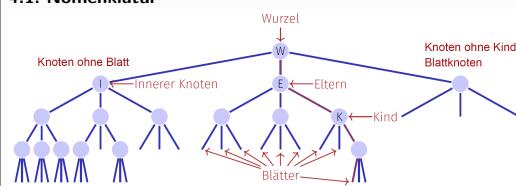
```

  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.
Im schlechtesten Fall: $\Theta(n^2)$

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 $\mathcal{O}(h(T))$.

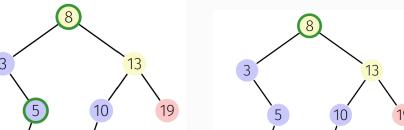
4.2.2. Operationen

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!

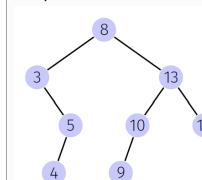
Grafik zu dem symmetrischen Vorgänger (links) oder Nachfolger (rechts)



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;
  bool insert(int key);
  bool 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 BST contains the given key.
bool BST::remove(int key) {
  if (root == nullptr) {
    return true;
  }
  else if (key < root->key) {
    curr = root->left;
  }
  else if (key > root->key) {
    curr = root->right;
  }
  else {
    assert(key == root->key);
    curr = root->right;
  }
}

// POST: Prints the tree's keys in preorder traversal to out.
void Node::print_preorder(std::ostream& out) const {
  out << key << ' ';
  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) {
  if (curr->left == nullptr) {
    curr->left = new Node(new_key);
    return true;
  }
  else if (curr->right == nullptr) {
    curr->right = new Node(new_key);
    return true;
  }
  else {
    assert(new_key > curr->key);
    if (curr->right == nullptr) {
      curr->right = new Node(new_key);
      return true;
    }
    else {
      curr = curr->right;
    }
  }
  return false;
}
  
```


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.

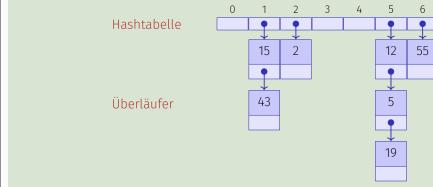
→ Resultiert im worst case in $\Theta(n^2)$ pro Operation

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 Annahme: 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 einfacherem 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**.

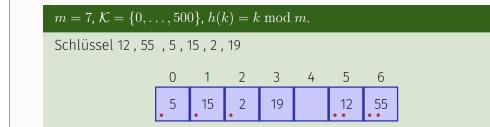
7.5.1. Lineares Sondieren

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

Problem → Primäre Häufung:

Ähnliche Hashadressen haben ähnliche Sondierungsfolgen → lange zusammenhängende belegte Bereiche.

Beispiel



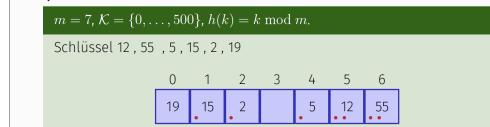
7.5.2. Quadratisches Sondieren

$$s(k, j) = h(k) + \lceil j/2 \rceil^2 (-1)^{j+1} \\ S(k) = (h(k), h(k)+1, h(k)-1, h(k)+4, h(k)-4, \dots) \bmod m$$

Problem → Sekundäre Häufung:

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

Beispiel



7.5.3. Double Hashing

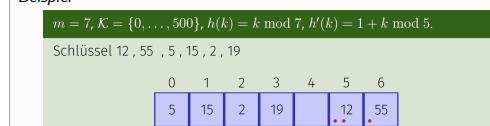
Verwendung von zwei Hashfunktionen $h(k)$ und $h'(k)$ → Vermeidung primärer und sekundärer Häufungen.

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

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



7.5.4. Beispiele

- $h'(k) = \lceil \ln(k+1) \rceil \bmod q$: This function is not suitable as a second hash function, because for the key $k = 0$ we have $h'(0) = \lceil \ln(1) \rceil = 0$.
- $s(j, k) = k^j \bmod p$: This function is not suitable as a probing function, because for the keys $k = 0$ and $k = 1$, the function $s(j, k)$ has constant value of 0 and 1.
- $s(j, k) = ((k \cdot j) \bmod q) + 1$: This function is also not suitable as a probing function because its value is constant 1 if the key k is a multiple of q .
Moreover, for all other keys, the image of $s(j, k)$ is $\{1, \dots, q\}$, i.e., $p - q$ addresses of the hash table cannot be reached.

