Karlsruher Institut für Technologie

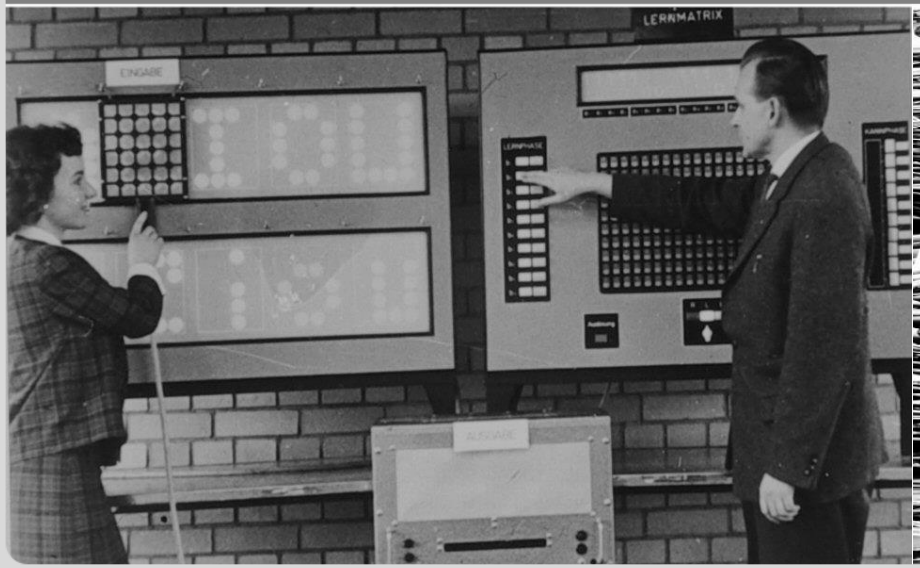
Institut für Technik der Informationsverarbeitung

Institutsleitung
Prof. Dr.-Ing. J. Becker
Prof. Dr.-Ing. E. Sax
Prof. Dr. rer. nat. W. Stork

Übung 2

Übung zu Informationstechnik II und Automatisierungstechnik – Nathalie Brenner

Prof. Dr.-Ing. Eric Sax



WIEDERHOLUNG ÜBUNG 1

Wiederholung Übung 1

Eigenschaften von Algorithmen

Algorithmen sind genau definierte Handlungsvorschriften zur Lösung eines Problems oder einer bestimmten Art von Problemen in endlich vielen Schritten

Beschreibung von Algorithmen

Merkmale und Eigenschaften

- Was fallen Ihnen für mögliche Eigenschaften von Algorithmen ein?

abbrechbar	Alg. Kann zu jedem beliebigen Zeitpunkt abgebrochen werden, liefert nicht optimales Ergebnis
Heuristisch	Mit geringem Rechenaufwand und kurzer Laufzeit zulässige Lösungen für bestimmtes Problem erhalten
Stabil	Der Alg. produziert für alle erlaubten und in der Größenordnung der Rechengenauigkeit gestörten Eingabedaten akzeptable Resultate
In place	Benötigt konstante Menge von Speicher, Eingabedaten werden von Ausgabedaten überschrieben
Inkrementell	Während der Laufzeit werden Ergebnisse hinzugefügt
Finitheit	Der Algorithmus benötigt zu jedem Schritt nur endlich viel Speicherplatz
Determiniertheit	Der Algorithmus liefert bei gleichen Voraussetzungen stets das gleiche Ergebnis
Ausführbarkeit	Jeder Einzelschritt muss ausführbar sein
Rekursiv	Algorithmus Ruft sich selber auf, aber es kommen keine Schleifen vor

Übung zu Informationstechnik II und Automatisierungstechnik — Institut für Technik der Informationsverarbeitung (ITIV)

- Laufzeitanalyse durch O-Notation

Asymptotische Laufzeit-komplexität	Bezeichnung	Erläuterung am Beispiel der Verdoppelung der Problemgröße und typische Algorithmen mit dieser Ordnung
$O(1)$	konstant	Laufzeit ist unabhängig von der Problemgröße, optimaler Fall, der praktisch nicht auftritt
$O(\log n)$	logarithmisch	Verdoppelung der Problemgröße bewirkt Anstieg der Laufzeit um $\log 2$, also um eine Konstante (um 1 für *logarithmus dualis*), sehr günstig und daher erstrebenswert, z.B. *binäre Suche* (siehe Kapitel 6)
$O(n)$	linear	Verdoppelung der Problemgröße bewirkt Verdoppelung der Laufzeit, immer noch zufrieden stellend, z.B. *sequenzielle Suche* (siehe Kapitel 6)
$O(n \log n)$	–	Fast so gut wie linear, weil $\log n$ im Verhältnis zu n klein ist, z.B. gute Sortierverfahren wie *Quicksort* (siehe Kapitel 7)
$O(n^2)$	quadratisch	Verdoppelung der Problemgröße bewirkt Vervierfachung der Laufzeit, ungünstig, z.B. schlechte Sortierverfahren wie *Bubblesort* (siehe Kapitel 7)
$O(n^3)$	kubisch	Verdoppelung der Problemgröße bewirkt Verachtfachung der Laufzeit, sehr unbefriedigend, z.B. einfache *Matrizenmultiplikation*
$O(k^n)$	exponentiell	Verdoppelung der Problemgröße bedeutet Quadrierung (weil $k^{2n} = (k^n)^2$) der Laufzeit, katastrophal, z.B. *Backtracking-* oder *Exhaustionsalgorithmen* (siehe Kapitel 9)

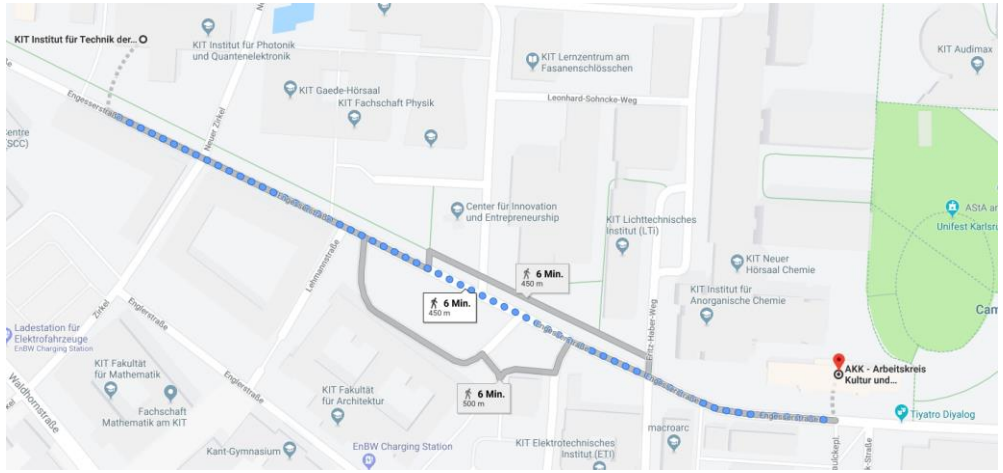
INHALT ÜBUNG 2

Sortier-, Such- und Optimierungsalgorithmen

Wozu braucht man diese Algorithmen?

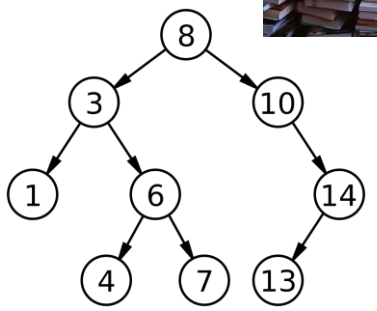
- 25% der Computer auf der Welt verbringen ihre Zeit mit Sortieren und Suchen!
- Durch einen sortierten Datenbestand kann eine Abfrage deutlich schneller bearbeitet werden!
  - Beim Einfügen der Daten
  - „Aufräumen“ der bestehenden Daten

- Durch geeignete Suchalgorithmen kann beispielsweise der kürzeste Pfad gefunden werden
  „vom ITIV bis zum Kaffee“
- Durch Optimierungsalgorithmen können diese Abfragen noch effizienter gestalten werden



Tiefensuche

Breitensuche



Bubble Sort ✓

Merge Sort

Insertion Sort ✓

Quick Sort

Ziele der heutigen Übung

- Nach der heutigen Übung können Sie….

- … bekannte Sortier-, Such- und Optimierungsalgorithmen gegenüberstellen und demonstrieren

1. … verschiedene Sortieralgorithmen, sowie deren Merkmale benennen
2. … verschiedene Sortieralgorithmen demonstrieren
3. … verschiedene Suchalgorithmen, sowie deren Merkmale benennen
4. … verschiedene Suchalgorithmen demonstrieren
5. … verschiedene Optimierungsalgorithmen, sowie deren Merkmale benennen
6. … verschiedene Optimierungsalgorithmen demonstrieren

SORTIERALGORITHMEN

Sortieralgorithmen

Wozu braucht man diese Algorithmen?

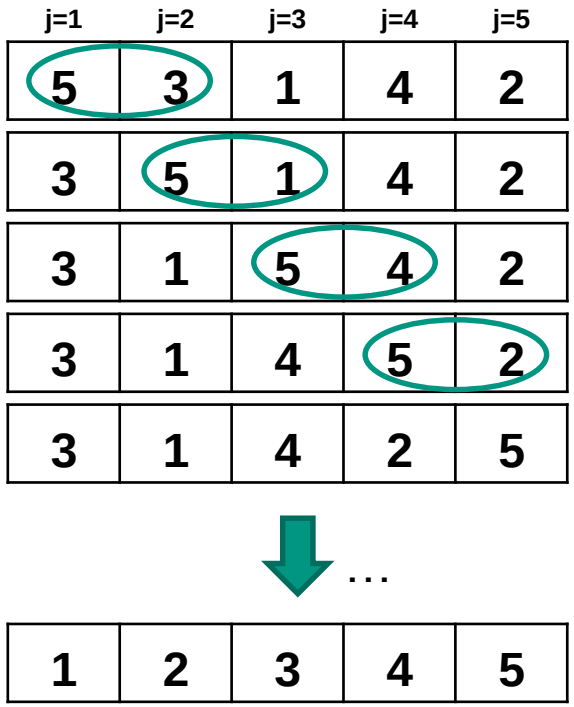
- Was wird alles sortiert?
- Was sortiert ein Computer?
- Warum werden Dinge sortiert?

