

Übung 2

Übung zu Informationstechnik II und Automatisierungstechnik – Nathalie Brenner

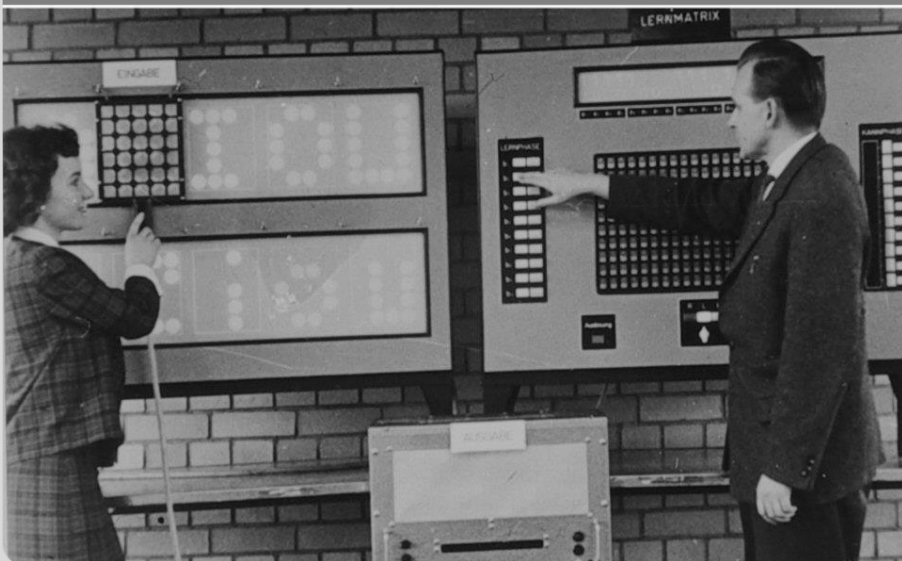
Prof. Dr.-Ing. Eric Sax

Institutsleitung

Prof. Dr.-Ing. J. Becker

Prof. Dr.-Ing. E. Sax

Prof. Dr. rer. nat. W. Stork



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
Fintheit	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

■ Laufzeitanalyse durch O-Notation

Asymptotische Laufzeitkomplexität	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 <i>logarithmus dualis</i>), sehr günstig und daher erstrebenswert, z.B. <i>binäre Suche</i> (siehe Kapitel 6)
$O(n)$	linear	Verdoppelung der Problemgröße bewirkt Verdoppelung der Laufzeit, immer noch zufrieden stellend, z.B. <i>sequenzielle Suche</i> (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 <i>Quicksort</i> (siehe Kapitel 7)
$O(n^2)$	quadratisch	Verdoppelung der Problemgröße bewirkt Vervielfachung der Laufzeit, ungünstig, z.B. schlechte Sortierverfahren wie <i>Bubblesort</i> (siehe Kapitel 7)
$O(n^3)$	kubisch	Verdoppelung der Problemgröße bewirkt Verachtfachung der Laufzeit, sehr unbefriedigend, z.B. einfache <i>Matrizenmultiplikation</i>
$O(k^n)$	exponentiell	Verdoppelung der Problemgröße bedeutet Quadrierung (weil $k^{2n} = (k^n)^2$) der Laufzeit, katastrophal, z.B. <i>Backtracking</i> - oder <i>Exhaustionsalgorithmen</i> (siehe Kapitel 9)

Rechengesetze zur O-Notation:

- $c * n \rightarrow O(n)$
- $O(c * n) \rightarrow O(n)$
- $c * O(n) \rightarrow O(n)$
- $O(O(n)) \rightarrow O(n)$
- $O(n_1) + O(n_2) + \dots + O(n_x) \rightarrow O(\max(n_i))$
- $O(n) * O(m) \rightarrow O(n * m)$
- $m = O(n) \rightarrow O(n + m) = O(n)$

- $n^a \rightarrow O(n^b)$, wenn $a < b$
- $\log_a n \rightarrow O(\log_b n)$, $\forall a, b > 1$

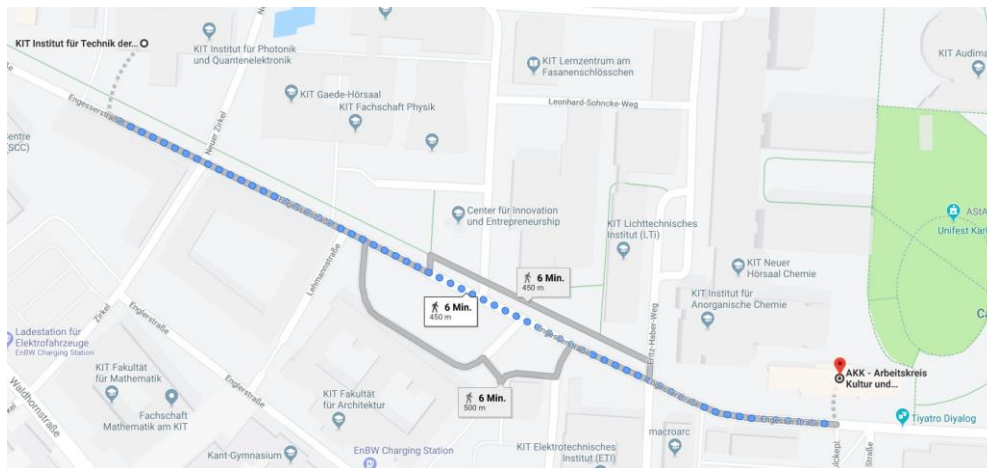
INHALT ÜBUNG 2



Sortier-, Such- und Optimierungsalgorithmen

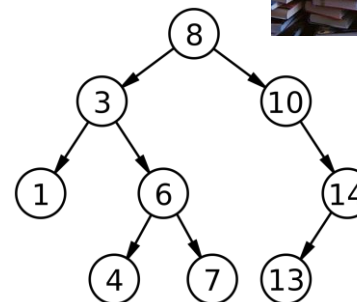
Wozu braucht man diese Algorithmen?

- 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“



Tiefensuche

Breitensuche



Bubble Sort ✓

Merge Sort

Insertion Sort ✓

Quick Sort



Ziele der heutigen Übung



■ Nach der heutigen Übung können Sie....

- ... bekannte Sortier- und Suchalgorithmen 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

SORTIERALGORITHMEN



Sortieralgorithmen

Wozu braucht man diese Algorithmen?