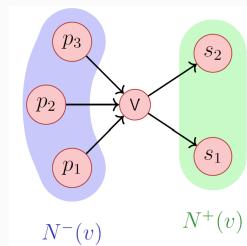
8. Graphen

8.1. Terminologie

Ein Graph $G = (V, E)$ besteht aus der Menge von Knoten $V = \{v_1, \dots, v_{1n}\}$ 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: $\deg^+(v) := |N^+(v)|$



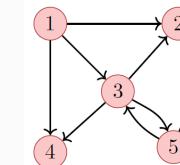
Beobachtungen

- 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

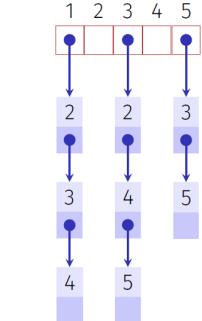
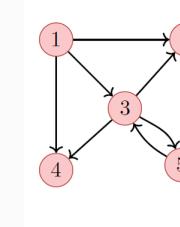
Graph $G = (V, E)$ mit Knotenmenge v_1, \dots, v_n gespeichert als Adjazenzmatrix $A_G = (a_{ij})_{1 \leq i, j \leq n}$ mit Einträgen aus $\{0, 1\}$. $a_{ij} = 1$ genau dann wenn Kante von v_i nach v_j . Speicherbedarf $\Theta(|V|^2)$. A_G ist symmetrisch, wenn G ungerichtet.



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

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 verkettete Liste aller Knoten in $N^+(v_i)$. Speicherbedarf $\Theta(|V| + |E|)$.



8.2.3. Laufzeiten einfacher Operationen

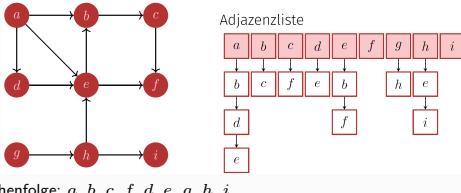
Operation

Matrix	Liste
--------	-------

$\Theta(n)$	$\Theta(\deg^+ v)$
$\Theta(n^2)$	$\Theta(n)$
$\Theta(1)$	$\Theta(\deg^+ v)$
$\Theta(1)$	$\Theta(1)$
$\Theta(1)$	$\Theta(\deg^+ v)$

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

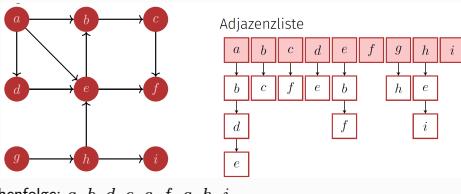
 if $v.color = \text{white}$ then

 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)

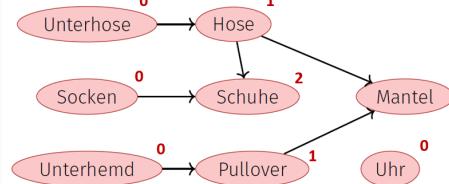
$w.color \leftarrow \text{black}$

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

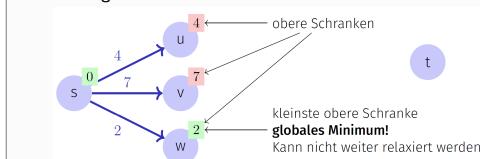
```
foreach  $u \in V$  do
     $d_s[u] \leftarrow \infty$ ;  $\pi_s[u] \leftarrow \text{null}$ 
 $d_s[s] \leftarrow 0$ ;  $R \leftarrow \{s\}$ 
while  $R \neq \emptyset$  do
     $u \leftarrow \text{ExtractMin}(R)$ 
    foreach  $v \in N^+(u)$  do
        if  $d_s[u] + c(u, v) < d_s[v]$  then
     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

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

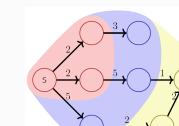
Beispiel Dijkstra



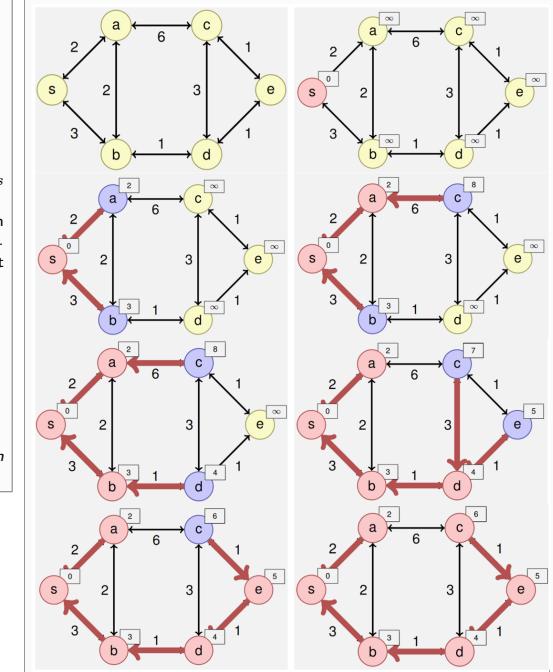
Grundidee

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. Floyd-Warshall-Algorithmus

Input: Graph $G = (V, E, c)$ ohne Tyklen mit negativem Gewicht.

Output: Minimale Gewichte aller Pfade d