Sortieralgorithmen

Wiederholung – Bubble Sort und Insertion Sort

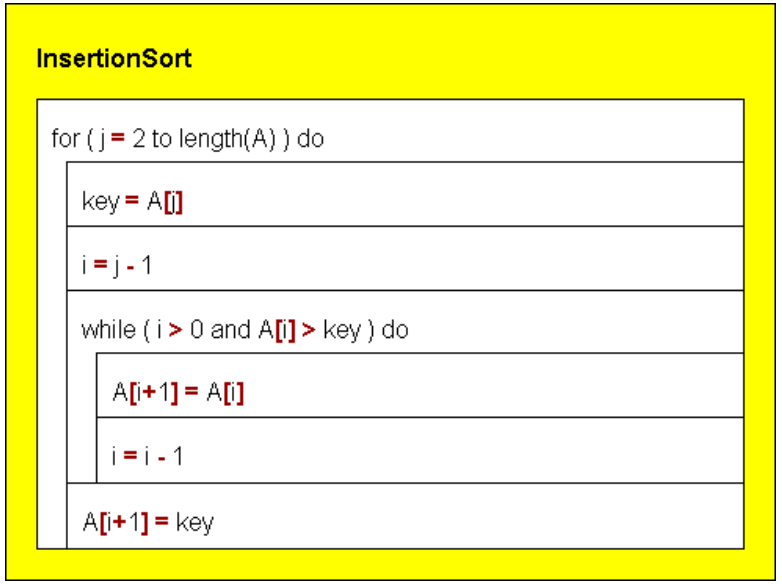
- Bubble Sort

```
A = [5, 3, 1, 4, 2]
for i = 1 to länge[A] – 1
  do for j = 1 to länge[A] – i
    do if A[j] > A[j+1]
      then vertausche A[j] mit A[j+1]
```



- Insertion Sort

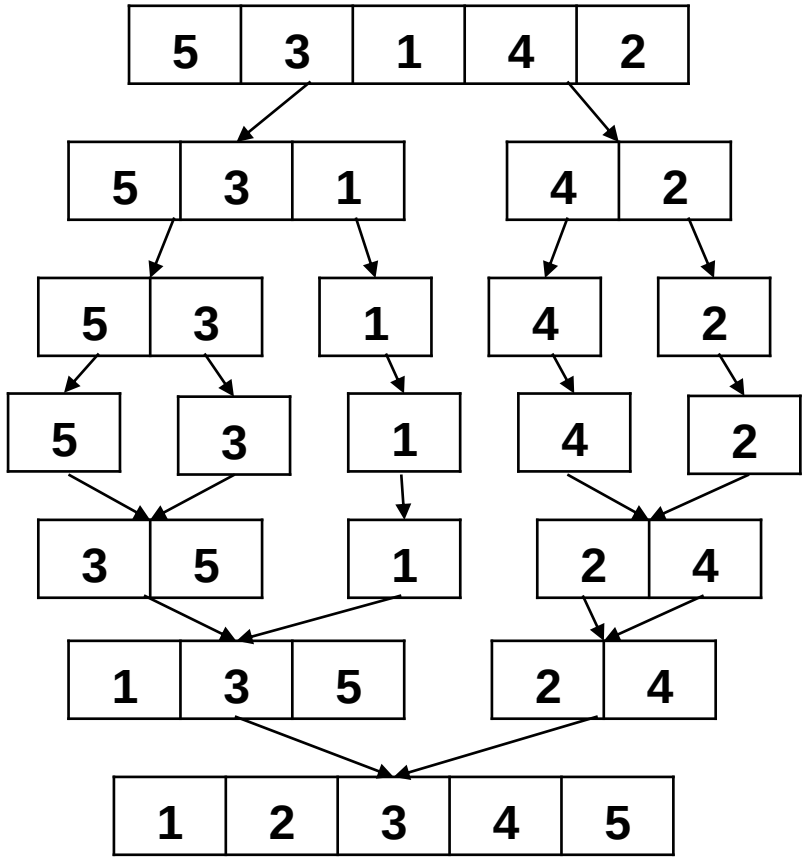
```
A = [5, 3, 1, 4, 2]
for j=2 to length(A){
     do key= A[j];
     I = j-1;
     while i>0 and A[i]>key{
          do A[i+1] = A[i];
          I = i-1;
     }
      A[i+1] = key
}
```



Sortieralgorithmen

Merge Sort

- Stabiler Sortieralgorithmus
- Besonders geeignet für verkettete Listen
- Prinzip: Teile und Herrsche/Divide and Conquer
  - Unsortierte Daten liegen als Liste vor
  - Zerlegung in immer kleinere Listen
  - Zusammenfügen durch Reißverschlussprinzip
  - Sortierte Daten liegen erneut als Liste vor

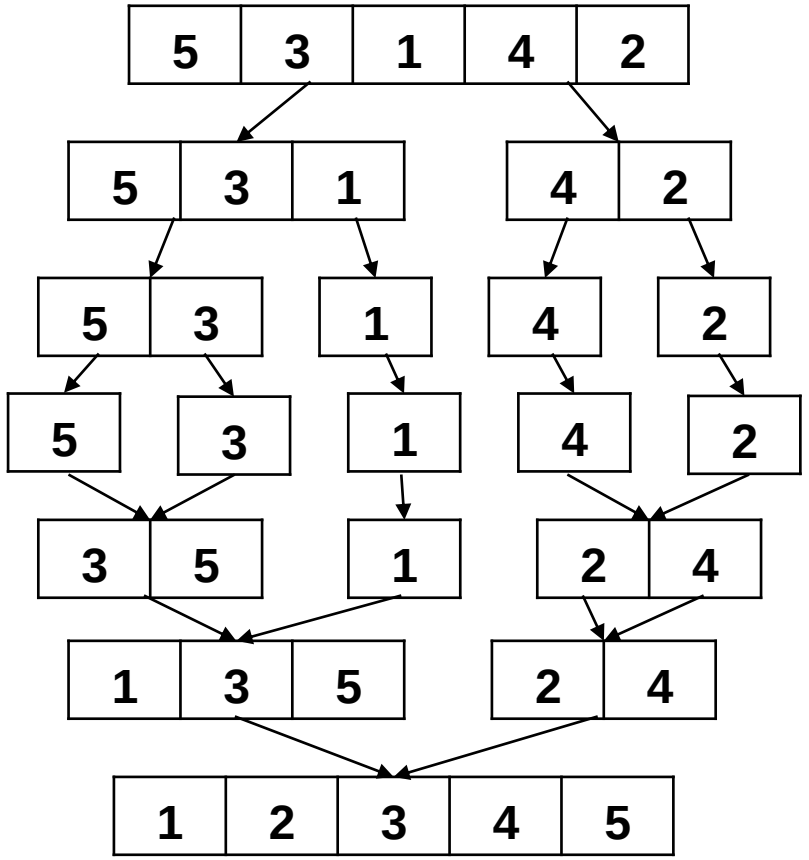


Sortieralgorithmen

Merge Sort

```
Mergesort(liste)
    if size(liste)>1
    then
      liste1=liste[erste Hälfte]
      liste2=liste[zweite Hälfte]
      liste1=mergesort(liste1)
      liste2=mergesort(liste2)
    return merge(liste1,liste2)
```

```
Merge(liste1,liste2)
    neueListe
    while liste1 und liste2 nicht leer
      do if liste1[1] > liste2[1]
        then liste1[1] zu neueListe & liste1[1] löschen
        else liste2[1] zu neueListe & liste2[1] löschen
      return neueListe
```



Sortieralgorithmen

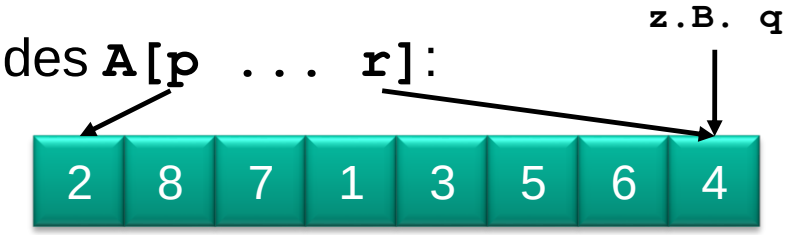
Quick Sort

- Schneller, rekursiver, nicht-stabiler Sortieralgorithmus nach dem Prinzip „Teile und Herrsche“
- Ca. 1960 von C. Antony R. Hoare in seiner Grundform entwickelt und seitdem von vielen Forschern verbessert
- Verfügt über eine sehr kurze innere Schleife (was die Ausführungsgeschwindigkeit stark erhöht)
- Kommt ohne zusätzlichen Speicherplatz aus (abgesehen von dem Platz auf dem Aufruf-Stack für die Rekursion)
- Im Mittel schnellster Sortieralgorithmus

Sortieralgorithmen

Quick Sort

- Drei Schritte für das Sortieren eines Feldes `A[p ... r]`:



a) Teile: Zerlege das Feld **A[p..r]** in zwei (möglicherweise leere) Teilfelder **A[p..q-1]** und **A[q+1..r]**. Dabei soll jedes Element von **A[p..q-1]** kleiner oder gleich **A[q]** und jedes Element von **A[q+1..r]** größer als **A[q]** sein. **A[q]** ist dabei ein beliebiges Element des Feldes. (Somit sind alle Werte von **A[p..q-1]** auch kleiner wie die Werte in **A[q+1..r]**.)

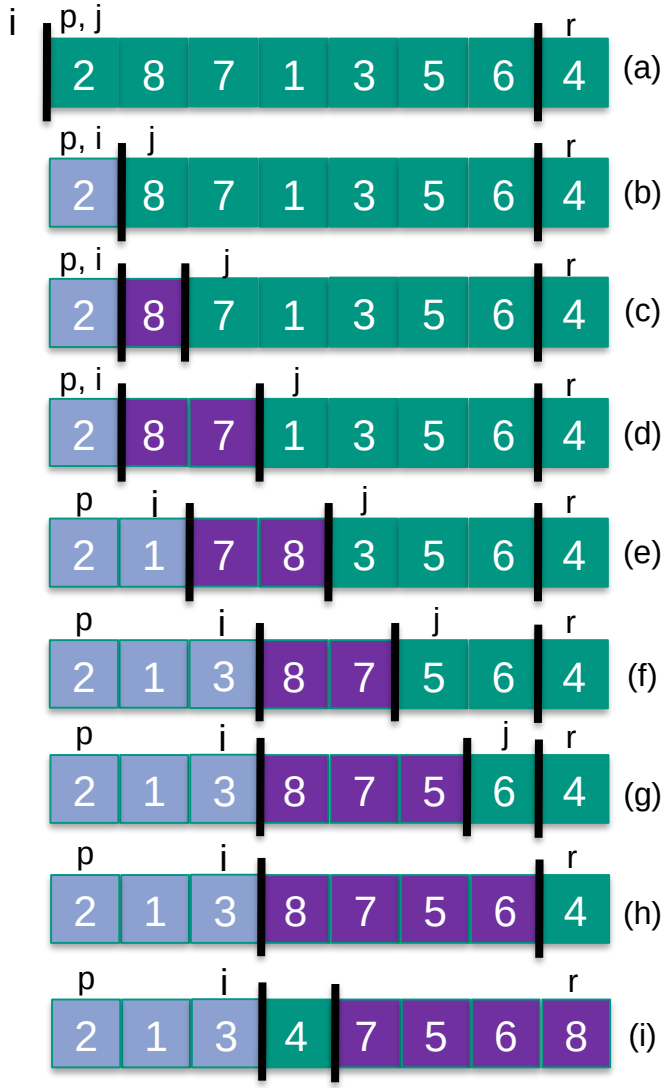
b) Beherrsche: Sortiere die beiden Teilfelder **A[p..q-1]** und **A[q+1..r]** wieder durch rekursiven Aufruf von QuickSort (Beginn wieder beim Teilen), wenn die Teilfelder mehr als ein Element haben.

c) Verbinde: Da die Teilfelder in-place sortiert werden (in Feld selbst), ist keine Arbeit erforderlich, um sie zu verbinden. Das gesamte Feld **A[p..r]** ist nun sortiert.