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

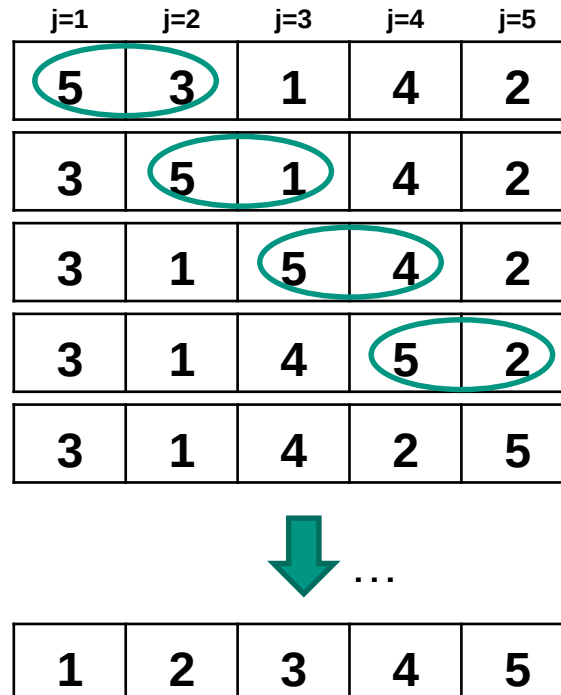


Sortieralgorithmen

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
}
```

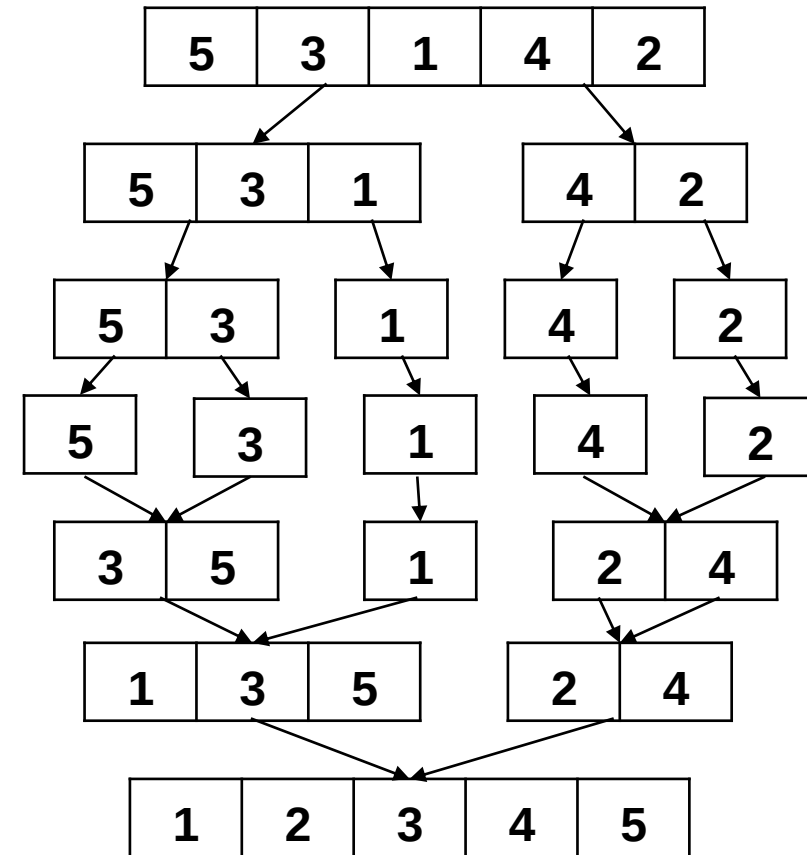
InsertionSort

```
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
```

Sortialgorithmen

Merge Sort

- Stabiler Sortialgorithmus
- 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



Sortialgorithmen

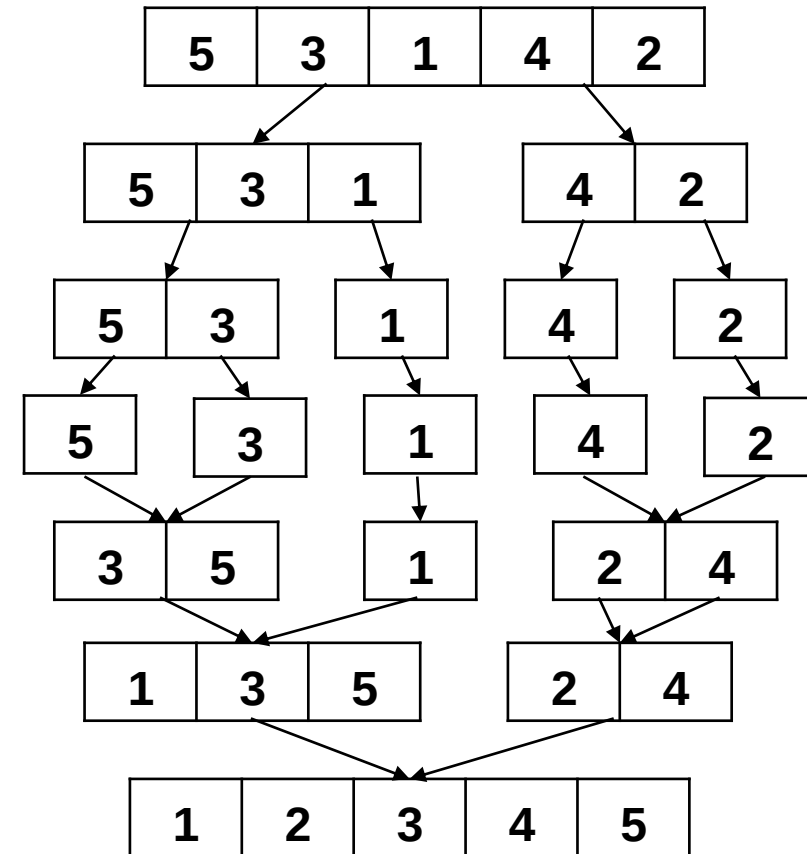
Merge Sort

```
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)
```

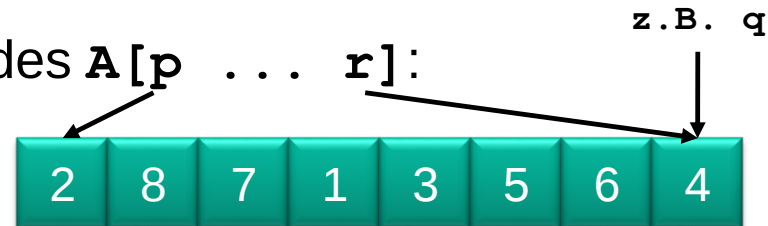
```
Merge(liste1,liste2)
```

