Karlsruher Institut für Technologie

Institut für Technik der Informationsverarbeitung

Institutsleitung
Prof. Dr.-Ing. K. D. Müller-Glaser
Prof. Dr.-Ing. J. Becker
Prof. Dr. rer. nat. W. Stork

Übung07: Informationstechnik (IT)

Tobias Schwalb & Timo Sandmann & Stephan Werner

Institut für Technik der Informationsverarbeitung (ITIV)

Teil2: Sortier- und Suchalgorithmen, Laufzeitanalyse

KIT – Universität des Landes Baden-Württemberg und
nationales Forschungszentrum in der Helmholtz-Gemeinschaft

www.kit.edu

Inhalt: Übung07

Karlsruher Institut für Technologie

1. Besprechung Übungsaufgaben 6.01-6.04
2. Aufg. 7.01 - Verständnisfragen
3. Aufg. 7.02 - Arrays und Sortieren
4. Aufg. 7.03 - Laufzeitanalyse von Insertion Sort
5. Aufg. 7.04 - Tiefensuche

Aufg. 7.01: Verständnisfragen Lsg. (1)

a) Ein Algorithmus ist eine genau definierte Handlungsvorschrift zur Lösung eines Problems oder einer bestimmten Art von Problemen in endlich vielen Schritten.

b) Dynamische Finitheit besagt, dass das Verfahren zu jedem Zeitpunkt nur endlich viel Speicherplatz benötigen darf.

c) Komplexitätstheorie bezeichnet das Verhalten von Algorithmen bezüglich Ressourcenbedarf, wie Rechenzeit und Speicherbedarf.

d) Berechenbarkeitstheorie besagt, dass:

   i. der Algorithmus bei denselben Voraussetzungen das gleiche Ergebnis liefern muss. Falsch

   ii. jeder Schritt des Verfahrens tatsächlich ausführbar sein muss. Falsch

   iii. das Verhalten bezüglich der Terminierung (ob der Algorithmus überhaupt jemals erfolgreich beendet werden kann) erfüllt sein soll. Richtig

Aufg. 7.01: Verständnisfragen Lsg. (2)

e) Ein Algorithmus kann mit Pseudo Code oder Nassi-Shneiderman Diagrammen oder Ablaufdiagrammen beschrieben werden.

f) Der Quicksortalgorithmus benötigt, abgesehen von dem für die Rekursion benötigten Platz auf dem Aufruf-Stack, keinen zusätzlichen Speicherplatz. Richtig

g) Die Laufzeit des Quicksortalgorithmus ist abhängig von der Zerlegung des Feldes (Aufbau des Feldes - Zufall).

h) Merge Sort ist ein Sortieralgorithmus, der auf dem Prinzip Teile und Herrsche basiert.

i) Die Tiefensuche ist geeignet, um Eigenschaften von allen Knoten in einem Graphen auf dem Bildschirm auszugeben. Richtig

Aufg. 7.01: Verständnisfragen Lsg. (3)

j) Nennen Sie vier verschiedene Suchalgorithmen.

Lineare Suche, Binäre Suche, Interpolationssuche, Breitensuche, Tiefensuche

k) Wann heißen zwei Graphen G1 und G2 isomorph?

Zwei Graphen G1 und G2 heißen isomorph, wenn es eine bijektive Abbildung der Knoten- und Kantenmengen von G1 auf die entsprechenden Mengen von G2 gibt, so dass die Inzidenzbeziehungen erhalten bleiben.

l) Welche zusätzlichen Eigenschaften muss ein Graph erfüllen, damit er ein Baum ist?

Ein Baum ist zyklenfrei und zusammenhängend.

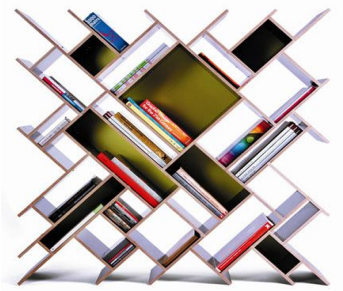
Algorithmen

- Genau definierte Handlungsvorschrift zur Lösung eines Problems oder einer bestimmten Art von Problemen in endlich vielen Schritten
- Wichtige Problemstellungen
  - Sortieren
  - Suchen
  - Optimieren

Sortieren

- Problemstellung
  - Wie kann ich Datensätze sortieren / sortiert ausgeben?
  - Wie kann ich Personen nach Einkommen sortieren?
  - Wie kann ich Daten sortieren, um diese schneller zu verarbeiten?