```

 $d^0 \leftarrow c$ 
for  $k \leftarrow 1$  to  $|V|$  do
    for  $i \leftarrow 1$  to  $|V|$  do
        for  $j \leftarrow 1$  to  $|V|$  do
             $d^k(v_i, v_j) = \min\{d^{k-1}(v_i, v_j), d^{k-1}(v_i, v_k) + d^{k-1}(v_k, v_j)\}$ 

```

Laufzeit: $\Theta(|V|^3)$

Bemerkung: Der Algorithmus kann auf einer einzigen Matrix d (in place) ausgeführt werden.

8.8. A*-Algorithmus

Idee: Abstandsheuristik \hat{h} lenkt Algorithmus in richtige Richtung. Diese Heuristik muss den echten Abstand zu t unterschätzen und zum Abstand d_s addiert: $f = \hat{h} + d_s$

Voraussetzungen

- Positiv gewichteter endlicher Graph $G = (V, E, c)$
- $s \in V, t \in V$
- Abstandsabschätzung: $\hat{h}_t(v) \leq h_t(v) := \delta(v, t) \quad \forall v \in V$
- Gesucht: kürzester Pfad: $P: s \rightarrow t$

Bemerkungen

- Mehrfaches Entnehmen & Einfügen von R beim A*-Algorithmus möglich \Rightarrow Eventuell suboptimales Laufzeitverhalten
- Falls \hat{h} zulässig ($\hat{h}(v) \leq h(v) \forall v \in V$) und zusätzliche monoton ($\hat{h}(v) \leq h(v) + c(u', u) \forall (u', u) \in E$) ist, entspricht A* dem Dijkstra-Algorithmus mit $\tilde{c} = c(u, v) + \hat{h}(u) - \hat{h}(v)$, dann wird das mehrfache einfügen in R vermieden.

A*-Algorithmus(G, s, t, \hat{h})

Input: Positiv gewichteter Graph $G = (V, E, c)$, Startpunkt $s \in V$, Endpunkt $t \in V$, Schätzung $\hat{h}(v) \leq \delta(v, t)$

Output: Existenz und Wert eines kürzesten Pfades von s nach t