```
  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
```



- 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

- Drei Schritte für das Sortieren eines Feldes $A[p \dots r]$:



- 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 $A[q+1..r]$.)
- 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.
- 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.

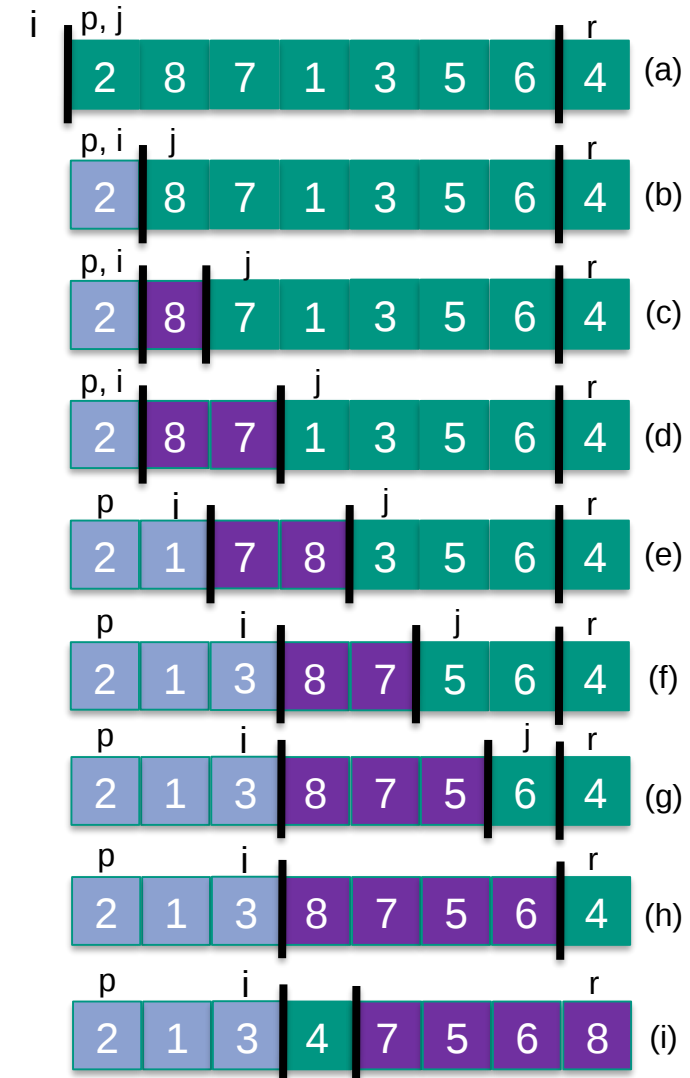
Sortieralgorithmen

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



■ 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

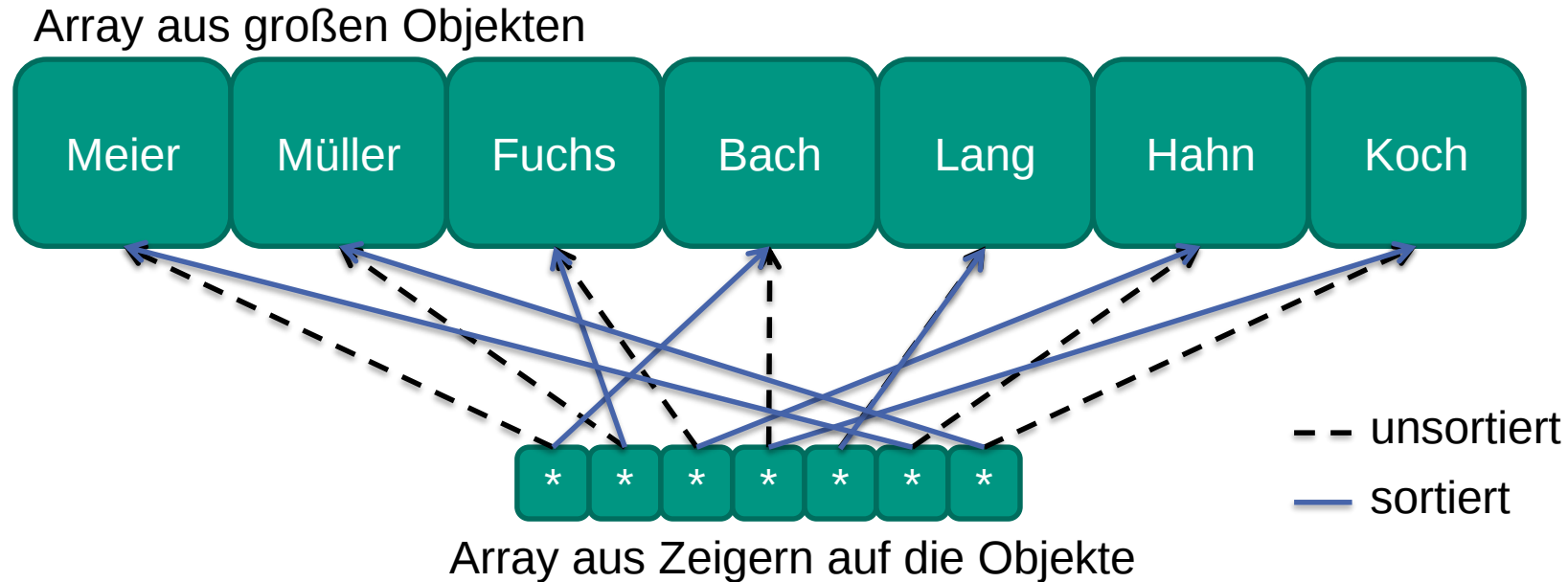
- Wesentlich schnelleres sortieren bei großen Objekten, da nur Zeiger verschoben werden
- Mit konstanten Zeigern können die Objekte vor einer ungewollten Veränderung geschützt werden (`const int* arr`)

■ 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

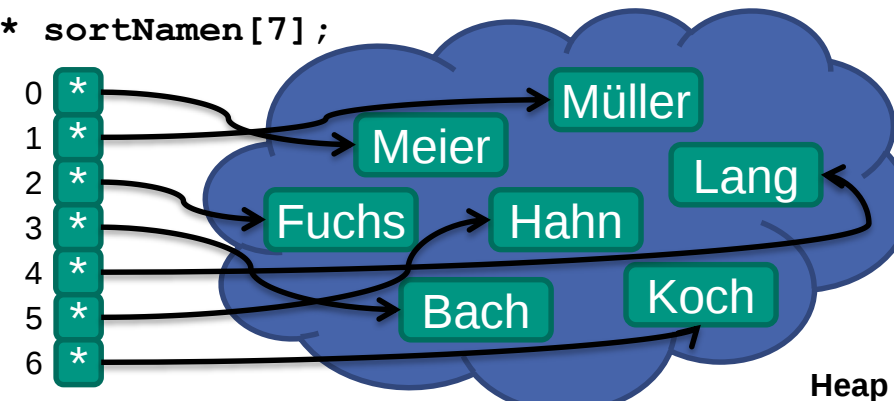
```
#include <iostream>
#include <iomanip>
#include <string>

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;  
            }  
        }  
    }  
}
```

BubbleSort-Algorithmus
zum Sortieren

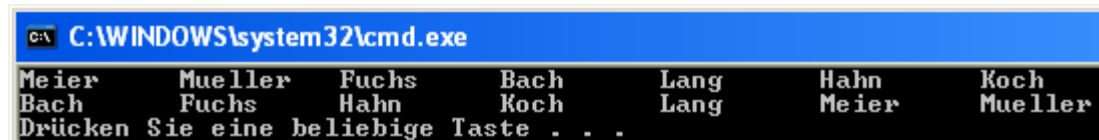
Vergleich des Inhalts, worauf das
Element im Zeigerarray zeigt

Nur verändern der Zeiger im
Zeigerarray (Adressen werden
vertauscht)

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

Linksbündig ausgeben

Ausgabe der Namen über
Dereferenzierung