Sortieralgorithmen

Quick Sort - Pseudocode

```
Quicksort( A, 1, länge[A] )

Quicksort( A, p, r )
      if p < r
      then q = Partition( A, p, r )
        Quicksort( A, p, q - 1 )
        Quicksort( A, q + 1, r )

Partition( A, p, r )
      x = A[r]
      i = p - 1
      for j = p to r - 1
      do if A[j] <= x
        then i = i + 1
        vertausche A[i] mit A[j]
      vertausche A[i+1] mit A[r]
      return i+1
```



Sortieralgorithmen

Quick Sort – leicht erklärt

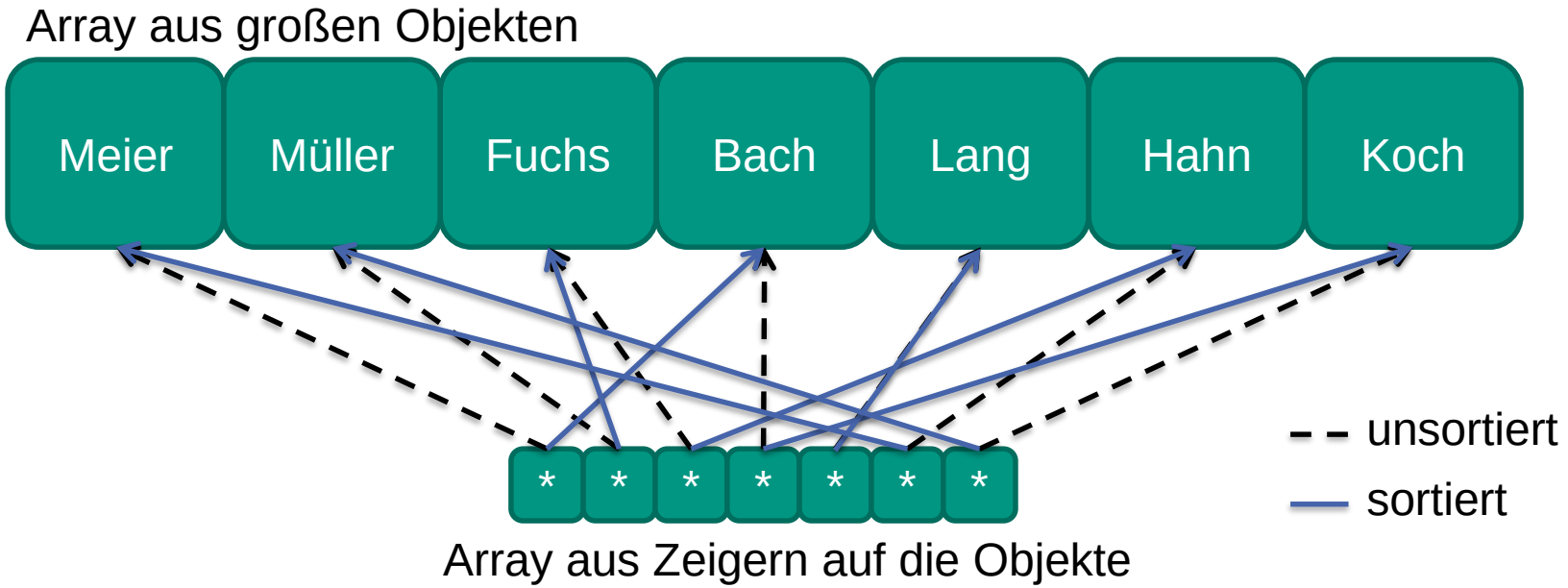
Möglichkeiten zur Anwendung von Sortieralgorithmen

Sortieren und Zeigern

- Problemstellung
  - Wie kann ich große Objekte effizient und schnell sortieren?
  - Große Objekte = viele Attribute, wobei nach einem bestimmten Attribut aufsteigend oder absteigend sortiert werden soll
- Lösung: Sortieren von Zeigern auf Objekte
  - Man erstellt ein Array aus Zeigern, welche auf alle Objekte zeigen und sortiert nur diese Zeiger
- Vorteile
  - Wesentlich schnelleres sortieren bei großen Objekten, da nur Zeiger verschoben werden
  - Mit konstanten Zeiger können die Objekte vor einer ungewollten Veränderung geschützt werden (`const int* arr`)
- Nachteile
  - Es wird zusätzlicher Speicherplatz für die Zeiger benötigt
  - Komplexer in der Programmierung

Sortieren und Zeiger

Prinzip



- Nur Veränderung der Zeiger, anstatt die großen Objekte im Array zu verschieben
- Für einzelne dynamisch erzeugte Objekte auf dem Heap ist kein anderes Sortierverfahren anwendbar

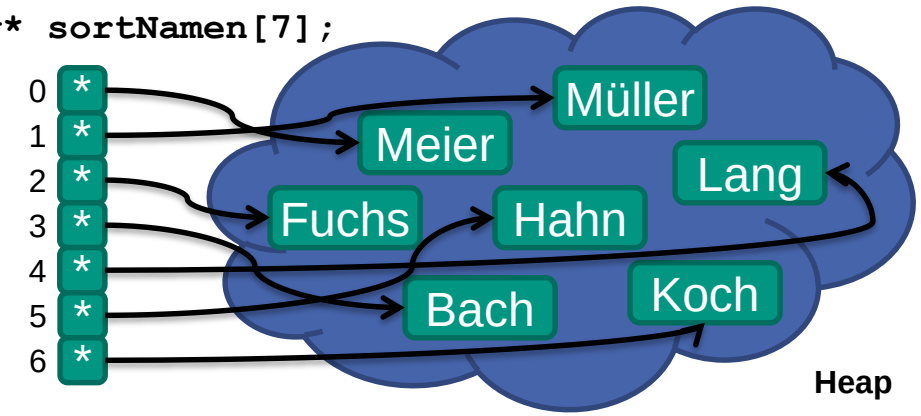
Sortieren und Zeiger

Beispiel

```
#include <iostream>
#include <iomanip>
#include <string>

using namespace std;

class NamenSortieren {
private:
  string* sortNamen[7];
  int laenge;

public:
  void sortieren();
  void erzeugen();
  void ausgeben();
};
```

```
void NamenSortieren::erzeugen() {
  sortNamen[0] = new string( "Meier" );
  sortNamen[1] = new string( "Mueller"
   );
  sortNamen[2] = new string( "Fuchs" );
  sortNamen[3] = new string( "Bach" );
  sortNamen[4] = new string( "Lang" );
  sortNamen[5] = new string( "Hahn" );
  sortNamen[6] = new string( "Koch" );
  laenge = 7;
}
```

```
string* sortNamen[7];
```



Sortieren und Zeiger

Beispiel – Sortieren dynamischer Objekte

```
void NamenSortieren::sortieren() {
  string* temp = NULL;

  for( int i = 0; i < laenge - 1; i++ ) {
    for( int j = laenge - 1; j >= i + 1; j-- ) {
      if( *sortNamen[j] < *sortNamen[j-1] ) {
        temp = sortNamen[j];
        sortNamen[j] = sortNamen[j-1];
        sortNamen[j-1] = temp;
      }
    }
  }
}

void NamenSortieren::ausgeben() {
  cout << left;
  for( int i = 0; i < laenge; i++ )
    cout << setw( 10 ) << *sortNamen[i];

  cout << endl;
}
```

BubbleSort-Algorithmus zum Sortieren

Vergleich des Inhalts, worauf das Element im Zeigerarray zeigt

Nur verändern der Zeiger im Zeigerarray (Adressen werden vertauscht)

Linksbündig ausgeben

Ausgabe der Namen über Dereferenzierung

Sortieralgorithmen

Zwischenübung - Lsg

- Gegeben sind folgende 6 Buchstaben

Z	B	G	R	O	I

- Aufgabe:
  Buchstaben alphabetisch, nach einem der vorgestellten Algorithmen sortieren.
  Dabei jeden Zwischenschritt/Iterationsschritt angeben

Bubble Sort
Merge Sort
Insertion Sort
Quick Sort

- Lösungsziel:

B	G	I	O	R	Z

- Sortieralgorithmen
  - Bubble Sort
  - Insertion Sort
  - Merge Sort
  - Quick Sort
- Anwendung von Sortieralgorithmen
  - Sortieren und Zeiger