```

foreach  $u \in V$  do
     $d[u] \leftarrow \infty; f[u] \leftarrow \infty; \pi[u] \leftarrow \text{null}$ 
 $d[s] \leftarrow 0; f[s] \leftarrow \hat{h}(s); R \leftarrow \{s\}; M \leftarrow \{\}$ 
while  $R \neq \emptyset$  do
     $u \leftarrow \text{ExtractMin}_{\hat{h}}(R); M \leftarrow M \cup \{u\}$ 
    if  $u = t$  then return success
    foreach  $v \in N^+(u)$  with  $d[v] > d[u] + c(u, v)$  do
         $d[v] \leftarrow d[u] + c(u, v); f[v] \leftarrow d[v] + \hat{h}(v); \pi[v] \leftarrow u$ 
         $R \leftarrow R \cup \{v\}; M \leftarrow M - \{v\}$ 
return failure

```

8.9. Minimale Spannbäume

Problem

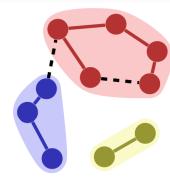
- Gegeben: Ungerichteter, zusammenhängender, gewichteter Graph $G = (V, E, c)$
- Gesucht: Minimaler Spannbaum $T = (V, E') :$ zusammenhängender, zyklenfreier Teilgraph $E' \subset E$, so dass $\sum_{e \in E'} c(e)$ minimal.

Greedy (gierige) Verfahren berechnen eine Lösung schrittweise, indem lokal beste Lösungen gewählt werden.

8.9.1. 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 abstrakten Datentyp (**Union-Find**) mit folgenden Operationen:

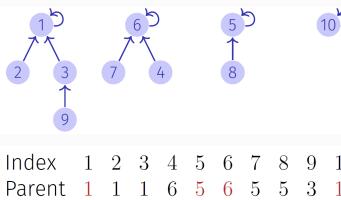
- MakeSet((i)): Hinzufügen einer neuen Menge i ($p[i] \leftarrow i; \text{return } i$)
- Find (e): Name i der Menge, welche e enthält (while $p[i] \neq 0$ do $i \leftarrow p[i]$; return i)
- Union(i, j): Vereinigung der Mengen mit Namen i und j ($p[j] = i$, wobei i und j die Wurzeln (Namen) sind.)

Laufzeitoptimierungen:

- Immer kleineren Baum an grösseren hängen
- 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 \rightarrow 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 $\text{Find}(u) \neq \text{Find}(v)$ then
 Union($\text{Find}(u), \text{Find}(v)$)
 $A \leftarrow A \cup e_k$
 else
 // konzeptuell: $R \leftarrow R \cup e_k$
return (V, A)

Laufzeit des Kruskal Algoritmus

- 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:** $\Theta(|E| \log |V|)$

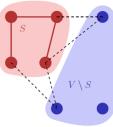
8.9.2. Algorithmus von Jarník, Prim, Dijkstra

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

```

 $A \leftarrow \emptyset$ 
 $S \leftarrow \{v_0\}$ 
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 \rightarrow MST ist zyklenfrei

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 Ursprungproblem (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