- Lösung: Sortieralgorithmen
  - Sortieren eines Arrays oder einer Liste
  - Verschiedene Algorithmen
    - Unterschiedliche Geschwindigkeit, Rechen- und Speicherbedarf



BubbleSort

- Einfacher Algorithmus – langsam, aber kaum zusätzlicher Speicherplatz

- Von Pseudocode zu C++ Quellcode

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

```
0  int A[] = {5, 3, 1, 4, 2};
1  for( int i = 0; i < 4; i++ ) {
2    for( int j = 0; j < 4 - i; j++ ) {
3      if( A[j] > A[j+1] ) {
4        int temp = A[j];
5        A[j] = A[j+1];
6        A[j+1] = temp;
7      }
8    }
9  }
```

```
C:\WINDOWS\system32\cmd.exe
Array vorher:      5   3   1   4   2
Array nachher:     1   2   3   4   5
Drücken Sie eine beliebige Taste . . .
```

InsertionSort

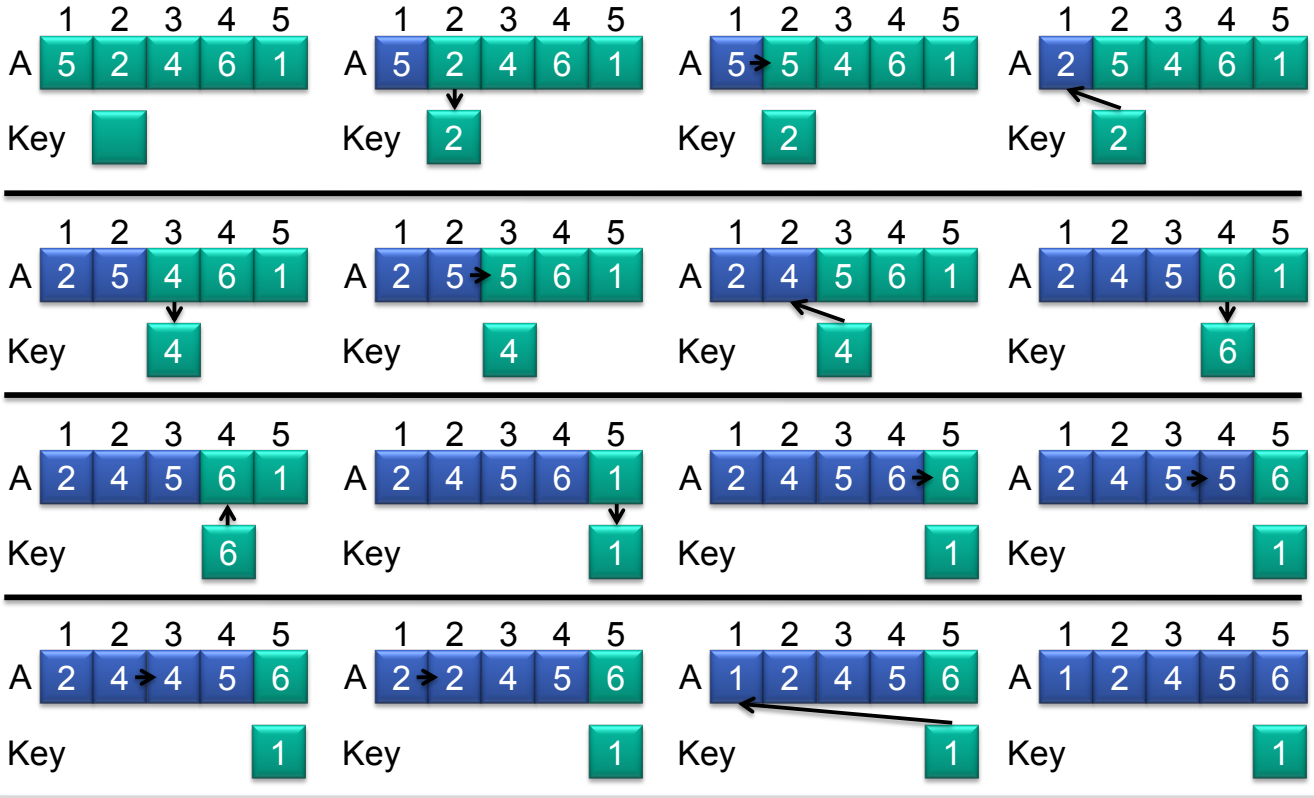
- Sortieren durch Einfügen
- Relativ einfacher Algorithmus
- Vorteile
  - Effizient bei kleinen & bei schon vorsortierten Eingabemengen
  - Einfache Implementierung
  - Stabil
  - Minimal im Speicherverbrauch
- Nachteile
  - Weniger effizient wie komplizierte Sortierverfahren