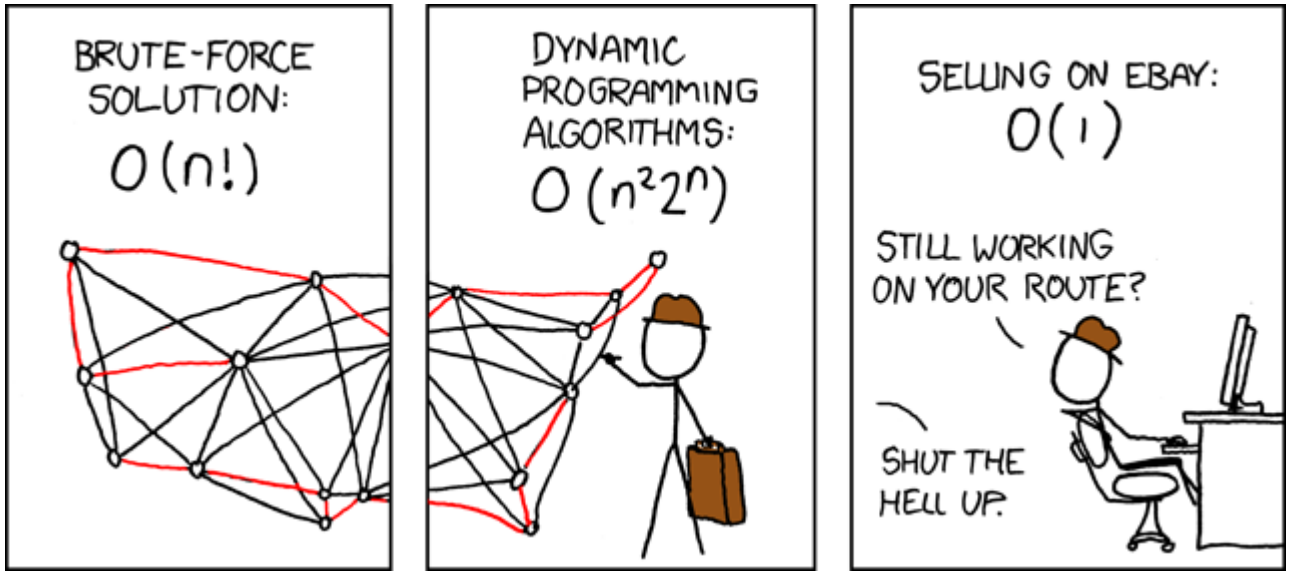
ALGORITHMEN AUF GRAPHEN

Suchalgorithmen

Suchalgorithmen und Laufzeit

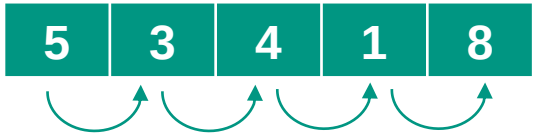
Definition:
„Ein Suchalgorithmus sucht in einer Gesamtmenge von Daten (Suchraum) nach Mustern oder Objekten mit bestimmten Eigenschaften“

- Graphen werden in verschiedenen Gebieten eingesetzt
  - Routenplanung
  - Kommunikationsnetwerke
  - Gantt-Diagramme
  - Hierarchische Strukturen
- Algorithmen auf Graphen können unterschiedliche Informationsgewinne bedeuten

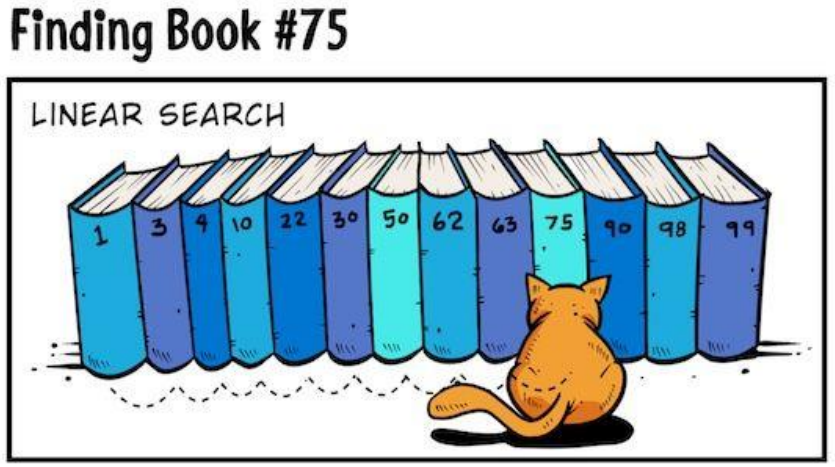


Suchalgorithmen
Lineare Suche

- Element in Liste suchen
  - Elemente nach Index in aufsteigender Reihe nach gesuchter Stelle
    „Suche Stelle, an der Element „2“ ist“



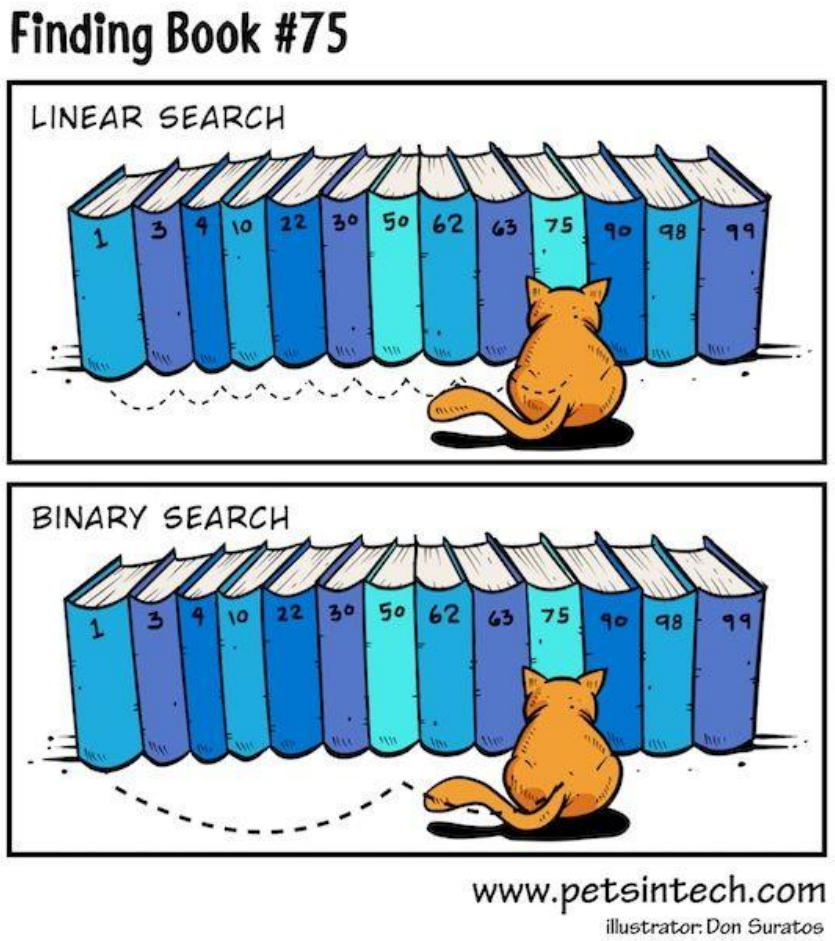
- Vorgehen: Sortieren!
  - Elemente nach Index in aufsteigender Reihe nach gesuchter Stelle, Abbruch wenn x > gesuchtes Element
    „Suche Stelle, an der Element „2“ ist“



Das muss doch auch effizienter gehen?!

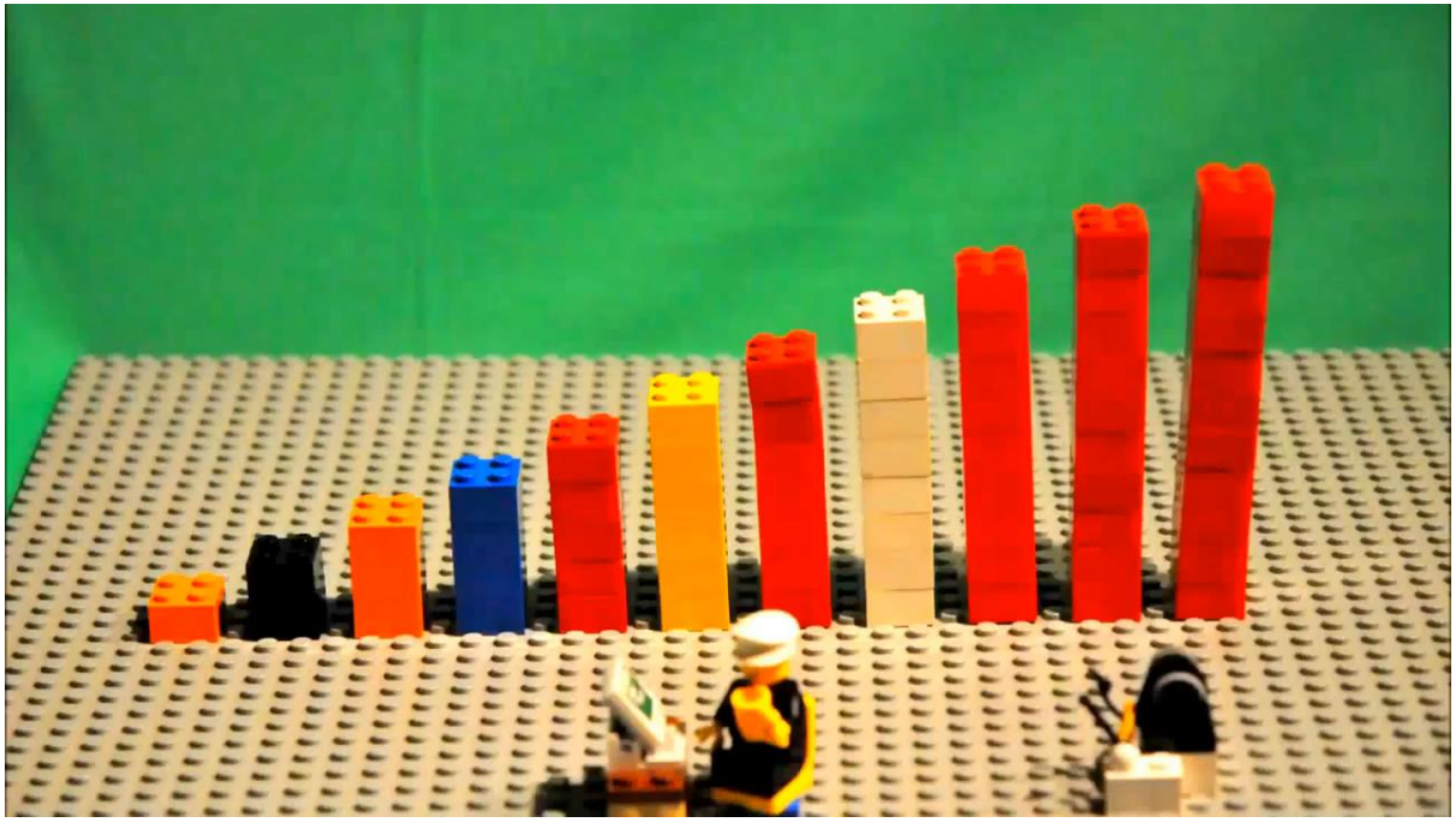
Suchalgorithmen
Binärsuche

- Vorgehen:
  - Suche beginnt immer in der Mitte des aktuell zu untersuchenden Arrays/Liste/Intervalls
  - Der Suchraum wird mit jeder Iteration halbiert
- Die Binärsuche benötigt von Anfang an sortierte Elemente




```
int binarySearch( list, key, bottom, top )
  center = ( bottom + top )/2
  if list[center] == key {
      return center }
  elseif top - bottom > 0 {
      if key < list[center]
        return binarySearch( list, key, bottom, center )
      else
        return binarySearch( list, key, center + 1, top ) }
```