```
C:\WINDOWS\system32\cmd.exe  
Meier    Mueller  Fuchs   Bach    Lang    Hahn    Koch  
Bach     Fuchs   Hahn    Koch    Lang    Meier   Mueller  
Drücken Sie eine beliebige Taste . . .
```

Sortialgorithmen

Zwischenübung

- Gegeben sind folgende 6 Buchstaben

E	C	F	A	D	B
---	---	---	---	---	---

- Aufgabe:
Buchstaben mit dem Merge/Quick Sort alphabetisch sortieren.
Dabei jeden Zwischenschritt/Iterationsschritt angeben

- Lösungsziel:

A	B	C	D	E	F
---	---	---	---	---	---



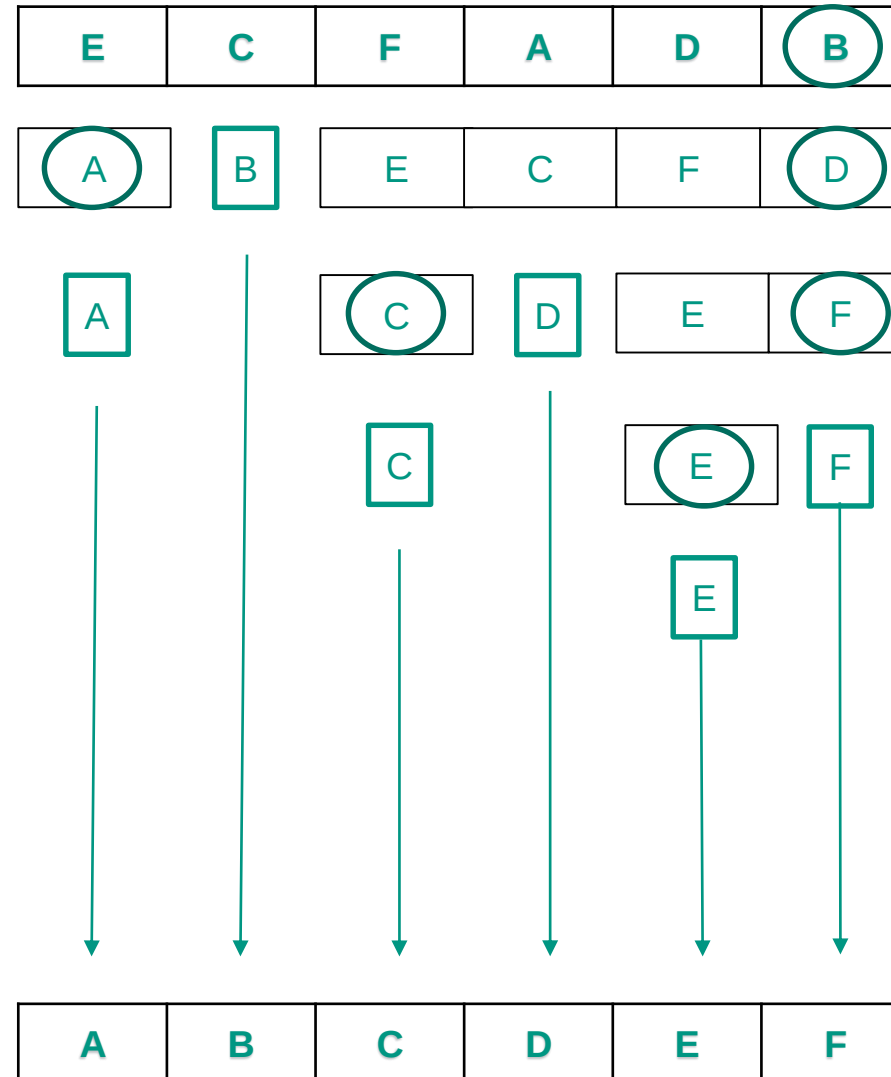
Sortialgorithmen

Zwischenübung - Lsg

Bubble Sort

E	C	F	A	D	B
C	E	F	A	D	B
C	E	F	A	D	B
C	E	A	F	D	B
C	E	A	D	F	B
C	E	A	D	B	F
C	E	A	D	B	F
C	A	E	D	B	F
C	A	D	E	B	F
C	A	D	B	E	F
A	C	D	B	E	F
A	C	D	B	E	F
A	C	B	D	E	F
A	C	B	D	E	F
A	B	C	D	E	F

Quick Sort



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



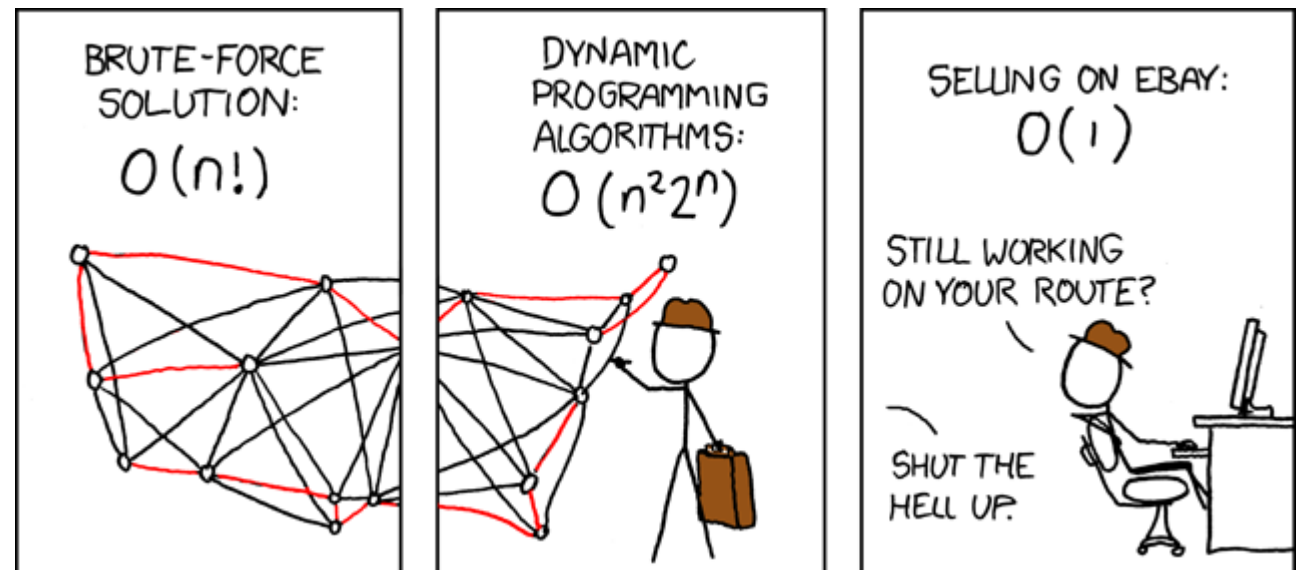
ALGORITHMEN AUF GRAPHEN



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

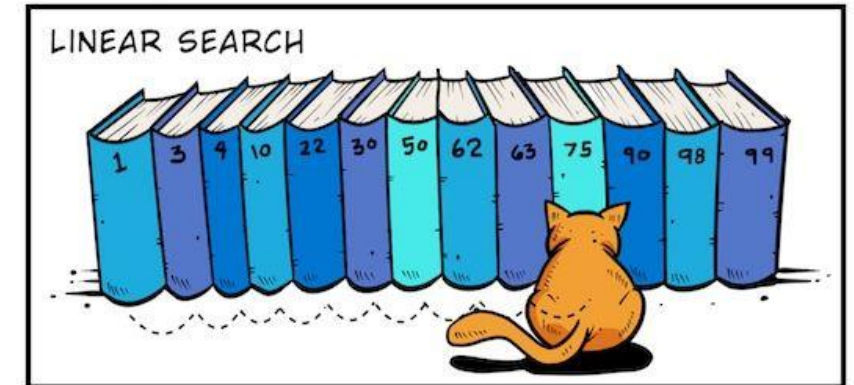


- Element in Liste suchen
 - Elemente nach Index in aufsteigender Reihe nach gesuchter Stelle
„Suche Stelle, an der Element „2“ ist“