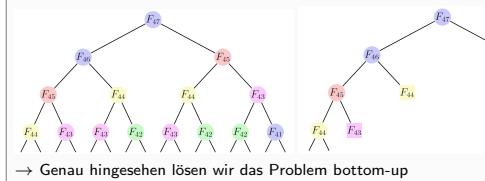
Beispiel Fibonacci

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  $f$ 

```



9.4. Dynamic Programming: Beschreibung am Beispiel

Beispiel Fibonacci

Dimension der Tabelle? Bedeutung der Einträge?

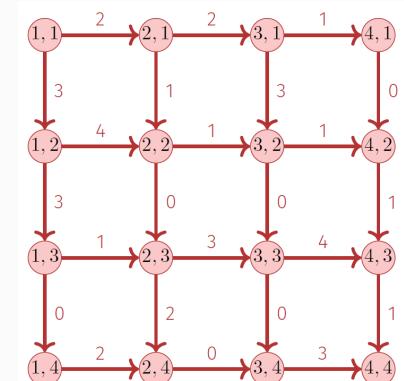
1. Tabelle der Grösse $n \times 1$. n -ter Eintrag enthält n -te Fibonacci Zahl.
2. Welche Einträge hängen nicht von anderen ab?
3. Werte F_1 und F_2 sind unabhängig einfach "berechenbar". Berechnungsreihenfolge?
4. F_i mit aufsteigenden i . Rekonstruktion einer Lösung?
5. F_n ist die n -te Fibonacci-Zahl.

9.5. Wie findet man den DP Algorithmus?

1. Genaue Formulierung der gesuchten Lösung
2. Definiere Teilprobleme (und bestimme deren Anzahl)
3. Raten / Aufzählen (und bestimme die Laufzeit für das Raten)
4. Rekursion: verbinde die Teilprobleme
5. Memoisieren / Tabellieren. Bestimme die Abhängigkeiten der Teilprobleme
6. Lösung des Problems:
 $\text{Laufzeit} = \text{Anz. Teilprobleme} \times \frac{\text{Zeit}}{\text{Teilproblem}}$

9.6. Beispiel Kaninchen

Ein Kaninchen sitzt auf Platz (1,1) eines $n \times n$ Gitters. Es kann nur nach Osten oder nach Süden gehen. Auf jedem Wegstück liegt eine Anzahl Rüben. Wie viele Rüben sammelt das Kaninchen maximal ein?



Rekursion

Gesucht: $T_{0,0}$ = Maximale Anzahl Rüben von (0,0) nach (n,n). Sei $w(i,j)-(i',j')$ Anzahl Rüben auf Kante von (i, j) nach (i', j') . Rekursion (maximale Anzahl Rüben von (i, j) nach (n, n))

$$T_{ij} = \begin{cases} \max\{w(i,j)-(i,j+1) + T_{i,j+1}, w(i,j)-(i+1,j) + T_{i+1,j}\}, & i < n, j < n \\ w(i,j)-(i,j+1) + T_{i,j+1}, & i = n, j < n \\ w(i,j)-(i+1,j) + T_{i+1,j}, & i < n, j = n \\ 0 & i = j = n \end{cases}$$

Teilabhängigkeit graph

- Richtung der Abhängigkeiten: Links oben nach rechts unten
- Richtung der Berechnung: Rechts unten nach links oben

Bottom-Up Beschreibung am Beispiel

Dimension der Tabelle? Bedeutung der Einträge?

1. Tabelle T der Grösse $n \times n$. Eintrag bei i, j enthält die maximale Anzahl Rüben von (i, j) nach (n, n) .
2. Welche Einträge hängen nicht von anderen ab?
3. Wert $T_{n,n}$ ist 0.
4. Berechnungsreihenfolge?
5. $T_{i,j}$ mit $i = n \searrow 1$ und für jedes $j: j = n \searrow 1$, (oder umgekehrt: $j = n \nearrow 1$ und für jedes $j: j = n \nearrow 1$).
6. Rekonstruktion einer Lösung?
7. $T_{1,1}$ enthält die maximale Anzahl Rüben

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	$\text{ins}(c)$
c löschen	1	1	$\text{del}(c)$
Ersetzen $c \rightarrow c'$	$\mathbb{1}(c \neq c')$	$\infty \cdot \mathbb{1}(c \neq c')$	$\text{rep}(c, c')$

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?

1. Genaue Formulierung der gesuchten Lösung:
 $E(n, m) = \text{minimale Anzahl Editieroperationen (ED Kosten) für } a_{1\dots n} \rightarrow b_{1\dots m}$
 2. Definiere Teilprobleme (und bestimme deren Anzahl):
Teilprobleme $E(i, j) = \text{ED von } a_{1\dots i}, b_{1\dots j}$ (Anz. $n \cdot m$)
 3. Raten / Aufzählen (und bestimme die Laufzeit für das Raten):
 $a_{1\dots i} \rightarrow a_{1\dots i-1}$ (löschen) $a_{1\dots i} \rightarrow a_{1\dots i} b_j$ (einfügen)
 $a_{1\dots i} \rightarrow a_{1\dots i-1} b_j$ (ersetzen)
 4. Rekursion: verbinde die Teilprobleme:
- $$E(i, j) = \min \left\{ \begin{array}{l} \text{del}(a_i) + E(i-1, j) \\ \text{ins}(b_j) + E(i, j-1) \\ \text{rep}(a_i, b_j) + E(i-1, j-1) \end{array} \right.$$
5. Memoisieren / Tabellieren. Bestimme die Abhängigkeiten der Teilprobleme:
-

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

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

Bottom-Up Beschreibung

Dimension der Tabelle? Bedeutung der Einträge?

1. Tabelle $E[0\dots m][0\dots n]$. $E[i, j]:$ Minimaler Editierabstand der Zeichenketten (a_1, \dots, a_i) und (b_1, \dots, b_j)
2. Berechnung eines Eintrags
 $E[0, i] \leftarrow i \forall 0 \leq i \leq m, E[j, 0] \leftarrow j \forall 0 \leq j \leq n$. Berechnung von $E[i, j]$ sonst mit $E[i, j] = \min\{\text{del}(a_i) + E(i-1, j), \text{ins}(b_j) + E(i, j-1), \text{rep}(a_i, b_j) + E(i-1, j-1)\}$
3. Berechnungsreihenfolge
4. Rekonstruktion einer Lösung?
Beginne bei $j = m, i = n$. Falls $E[i, j] = \text{rep}(a_i, b_j) + E(i-1, j-1)$ 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: $\mathcal{O}(m \cdot n)$

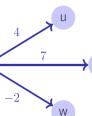
9.8. Kürzeste Wege: DP Ansatz (Bellman)

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

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

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

	s	\dots	v	\dots	w
0	0	∞	∞	∞	∞
1	0	∞	7	∞	-2
:	:	:	:	:	:
$n-1$	0	\dots	\dots	\dots	\dots



Algorithmus: Iteriere über letzte Zeile bis die Relaxationsschritte 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  $u \in V$  do
     $d_s[u] \leftarrow \infty; \pi_s[u] \leftarrow \text{null}$ 
 $d_s[s] \leftarrow 0;$ 
for  $i \leftarrow 1$  to  $|V|$  do
     $f \leftarrow \text{false}$ 
    foreach  $(u, v) \in E$  do
         $f \leftarrow f \vee \text{Relax}(u, v)$ 
        if  $f = \text{false}$  then return true
return false;
```