Suchalgorithmen
Binärsuche



Algorithmen auf Graphen
Breitensuche

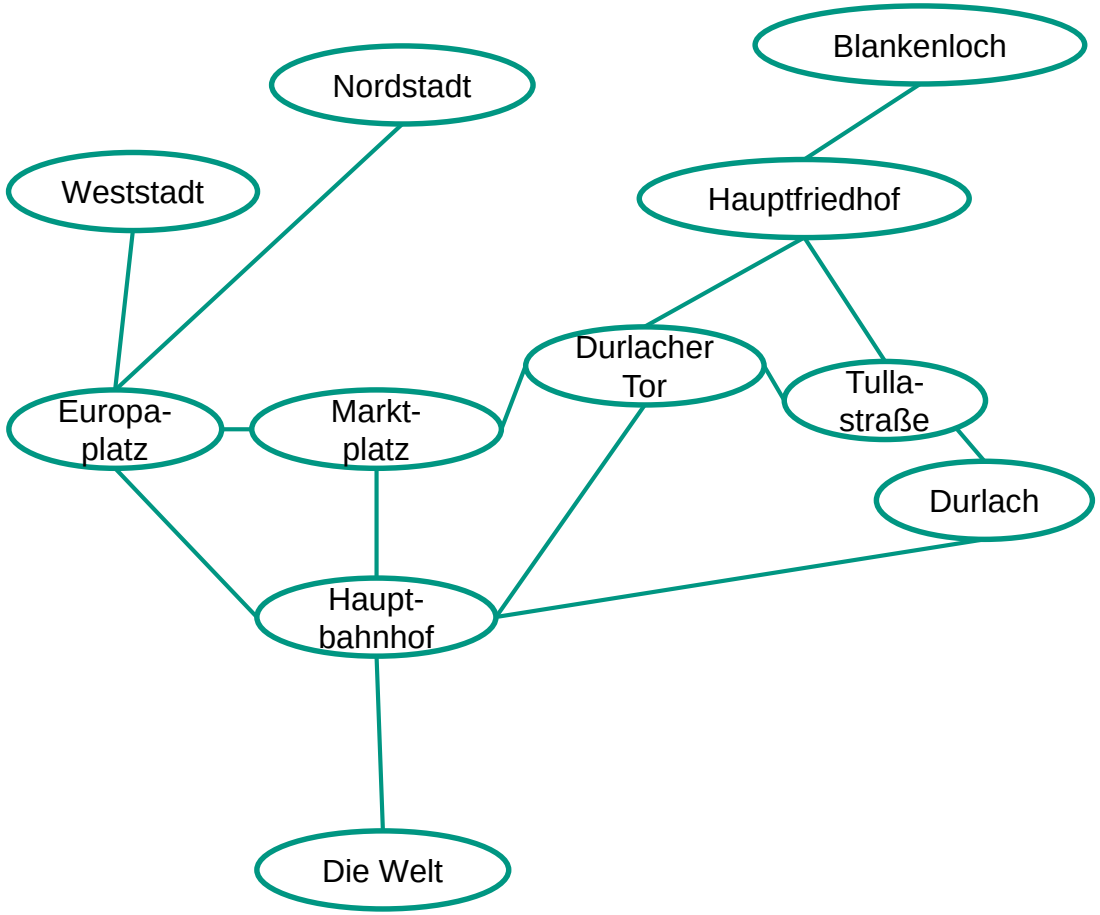
- Verfahren zum durchlaufen der einzelnen Knoten in einem Graphen
  - Durchläuft alle Knoten, welche mit dem aktuellen (z.B. Start-)Knoten verbunden sind
  - Im Gegensatz dazu durchläuft die Tiefensuche einen Pfad bis zu seinem Endknoten, bevor er zurückkehrt und den nächsten unbesuchten Pfad abläuft
- Ergebnis der Breitensuche ist ein Breitensuchbaum
  - Der Breitensuchbaum ist ein gerichteter Graph

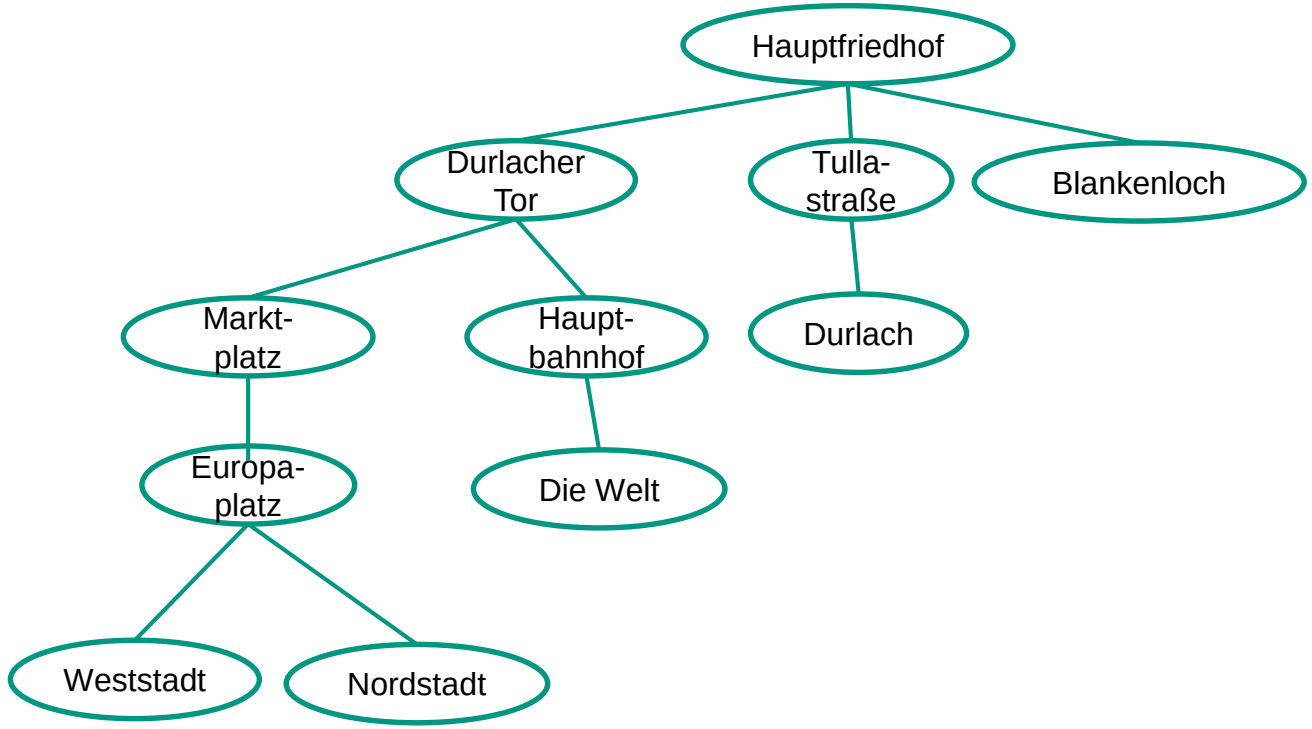
```
BreadthFirstSearch( G, s )
  for alle Knoten u in G                  // Initialisiere alle Knoten im Graph
  do farbe[u] = weiss                     // Alle Knoten wurden noch nicht gefunden
     d[u] = ∞                             // Alle Knoten haben noch keinen Abstand
     vater[u] = NIL                       // Alle Knoten haben noch keinen Vater

  create queue Q                          // Erzeuge eine Warteschlange Q
  farbe[s] = grau                         // Starte mit Startknoten s
  d[s] = 0                                // Abstand zum Startknoten d
  enqueue( Q, s )                         // Hänge s an die Warteschlange

  while Q not empty                       // Solange noch Knoten abgearbeitet werden
  do u = dequeue( Q )                     // hole den nächsten Knoten u
    for alle Knoten v aus Adj[u]          // Für alle zu u adjazente Knoten v
    do if farbe[v] == weiss               // prüfe ob v schon gefunden wurde
       then farbe[v] = grau               // wenn nicht, dann setze v auf gefunden
            d[v] = d[u] + 1               // Weise Knoten v seinen Abstand zu
            vater[v] = u                  // Der Vater von Knoten v ist der Knoten u
            enqueue( Q, v )               // Merke v für die weitere Verarbeitung
    farbe[u] = dunkelgrün                 // Markiere Knoten u als abgearbeitet
```

Algorithmen auf Graphen

Breitensuche – Karlsruher Liniennetz

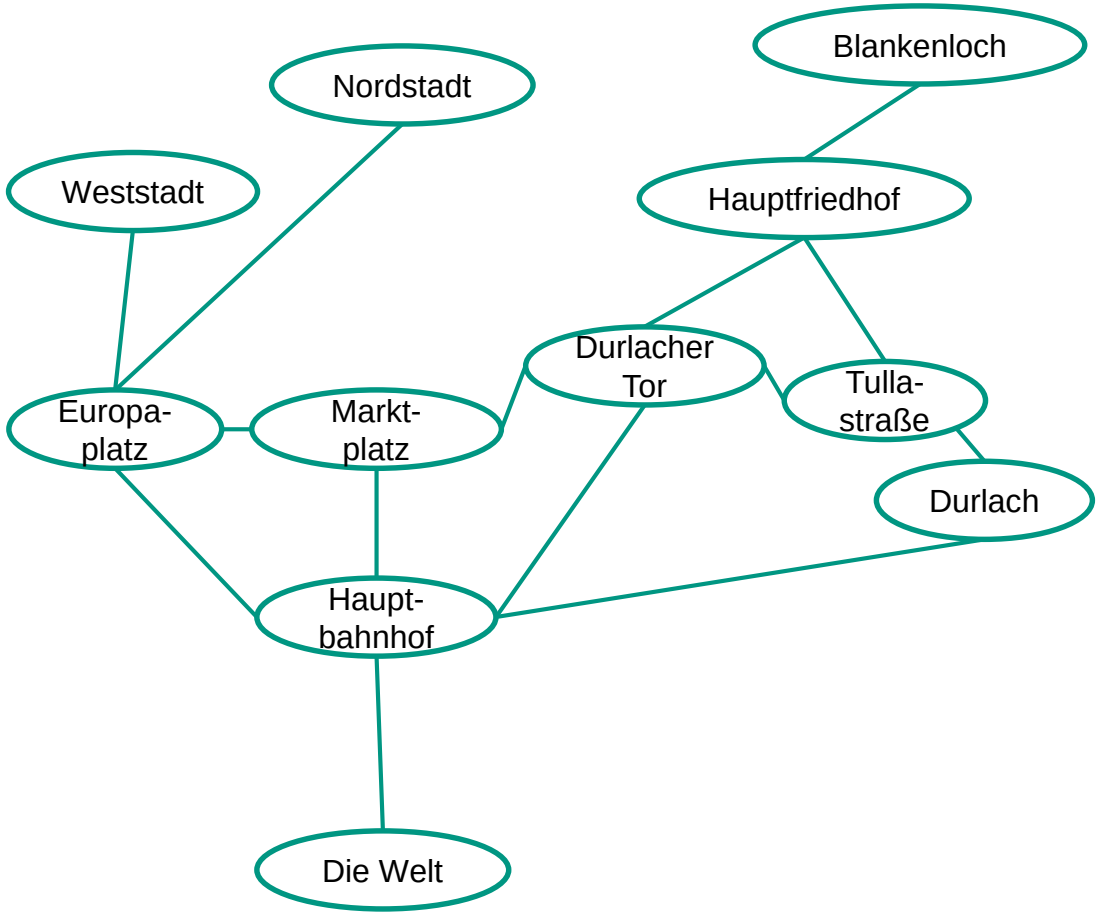
- Gesucht: Breitensuchbaum mit Start Hauptfriedhof