```
int lineare_suche( list, key)
    int I;
    for (i=0;i<länge(list); i++){
        if (list(i)==key) {
            return=I;
        }
    }
end
```

Finding Book #75



Suchalgorithmen

Lineare Suche

■ Element in Liste suchen

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

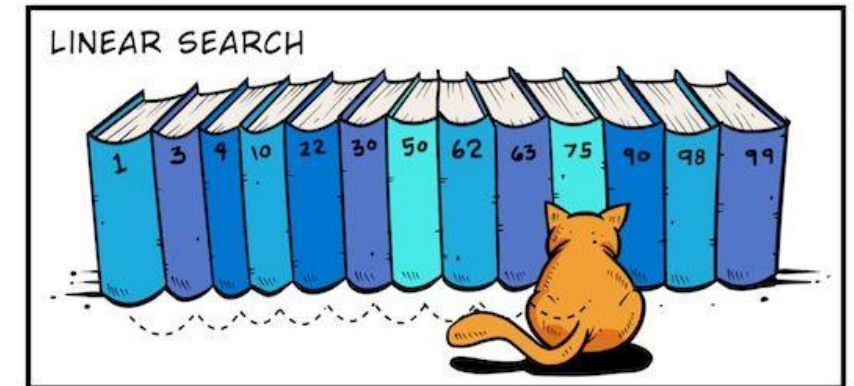


■ Vorgehen: Sortieren!

- Elemente nach Index in aufsteigender Reihe nach gesuchter Stelle, Abbruch wenn $x > \text{gesuchtes Element}$
„Suche Stelle, an der Element „2“ ist“



Finding Book #75

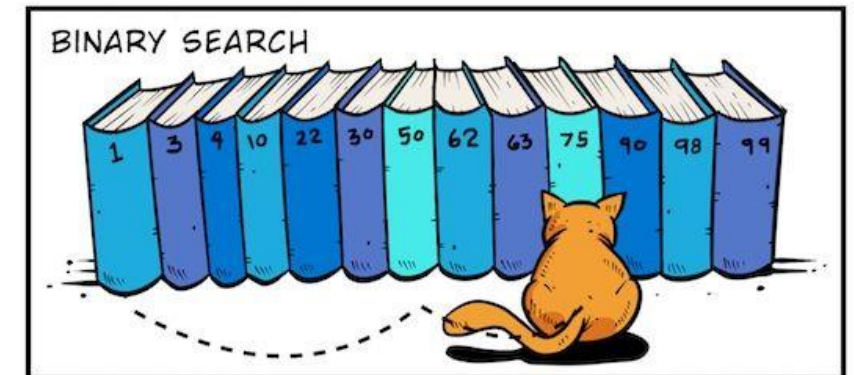
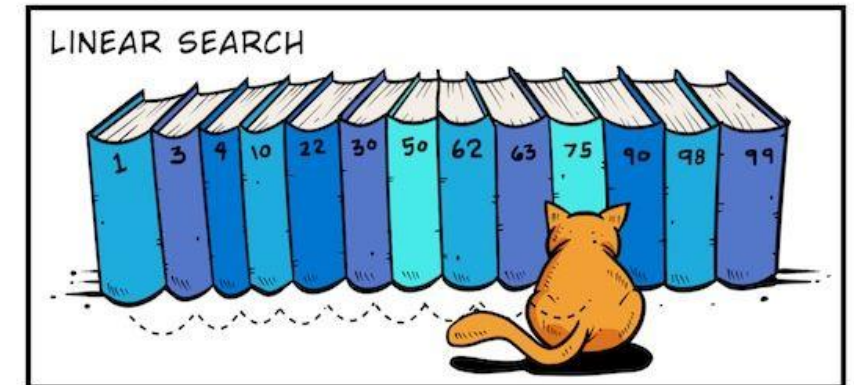


Das muss doch auch
effizienter gehen?!

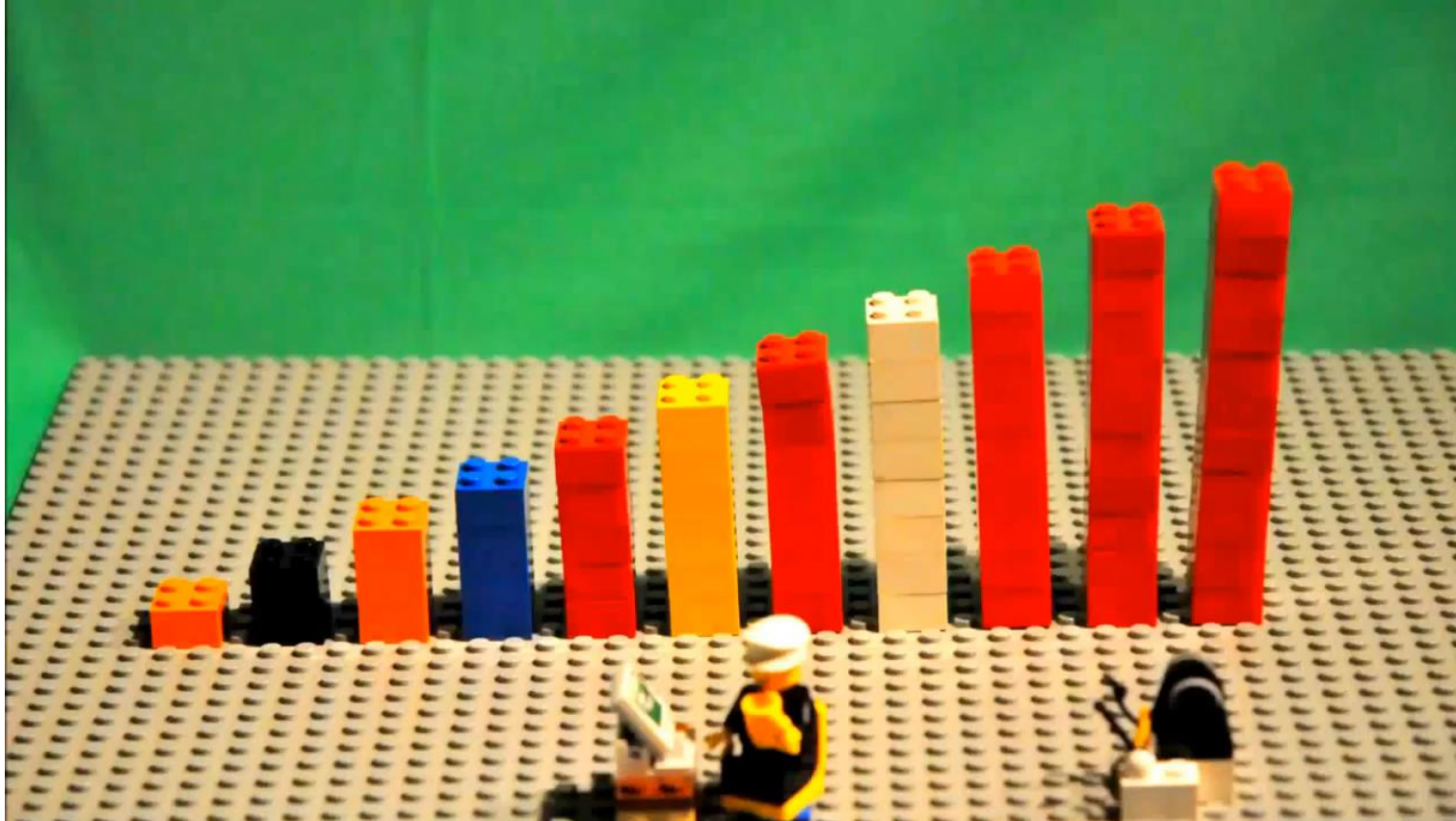
- Vorgehen:
 - Suche beginnt immer in der Mitte des aktuell zu untersuchenden Arrays/Liste/Intervalls
 - Der Suchraum wird mit jeder Iteration halbiert
- Die Binärsuche benötigt von Anfang an sortierte Elemente