Analyse

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. Greedy-Algorithmen

10.1. Eigenschaften

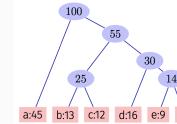
Ein rekursiv lösbares Optimierungsproblem kann mit einem **gierigen (greedy) Algorithmus** gelöst werden, wenn es die folgende Eigenschaften hat:

- Das Problem hat **optimale Substruktur**: die Lösung eines Problems ergibt sich durch Kombination optimaler Teillösungen.
- Es gilt die **greedy choice property**: Die Lösung eines Problems kann konstruiert werden, indem ein lokales Kriterium herangezogen wird, welches nicht von der Lösung der Teilprobleme abhängt.

10.2. Huffman-Code

Betrachten Präfixcodes: kein Codewort kann mit einem anderen Codewort beginnen.

Präfixcodes können im Vergleich mit allen Codes die optimale Datenkompression erreichen.



Huffman(C)

Input: Codewörter $c \in C$

Output: Wurzel eines optimalen Codebaums

```
n ← |C|
Q ← C
for i = 1 to n - 1 do
    Alloziere neuen Knoten z
    z.left ← ExtractMin(Q) // Extrahiere Wort mit minimaler Häufigkeit.
    z.right ← ExtractMin(Q)
    z.freq ← z.left.freq + z.right.freq
    Insert(Q, z)
return ExtractMin(Q)
```

Analyse: Heap bauen in $\mathcal{O}(n)$. Extract-Min in $\mathcal{O}(\log(n))$ für n Elemente. Somit Laufzeit $\mathcal{O}b(n \log(n))$.

11. Generic Programming

11.1. Type-Genericity

Class Templates

- Given an implementation, e.g. class BST, for a single, specific element type, e.g. int, replace each occurrence of the element type with a placeholder T
- Prepend the class with template<typename T>.
- Node<type> creates a type-specific instance of Node, using the substitution T=type. Therefore, Node<T> is sometimes called a **type constructor**.
- The compiler generates the code of each instantiated class for us.

Function Templates

- Given an implementation, e.g. BST::insert(), for a single, specific element type, e.g. int, replace each occurrence of the element type with a placeholder T.
- Prepend the function with template<typename T>.

Type Inference Generally, the types must be explicitly specified upon instantiation (e.g. Node<int>); with C++17, type inference improved. Type Checking Commonly, one has to enforce certain properties of a generic type, typical examples are: Default-Constructable, Iterable, Copyable, Comparable

```
template <typename T>
class Node {
    T key;
    Node* left, right;
public:
    Node(T t, Node* l, Node* r): key(t), left(l), right(r) {}
    bool contains(T search_key) const {
        if (search_key < key) {
            return left->contains(search_key);
        }
    }
}
```

```
else {...}
bool insert(T insert_key) { ... }

T max() const { ... }
...
```

```
// For free functions
template <typename T>
void swap(T& x, T& y) {
    T temp = x; x = y;
    y = temp;
}

// For free functions
template <typename Iter>
bool is_sorted(Iter begin, Iter end) {
    ...
}
```

```
// For operators
template <typename T>
ostream& operator<<(ostream& out, const Node<T> root) {
    ...
}

// For member functions
template <typename E>
class vector {
    ...
};

template <typename C>
void push_back_all(const C& other) {...}
```