Breitensuche

Zwischenübung

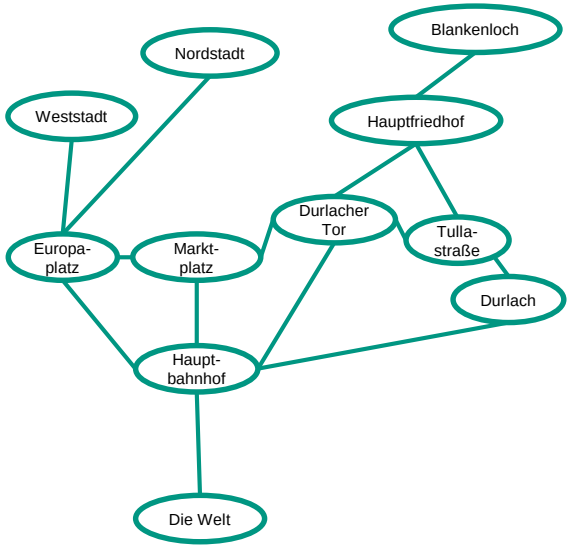


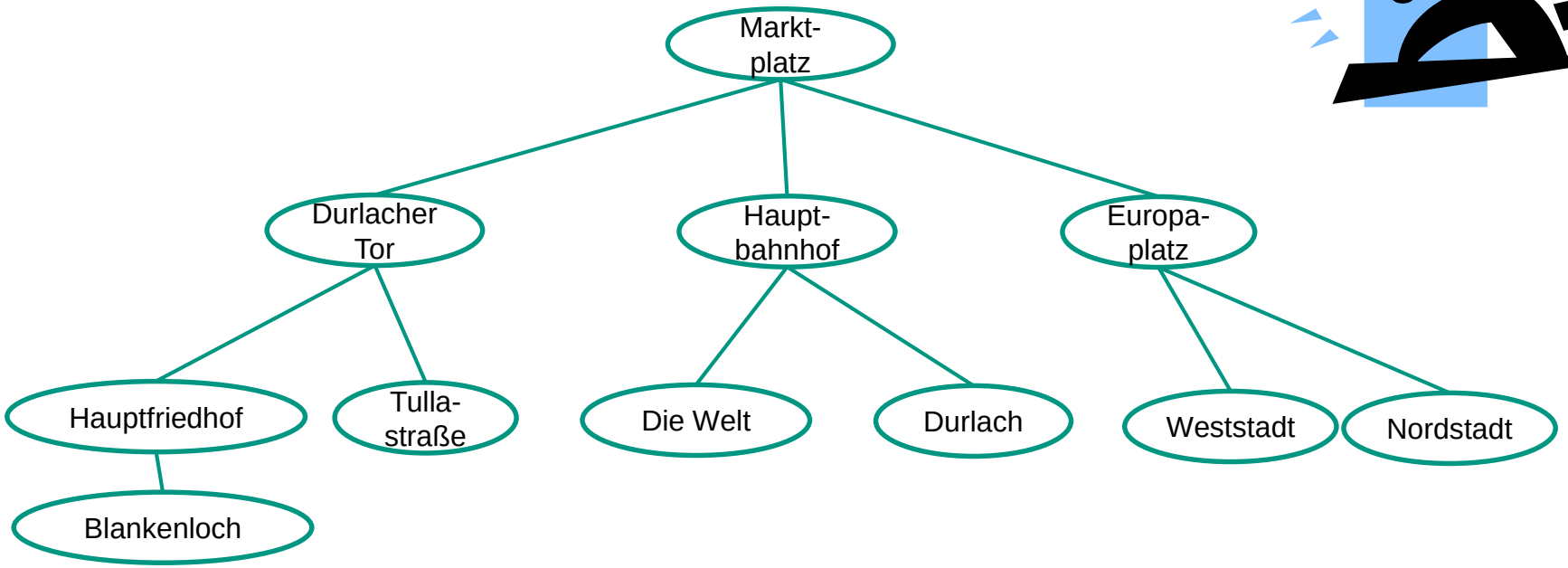
- Stellen Sie den zugehörigen Breitensuchbaum auf mit Start am Marktplatz

Breitensuche

Zwischenübung - Lsg

zugehöriger Breitensuchbaum, mit Start am Marktplatz

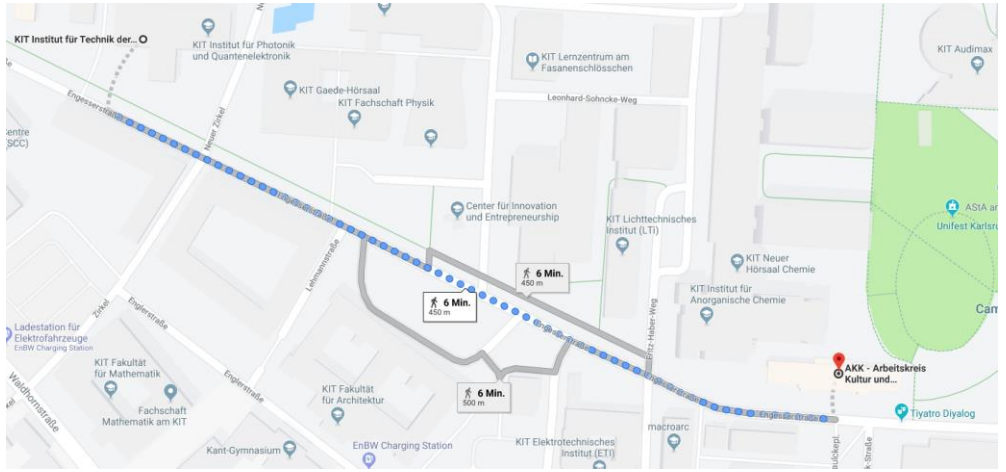




Suchalgorithmen – kürzester Pfad

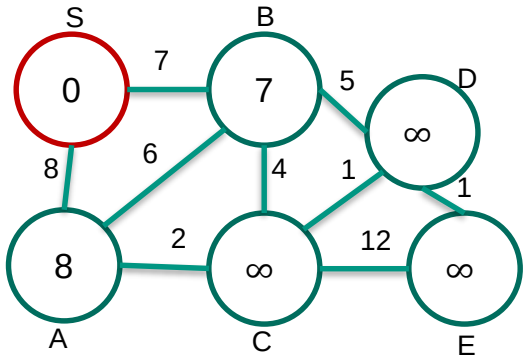
Dijkstra Algorithmus

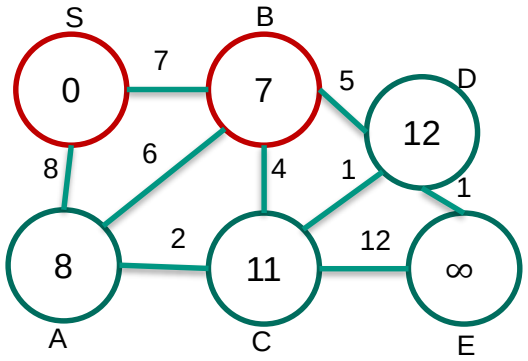
- Ziel: kürzester Weg zwischen Start- und Zielknoten finden
  - Kürzester weg (Ablauf)
  - Kleinste Wegkosten
- Vorgehen:
  - Initialisiere die Distanz im Startknoten mit 0 und in allen anderen Knoten mit ∞ (oder hier ⊥).
  - Solange es noch unbesuchte Knoten gibt, wähle darunter denjenigen mit minimaler Distanz aus und speichere, dass dieser Knoten schon besucht wurde.
  - Berechne für alle noch unbesuchten Nachbarknoten die Summe des jeweiligen Kantengewichtes und der Distanz zum aktuellen Knoten.
  - Ist dieser Wert für einen Knoten kleiner als die dort gespeicherte Distanz, aktualisiere sie und setze den aktuellen Knoten als Vorgänger.
    - Dieser Schritt wird auch als Update oder Relaxieren bezeichnet.
  - Hier: Netzwerk mit Knoten A, B, C, D, E, F, G und den jeweiligen Verbindungsgewichten