```
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 ) }
```

Finding Book #75



www.petsintech.com
illustrator: Don Suratos



- 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
do farbe[u] = weiss
  d[u] =  $\infty$ 
  vater[u] = NIL

create queue Q
farbe[s] = grau
d[s] = 0
enqueue( Q, s )

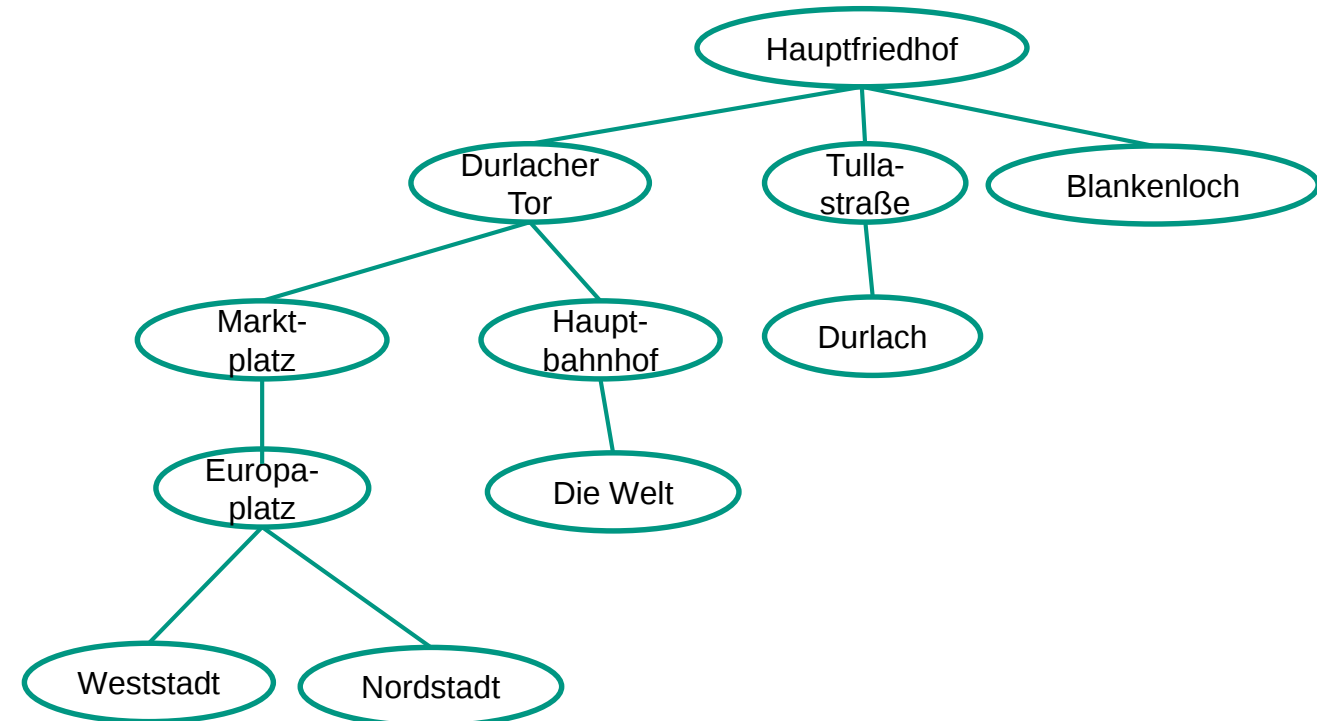
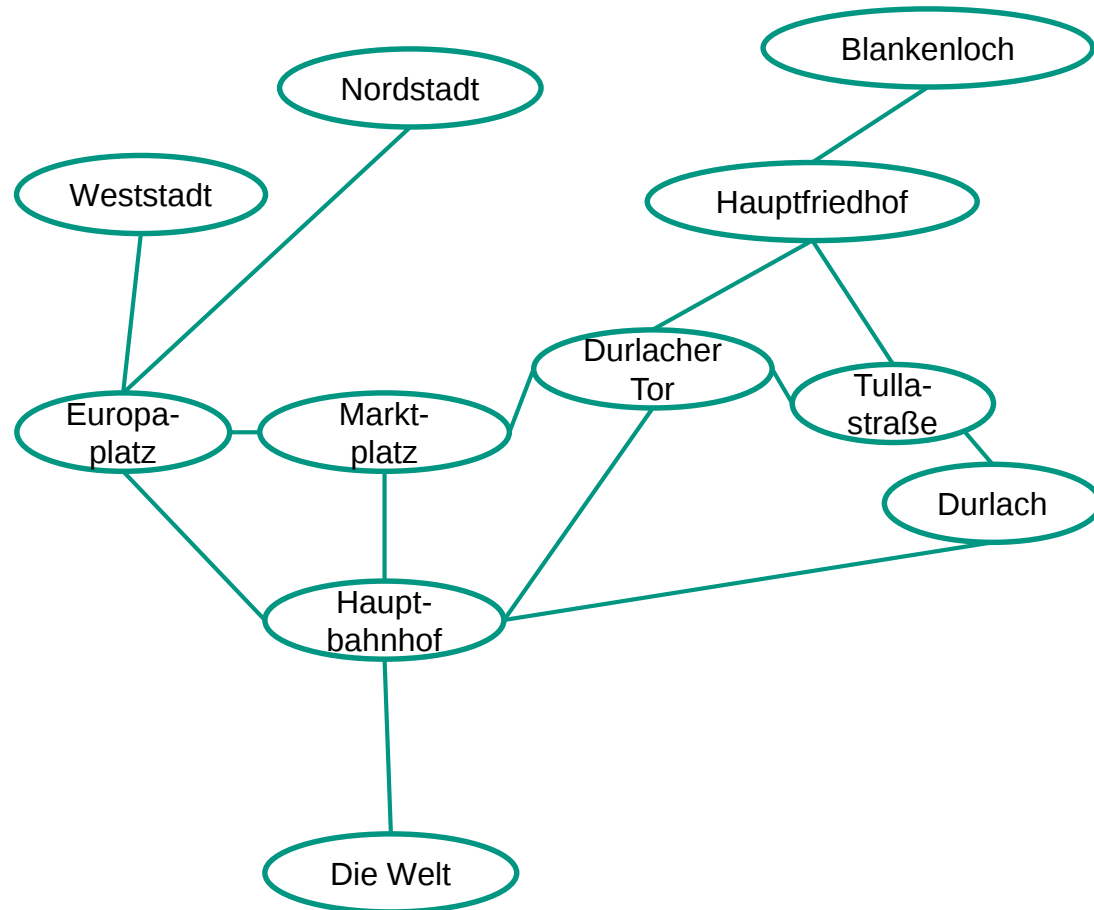
while Q not empty
do u = dequeue( Q )
  for alle Knoten v aus Adj[u]
  do if farbe[v] == weiss
    then farbe[v] = grau
      d[v] = d[u] + 1
      vater[v] = u
      enqueue( Q, v )
  farbe[u] = dunkelgrün

// Initialisiere alle Knoten im Graph
// Alle Knoten wurden noch nicht gefunden
// Alle Knoten haben noch keinen Abstand
// Alle Knoten haben noch keinen Vater

// Erzeuge eine Warteschlange Q
// Starte mit Startknoten s
// Abstand zum Startknoten d
// Hänge s an die Warteschlange

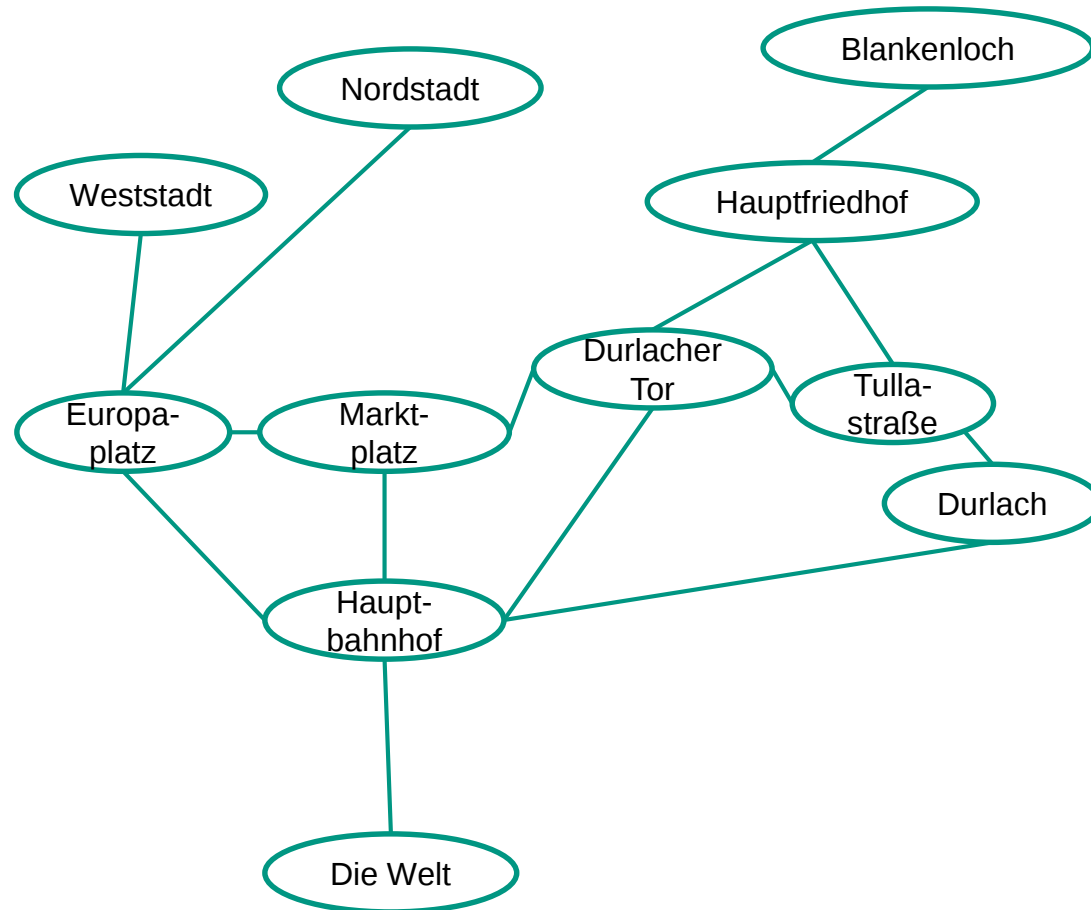
// Solange noch Knoten abgearbeitet werden
// hole den nächsten Knoten u
// Für alle zu u adjazente Knoten v
// prüfe ob v schon gefunden wurde
// wenn nicht, dann setze v auf gefunden
// Weise Knoten v seinen Abstand zu
// Der Vater von Knoten v ist der Knoten u
// Merke v für die weitere Verarbeitung
// Markiere Knoten u als abgearbeitet
```