11.2. Algorithmic Genericity

- Higher-Order functions: If a type-generic function takes a callable object as an argument, it is called a higher order function; these functions are **parametric in their functionality**.
- Functors: Callable Objects with a state

```
// generic filter function
template <typename C, typename P>
C filter(const C& src_data, P pred) {
    C data;
    for (const auto& e : src_data)
        if (pred(e)) data.push_back(e);
    return data;
}

// stateless predicate as function
bool is_nonneg(int i) {
    return 0 <= i; // lower bound fixed
}

// stateful predicate as functor
template <typename T> struct At_least {
    T min;
    At_least(T m): min(m) {};
    bool operator()(T i) const {
        return min <= i;
    }
};

int main () {
    std::vector<int> data = {-1,0,1,2,-2,4,5,-3};
    sel1 = filter(data, is_nonneg); // {0,1,2,4,5}
    sel2 = filter(data, At_least(-1)); // = {-1,0,1,2,4,5}
    sel3 = filter(data, At_least(4)); // = {4,5}
}
```

11.3. Lambda Expressions

anonymous functions - function object - function literal

In C++: just syntactic sugar, compiler generates a suitable functor

General form:

$[e_1, \dots, e_n] \quad (T_1 x_1, \dots, T_m x_m) \rightarrow R \{ \text{stmt} \}$

captures **parameters** **return type** **body**
(potentially zero) (potentially zero)

- [] no context access
- [x] x is copied – and the copy is const, i.e. can only be read
- [&x] x is accessible by reference
- [x, &y] x is copied, y is referenced
- [&] all necessary variables are referenced
- [=] all necessary variables are copied
- [&, x] all necessary variables are referenced, except x, which is copied
- [=, &x] all necessary variables are copied, except x, which is referenced

12. Parallel Programming

12.1. Parallelism

Syntax best explained by example:

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <thread>
#include <cassert>
#include <cmath>

// PRE: [begin, end) must be a valid interval.
// POST: Returns the sum of the values in [begin, end].
template <typename It, typename T = typename It::value_type>
void sum_seq(It begin, It end, T& result) {
    T sum = 0;

    for (It curr = begin; curr != end; ++curr)
        sum += *curr;

    result = sum;
}

// PRE: [begin, end) must be a valid interval.
// PRE: 0 < max_threads
// POST: Returns the sum of the values in [begin, end],
//       using at most max_threads threads.
template <typename It, typename T = typename It::value_type>
double sum_par(
    const It begin,
    const It end,
    unsigned max_threads)
{
    assert(0 < max_threads);

    // Prevents corner cases further down
    if (begin == end) return 0;

    // Using more threads than there are values to sum up would
    // yield a partition size of zero
    unsigned thread_count = std::min(
        (unsigned)(end - begin), max_threads);
    assert(thread_count != 0); // Should not happen

    std::vector<std::thread> threads; // Stores forked threads
    // Partial sums
    std::vector<double> partial_sums(thread_count, 0);

    // Summation will be parallelised by (conceptually) split-
    // ting the input interval into thread_count partitions
    unsigned partition_size = (end - begin) / thread_count;
    assert(partition_size != 0); // Should not happen

    It partition_begin = begin;

    // Fork sequential sum computations
    for (unsigned i = 0; i < thread_count - 1; ++i) {
        It partition_end = partition_begin + partition_size;
```

```
        auto worker = std::thread(
            [=,&partial_sums] () {
                T t;
                sum_seq(partition_begin, partition_end, t);
                partial_sums[i] = t;
            }
        );
        threads.push_back(std::move(worker));
    }

    // The current chunk's end is the next chunk's begin
    partition_begin = partition_end;
}

auto worker = std::thread(
    sum_seq<It, double>,
    partition_begin,
    end,
    std::ref(partial_sums[partial_sums.size() - 1]));
threads.push_back(std::move(worker));

// Iterate over the forked threads and wait for each thread
// to finish
for (auto& td : threads)
    td.join();

// As a last step we have to sum up the partial sums that
// the forked threads computed. We could parallelise this as
// well, but since partial_sums is typically a small vector,
// we sum it up sequentially.
double result = 0;
sum_seq(partial_sums.begin(), partial_sums.end(), result);