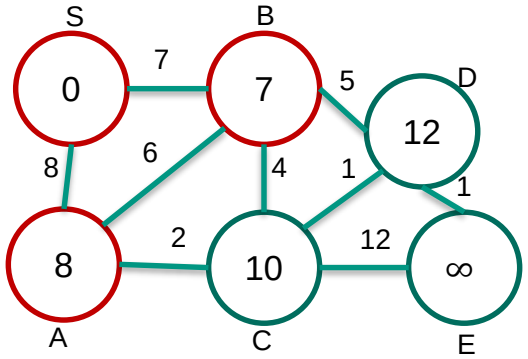


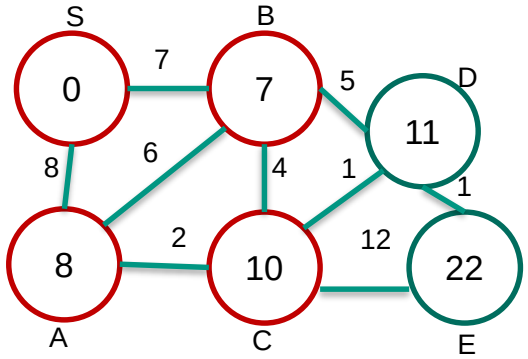
Suchalgorithmen – kürzester Pfad

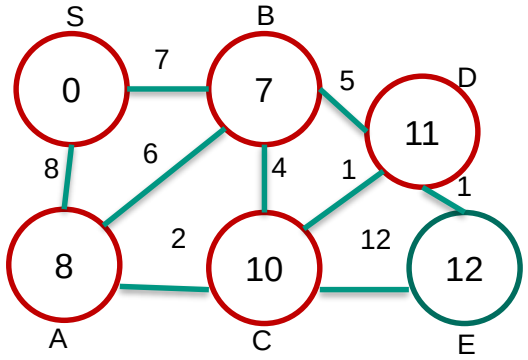
Dijkstra Algorithmus

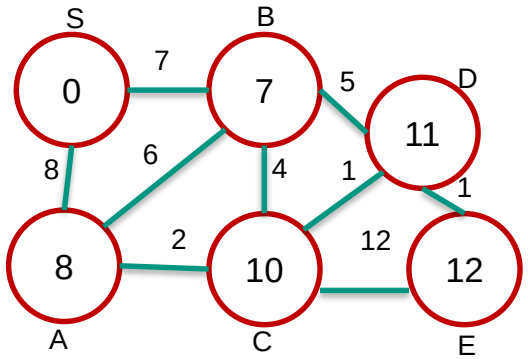












Kürzester Pfad von S nach E:
S→A→C→D→E

- Suchalgorithmen
  - Lineare Suche
  - Binäre Suche
- Algorithmen auf Graphen
  - Breitensuche
  - Dijkstra

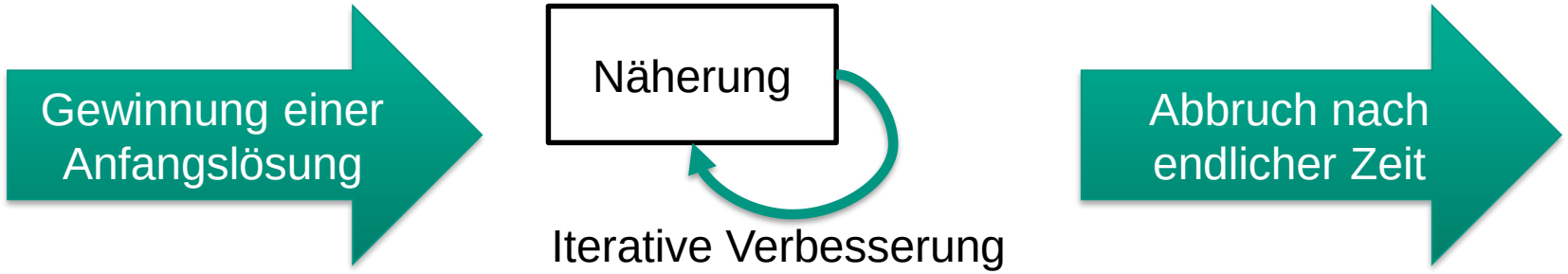
OPTIMIERUNGSALGORITHMEN

Optimierungsalgorithmen

Motivation

- Problemstellung: große Menge möglicher Lösungen, exakte Lösung nicht möglich oder sinnvoll
  - Bewertungs- / Kostenfunktion gegeben, damit jede Lösung bewertbar
  - Beispiel Parameteroptimierung: alle Parameterwerte und Kombinationen durchprobieren dauert zu lange

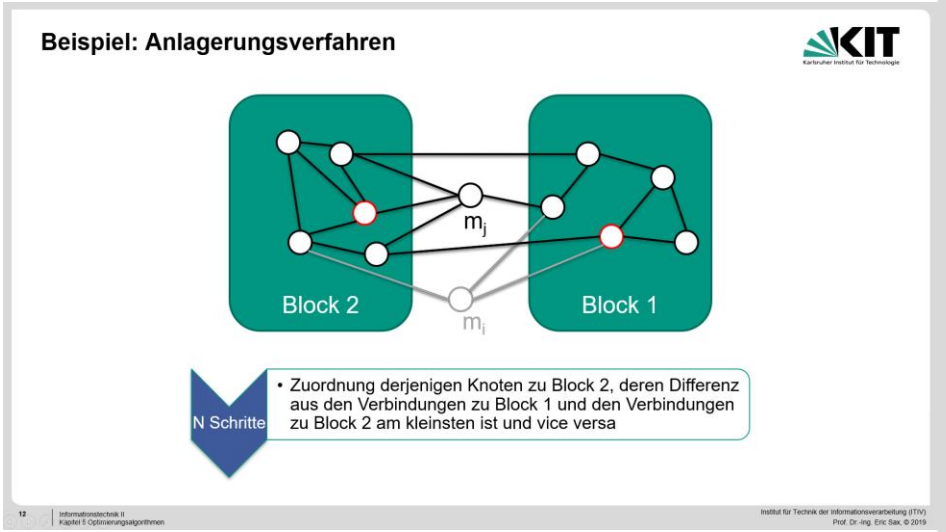
- Lösung: **Heuristische Verfahren**
  - nicht die perfekte Lösung suchen, wenn eine gute Lösung ausreichend ist



- In IT behandelte Verfahren:
  - Anlagerungsverfahren (für Anfangslösung)
  - Random Interchange
  - Kernighan-Lin
  - Greedy
  - Simulated Annealing

Optimierungsalgorithmen
Partitionierung

- Beispiel Problemstellung: Partitionierung parallelisieren um rechenintensives Programm schneller verarbeiten zu können!
  - Notwendige Kommunikation zwischen Recheneinheiten möglichst klein halten
  - Rechenlast gleichmäßig verteilen
- Umformulierung für Graphentheorie:
  Rechenaufgaben → Knoten
  Kommunikation zwischen Recheneinheiten→ Kanten
  - Knoten gleichmäßig auf Partitionen aufteilen
  - Möglichst wenig Kanten zwischen den Partitionen



- Mögliche Aufgabe: Knoten auf zwei Partitionen aufteilen (Bipartition)
  Welchem Softwaremodul (Partition) muss ich meine Funktion (Knoten) zuordnen?
  - Ziel: Kommunikation zwischen den Modulen A und B minimieren
  - Lösung durch Optimierung der Kostenfunktion = Summe der Kantengewichte zwischen A und B

Optimierungsalgorithmen

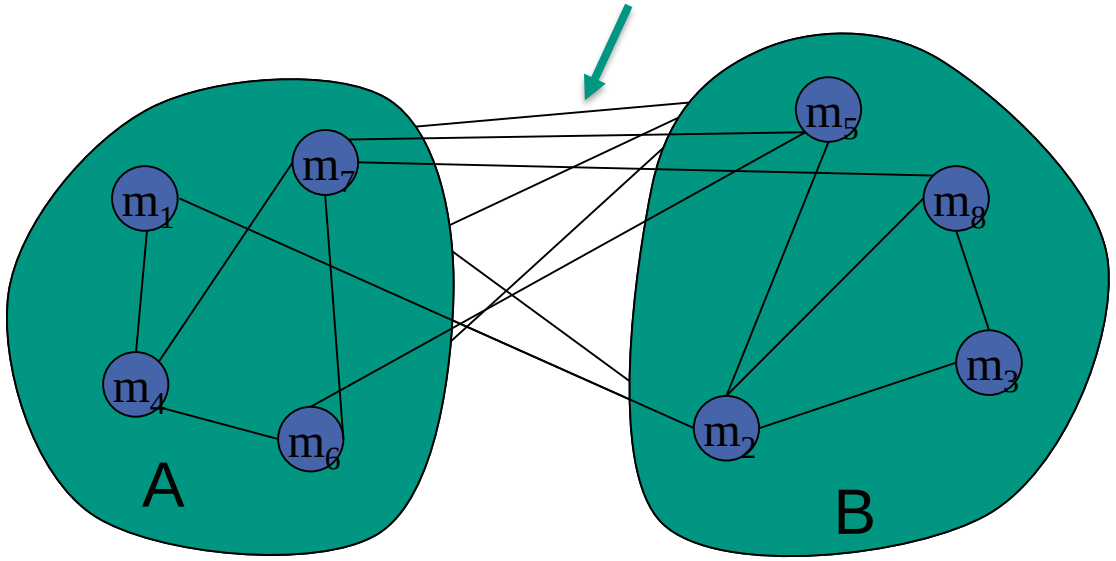
Partitionierung - Kostenfunktion

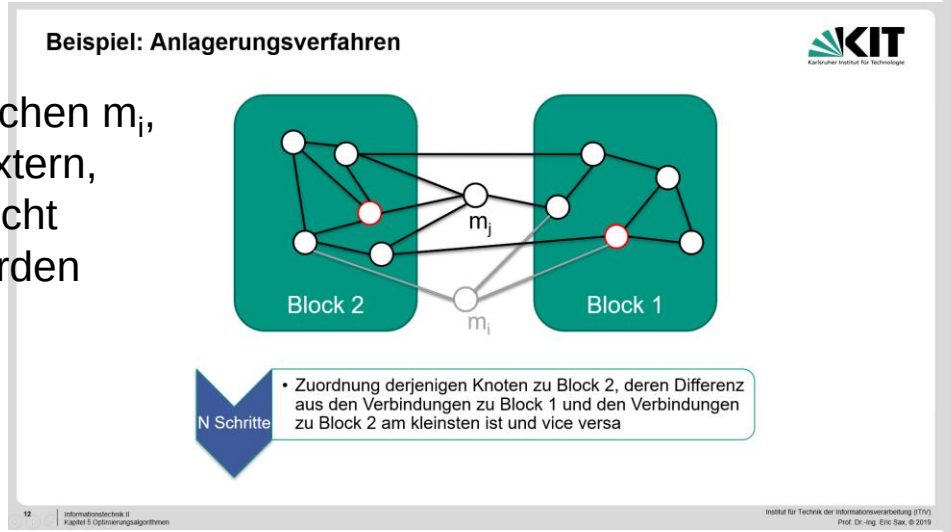
- Optimierung durch Vertauschen von immer zwei Knoten (m_i, m_j) aus unterschiedlichen Partitionen
- Kostenveränderung durch Tausch von (m_i, m_j):