- Gesucht: Breitensuchbaum mit Start Hauptfriedhof



Breitensuche

Zwischenübung

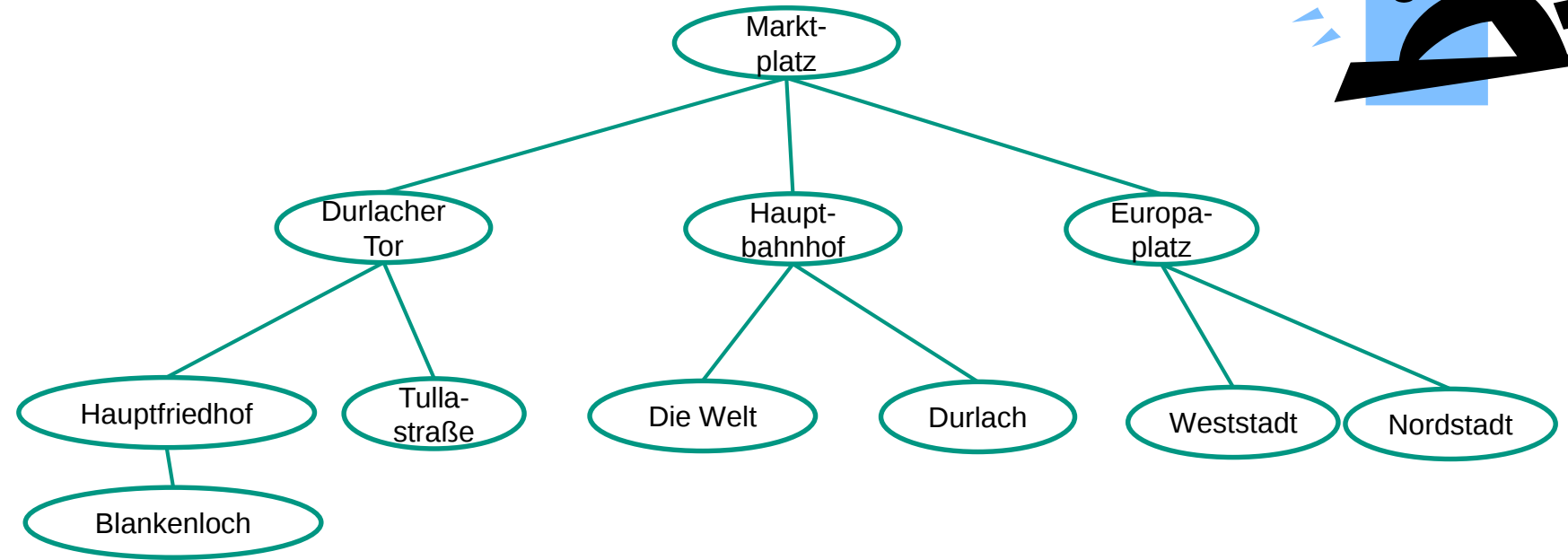
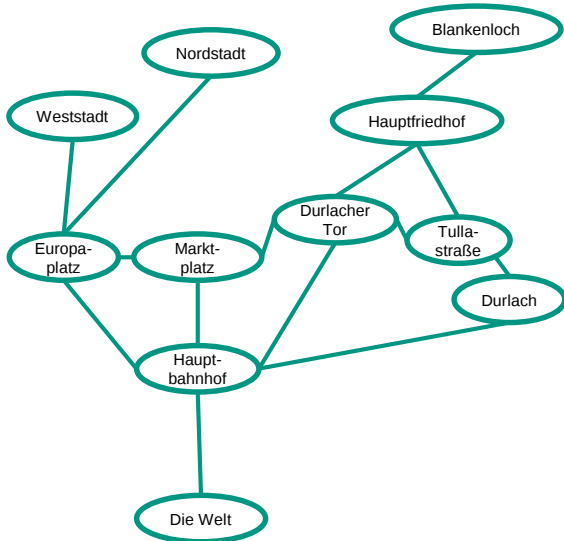


- Stellen Sie den zugehörigen Breitensuchbaum auf mit Start am Marktplatz

Breitensuche

Zwischenübung - Lsg

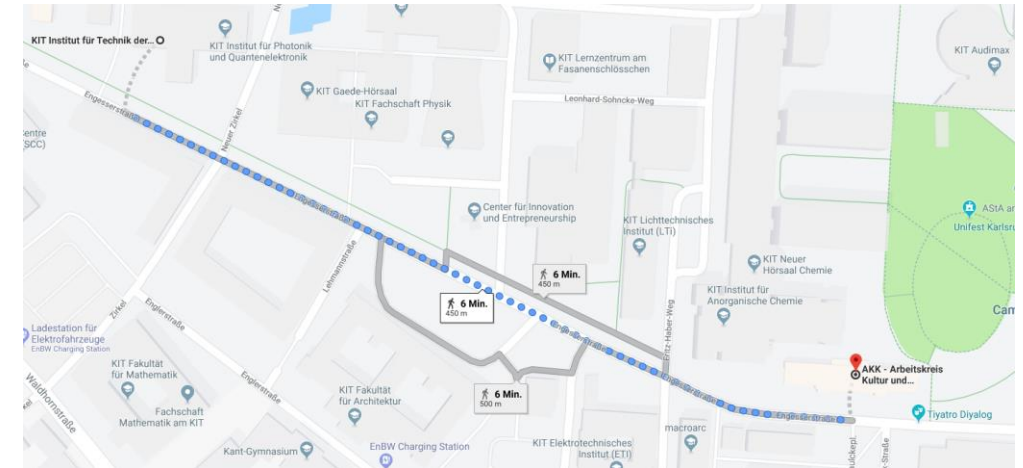
zugehöriger Breitensuchbaum, mit Start am Marktplatz