return result;
}
```

12.2. Concurrency

Important Concepts

- Nondeterministic Thread Scheduling: The scheduler can interrupt (pause) any thread at virtually any moment, to schedule another one
- ⇒ multiple threads are executed in a **non-deterministic order**, and many different thread-interleavings are possible
- A program has a **race condition** if, during any possible execution with the same inputs, its observable behaviour (results, output, ...) can change if events happen in different order
- A program has a **data race** if, during any possible execution, a memory location could be written from one thread, while concurrently being read or written from another thread

Options for Preventing Bad Interleavings

1. Share only immutable data (if possible)
2. Use atomic data types (if possible)
3. Use mutual exclusion to make arbitrary code atomic

Mutual Exclusion

- C++ provides std::mutex with .lock() and .unlock() for protecting sensitive code-regions.
- A **deadlock** is a situation in which the overall program cannot make any progress, because each thread waits for another thread to finish its action.
 1. Establish a strict total order between the shared resources
 2. Ensure that the resources are always acquired according to this order

Lock Guards Guard automatically locks mutex - and more importantly, also unlocks it at the end of the scope of the guard.

Different locks exist:

- std::lock_guard: basic lock for single mutex
- std::scoped_lock: multiple mutexes, prevents deadlocks (if used exclusively)
- std::unique_lock: single mutex, more control (e.g. when mutex is locked)
- std::shared_lock: for reader-writer situations
 - Many threads only read the shared data in their critical section
 - Few threads write the shared data
 - Reading in parallel is fine, but writers need exclusive access
 - Example: shared phonebook; much more often read than updated

```
void work(...) {
    some_mutex.lock();
    fun() // might throw exception
    some_mutex.unlock();
}
```

```
void foo(...) {
    std::lock_guard<std::mutex>
        guard(some_mutex);
    fun() // might throw exception
}
```

```
} else {
    std::cout << "Not found\n";
}
```

```
// iterate
for (const auto& entry : colours) {
    std::cout << entry.first << " : " << entry.second << ", ";
// BLUE : #0000FF, RED : #FF0000, GREEN : #00FF00,
}
```

14. Anhang

14.1. Nützliche Formeln für asymptotische Laufzeiten

Gauss'sche Summenformel

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

Binomialkoeffizient

$$\binom{n}{k} = \frac{n!}{k! \cdot (n-k)!} = \frac{n \cdot (n-1) \cdots (n-k)!}{k! \cdot (n-k)!} \in \Theta(n^k)$$

Spezielle Summen

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

14.2. Asymptotische Laufzeiten C++

	Wahlfreier Zugriff	Einfügen	Iteration nächstes
std::vector	$\Theta(1)$	$\Theta(1)$ A	$\Theta(1)$
std::list	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
std::set	-	$\Theta(\log(n))$	$\Theta(\log(n))$
std::unordered_set	-	$\Theta(1)$ P	-
		Einf. nach Element	Suchen (x in S)
std::vector		$\Theta(n)$	$\Theta(n)$
std::list		$\Theta(1)$	$\Theta(n)$
std::set		-	$\Theta(\log(n))$
std::unordered_set		-	$\Theta(1)$ P
A: Amortisiert			
P: Erwartet			
sonst: worst case			

13. Code Beispiele

13.1. Beispiel: Levenshtein-Algorithmus

```
#include <string>
#include <vector>

unsigned Levenshtein(const std::string& x,
                     const std::string& y){
    // D[n,m] = distance between x and y
    // D[i,j] = distance between strings x[i..i] and y[i..j]
    unsigned n = x.size();
    unsigned m = y.size();
    std::vector<std::vector<unsigned>> D(
        n+1, std::vector<unsigned>(m+1, 0)
    );
    for (unsigned j = 0; j <= m; ++j){
        D[0][j] = j;
    }
    for (unsigned i = 1; i <= n; ++i){
        D[i][0] = i;
        for (unsigned j = 1; j <= m; ++j){
            // D[i,j] = min{
            //   D[i-1,j-1] + d(x[i],y[j]),
            //   D[i-1,j] + 1
            //   D[i,j-1] + 1
            D[i][j] = std::min( {
                D[i-1][j-1] + (x[i-1] != y[j-1]),
                D[i-1][j] + 1,
                D[i][j-1] + 1
            });
        }
    }
    return D[n][m];
}
```

13.2. Dictionary in C++

```
#include <unordered_map>

// Create an unordered_map of strings that map to strings
std::unordered_map<std::string, std::string> colours = {
    {"RED", "#FF0000"}, {"GREEN", "#00FF00"}
};
colours["BLUE"] = "#0000FF"; //Add element
std::cout << "hex of red: " << colours["RED"] << "\n";

auto search = colours.find("BLUE"); //iterator to object
if (search != colours.end()) {
    std::cout << "Found " << search->first << " : " << search->second << '\n';
}
```