$$D(m_i,m_j) = E(m_i) - I(m_i) + E(m_j) - I(m_j) - 2\,c_{ij}$$

Externe Kosten — Interne Kosten

Kante zwischen m_i, m_j bleibt extern, darf also nicht gezählt werden



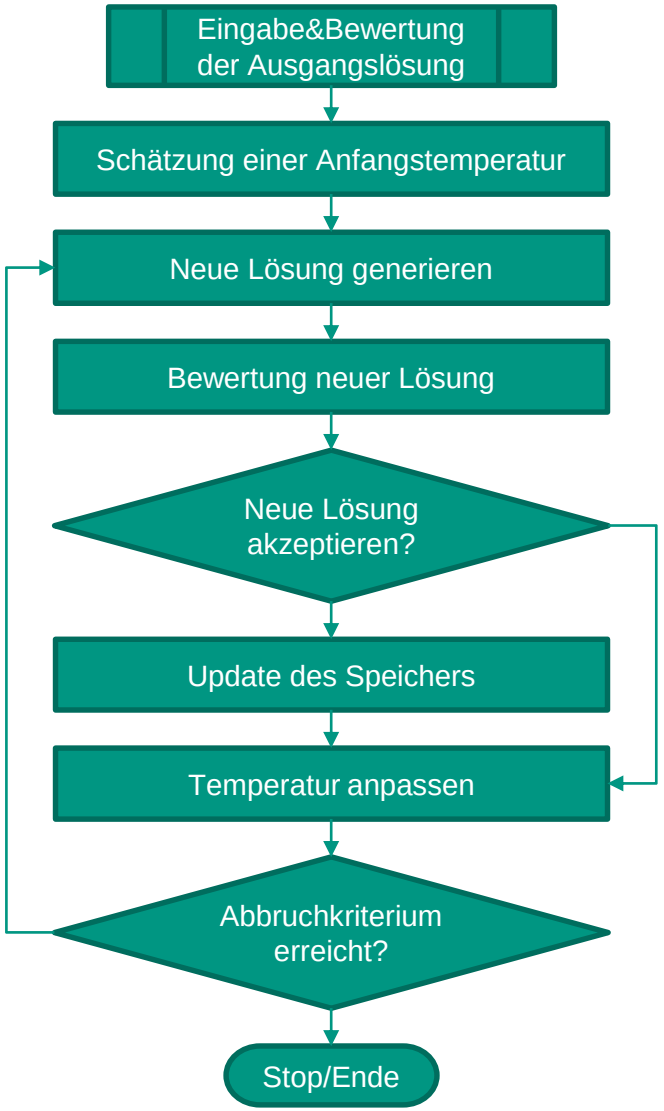


Optimierungsalgorithmen
Greedy Algorithmus

- greedy = “gierig”, “gefräßig”
- Greedy Prinzip:
  - Lösung eines Problems durch schrittweise Erweiterung der Lösung ausgehend von Startlösung
  - in jedem Schritt wähle den bestmöglichen Schritt (ohne Berücksichtigung zukünftiger Schritte) → greedy
  - gefundene Lösung muss nicht immer optimal sein!
    Drei Möglichkeiten
    - Wir erhalten die optimale Gesamtlösung
    - Wir erhalten eine Lösung, welche zwar nicht optimal ist, aber vom Optimum nur wenig abweicht
    - Die berechnete Lösung ist (beliebig) schlecht

Treffe zu jedem Verfahrensschritt diejenige Entscheidung, die in diesem Moment am besten ist!

Optimierungsalgorithmen
Simulated Annealing



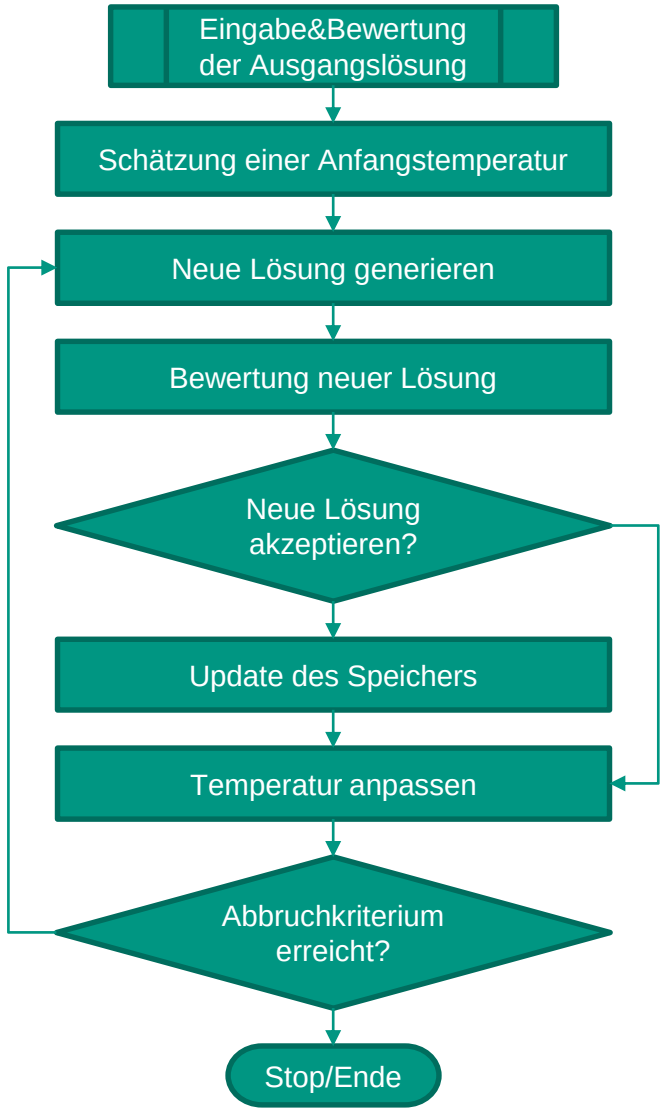
- Die äußere Schleife wird solange iteriert, bis ein vordefiniertes Abbruchkriterium erfüllt ist.
- In jedem Durchlauf der äußeren Schleife wird der Wert einer Temperatur T, ausgehend von einer Starttemperatur kontinuierlich reduziert. In einer inneren Schleife wird ein neuer Zustand s_{new} generiert und mit einer Kostenfunktion c(s) bewertet.

Pseudocode:

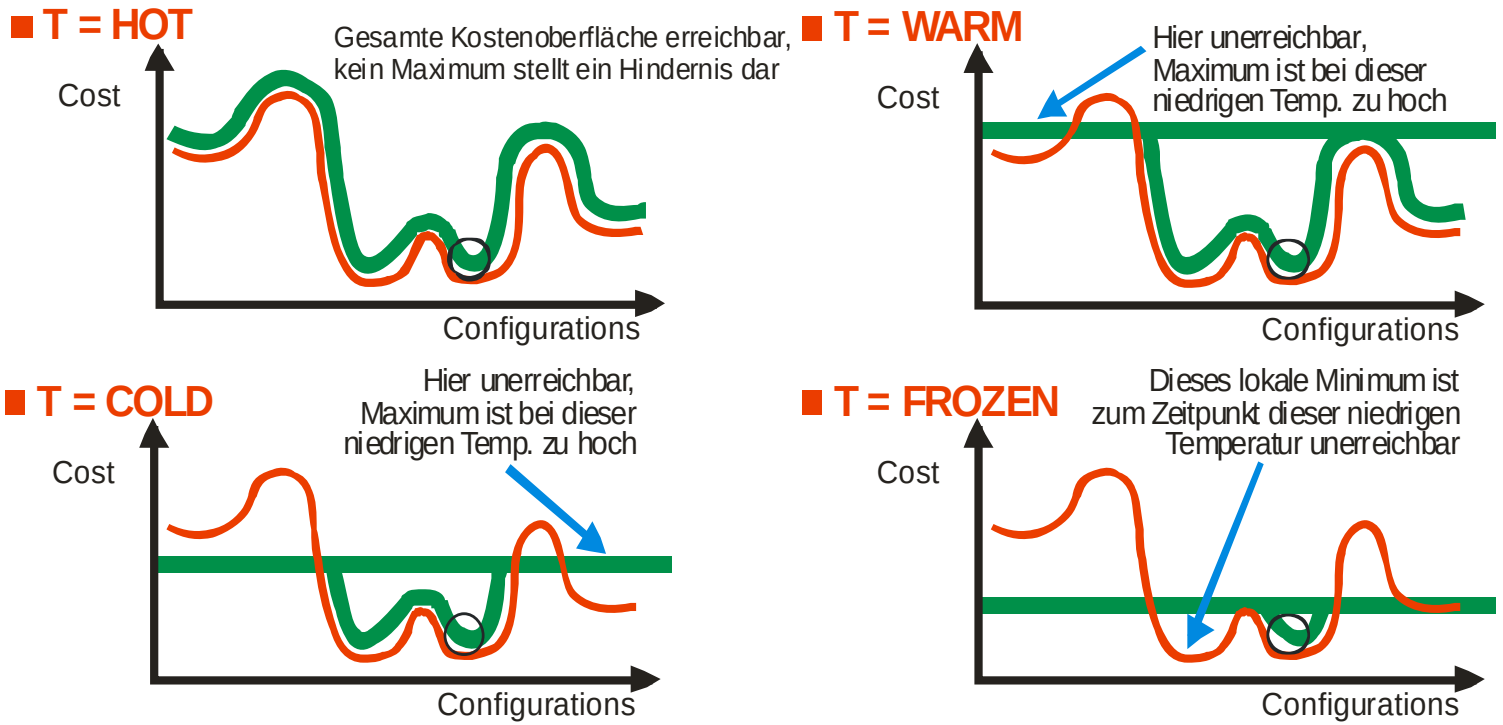
```
procedure simulated_annealing(S,c)
begin
    s:=start_state(S)
    T:=start_temperature(S)
    repeat
        while not stationary_state(S,T) do
            s_new:=random_modify(s)

            if random([0,1]) ≤                then
                s:=s_new
        end
        T:=update(T)
    until convergence
end
```

Optimierungsalgorithmen

Simulated Annealing



- Erreichbarkeit benachbarter bzw. entfernter Lösungen in Abhängigkeit von der Temperatur T
- Hill-Climbing Eigenschaft



- Optimierungsalgorithmen
  - Partitionierung
  - Greedy Algorithmus
  - Simulated Annealing

Ziele der heutigen Übung

- Nach der heutigen Übung können Sie….

- … bekannte Sortier-, Such- und Optimierungsalgorithmen gegenüberstellen und demonstrieren

1. … verschiedene Sortieralgorithmen, sowie deren Merkmale benennen
2. … verschiedene Sortieralgorithmen demonstrieren
3. … verschiedene Suchalgorithmen, sowie deren Merkmale benennen
4. … verschiedene Suchalgorithmen demonstrieren
5. … verschiedene Optimierungsalgorithmen, sowie deren Merkmale benennen
6. … verschiedene Optimierungsalgorithmen demonstrieren