Suchalgorithmen – kürzester Pfad

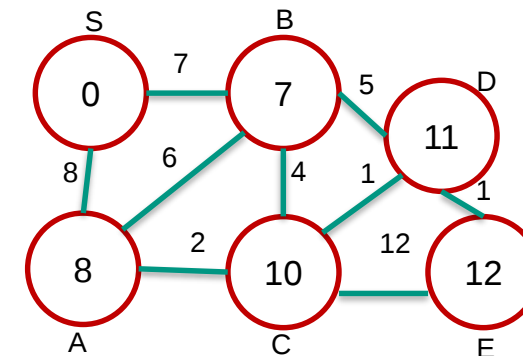
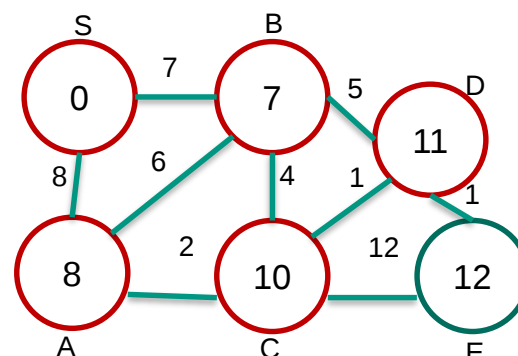
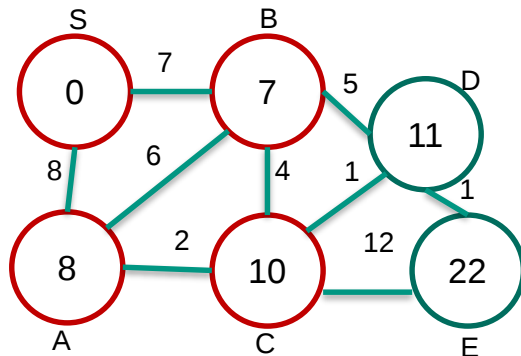
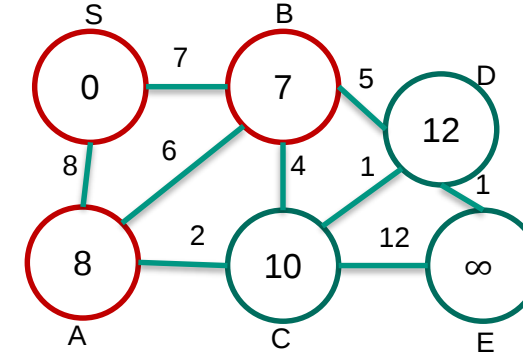
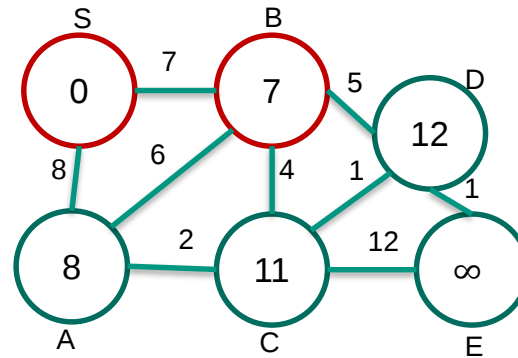
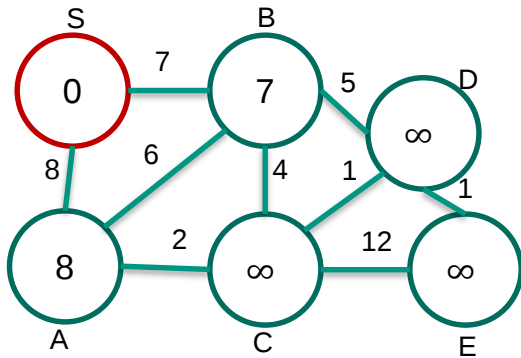
Dijkstra Algorithmus

- 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 $\perp$).
 - 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:
 $S \rightarrow A \rightarrow C \rightarrow D \rightarrow E$

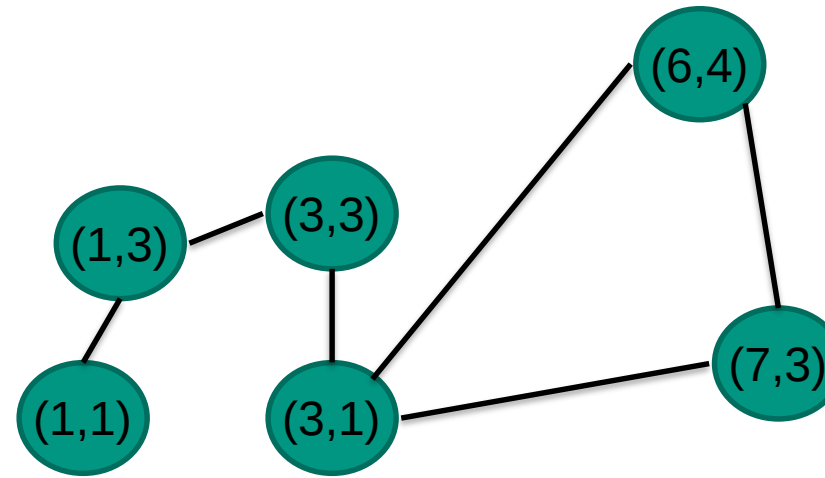
- Suchalgorithmen
 - Lineare Suche
 - Binäre Suche
- Algorithmen auf Graphen
 - Breitensuche
 - Dijkstra



Zwischenübung

Graphen und Heuristik

- Gegeben:
Graph G mit Position (x,y) für jeden Knoten



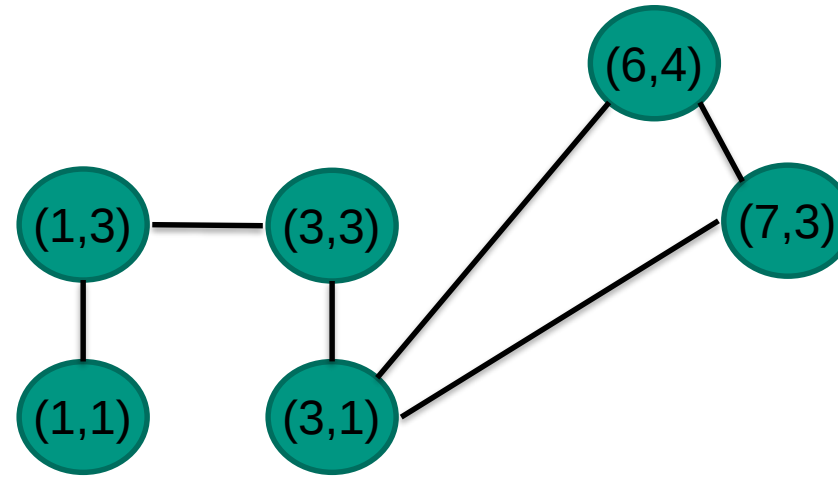
- Aufgabe:
Berechnung des Heuristikwerts bezogen auf Startknoten (1,3)
Heuristikfunktion h für zwei Knoten i und j : (auch bekannt als Manhattan-Distanz-Funktion)
$$h(i,j) = \left\| x_i - x_j \right\| + \left\| y_i - y_j \right\|$$



Zwischenübung

Graphen und Heuristik -Lsg

- Gegeben:
Graph G mit Position (x,y) für jeden Knoten



- Aufgabe:
Berechnung des Heuristikwerts bezogen auf Startknoten (1,3)
Heuristikfunktion h für zwei Knoten i und j : (auch bekannt als Manhattan-Distanz-Funktion)

$$h(i,j) = \|x_i - x_j\| + \|y_i - y_j\|$$

Zielknoten	Heuristikwert (bezogen auf (1,3))	
(1,1)	2	$h = \ 1 - 1\ + \ 3 - 1\ = 0 + 2 = 2$
(3,3)	2	$h = \ 1 - 3\ + \ 3 - 3\ = 2 + 0 = 2$
(3,1)	4	
(6,4)	6	
(7,3)	6	$h = \ 1 - 7\ + \ 3 - 3\ = 6 + 0 = 6$

Ziele der heutigen Übung



■ Nach der heutigen Übung können Sie....

- ... bekannte Sortier- und Suchalgorithmen 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