Link zu Demonstration:
http://einstein.informatik.uni-oldenburg.de/20907.html

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
```


InsertionSort Ablauf



QuickSort

- 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 $O(n \log_2 n)$ im schlechtesten Fall $O(n^2)$

QuickSort Prinzip

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

a) Teile: Zerlege das Feld `A[p..r]` so 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.

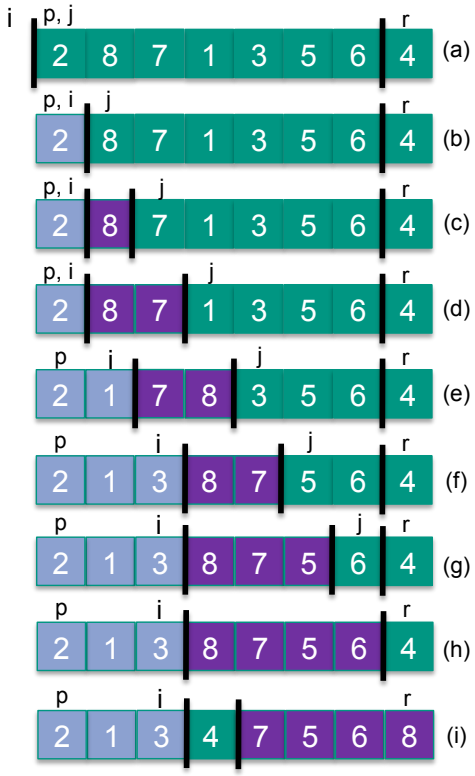
c) Verbinde: Da die Teilfelder in-place sortiert werden (in Feld selbst) ist keine Arbeit erforderlich, um sie zu verbinden. Das gesamte Feld `A[p..r]` ist nun sortiert.

QuickSort Pseudocode

```
Quicksort( A, 1, länge[A] )

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

Partition( A, p, r )
1  x = A[r]
2  i = p - 1
3  for j = p to r - 1
4  do if A[j] <= x
5     then i = i + 1
6          vertausche A[i] mit A[j]
7  vertausche A[i+1] mit A[r]
8  return i+1
```



QuickSort Partition Erklärung

- Die Arbeitsweise von Partition angewendet auf ein Beispielfeld. Hellblau schattierte Feldelemente befinden sich alle in der ersten Partition, in der die Werte nicht größer als x sind. Lila schattierte Elemente befinden sich in der zweiten Partition mit Werten, die größer als sind als x. Die grün schattierten Elemente sind noch nicht in eine der beiden ersten Partitionen eingefügt worden. Das letzte Element ist das Pivoelement.

(a) Das Anfangsfeld und die gesetzten Parameter. Keines der Elemente ist in eine der beiden ersten Partitionen eingefügt worden.

(b) Der Wert 2 wird „mit sich selbst vertauscht“ und in die Partition mit den kleineren Werten eingefügt.

(c-d) Die Werte 8 und 7 werden der Partition mit den größeren Werten hinzugefügt.

(e) Die Werte 1 und 8 werden vertauscht und die kleinere Partition wächst.

(f) Die Werte 3 und 7 werden vertauscht und die kleinere Partition wächst.

(g-h) Die größere Partition wächst, um die Elemente 5 und 6 aufzunehmen. Anschließend terminiert die Schleife.

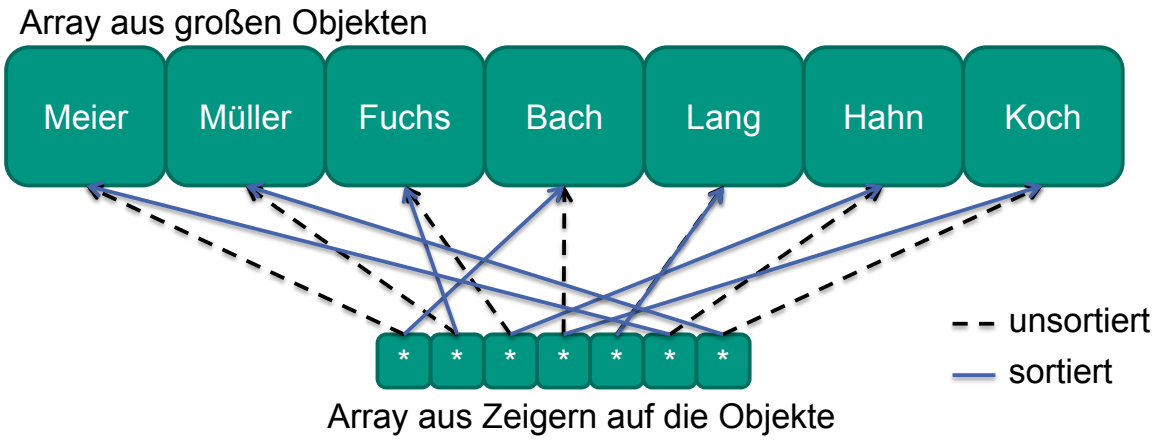
(i) In den Zeilen 7-8 wird das Pivoelement so eingefügt, dass es zwischen den beiden Partitionen liegt.

Link zu Demonstration: http://www.hermann-gruber.com/lehre/sorting/Quick/Quick.html

Sortieren und Zeiger

- 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

Prinzip: Sortieren von Zeigern



- 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

Beispiel: Sortieren dynamischer Objekte (1)

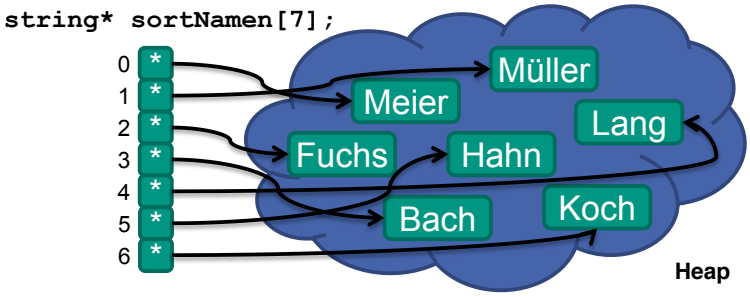
```
#include <iostream>
#include <iomanip>
#include <string>

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;
}
```



Beispiel: Sortieren dynamischer Objekte (2)

```
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 . . .
```

Aufg. 7.02: Arrays und Sortieren

- Schreiben Sie ein Programm, welches mit Hilfe eines InsertionSort-Algorithmus ein 1-dimensionales Array in aufsteigender Reihenfolge sortiert. Testen Sie anschließend Ihr Programm mit geeigneten Testarrays.
- Der folgende Pseudocode beschreibt den InsertionSort-Algorithmus.
- Hinweis: Beachten Sie, dass bei C++ ein Array an der Stelle 0 beginnt, dies ist nicht im Pseudocode berücksichtigt.

```
1 for j = 2 to length[A]
2 do key = A[j]
      //Füge A[j] ein in die sortierte Folge A[1 .. j - 1].
3     i = j - 1
4     while i > 0 and A[i] > key
5     do A[i + 1] = A[i]
6        i = i - 1
7     A[i + 1] = key
```


Aufg. 7.02: Arrays und Sortieren Lsg. (1)

```
int main() {
  long matrix[10] = {546, 21, 65, 1, 987, 88, 654, 5, 98, 123};
  int i, j, key;

  for( j = 1; j <= 9; j++ ) {
    key = matrix[j];
    i = j - 1;
    while( i > -1 && matrix[i] > key ) {
      matrix[i+1] = matrix[i];
      i = i - 1;
    }
    matrix[i+1] = key;
  }

  for( i = 0; i <= 9; i++ )
    cout << setw( 4 ) << matrix[i];
  cout << endl;

  return 0;
}
```

Startet bei 1, da Array bei 0 beginnt

Jedes Element nach dem Ersten

Schiebt die größeren Elemente nach hinten im Array

Fügt das Element an der richtigen Stelle ein

Arrayausgabe

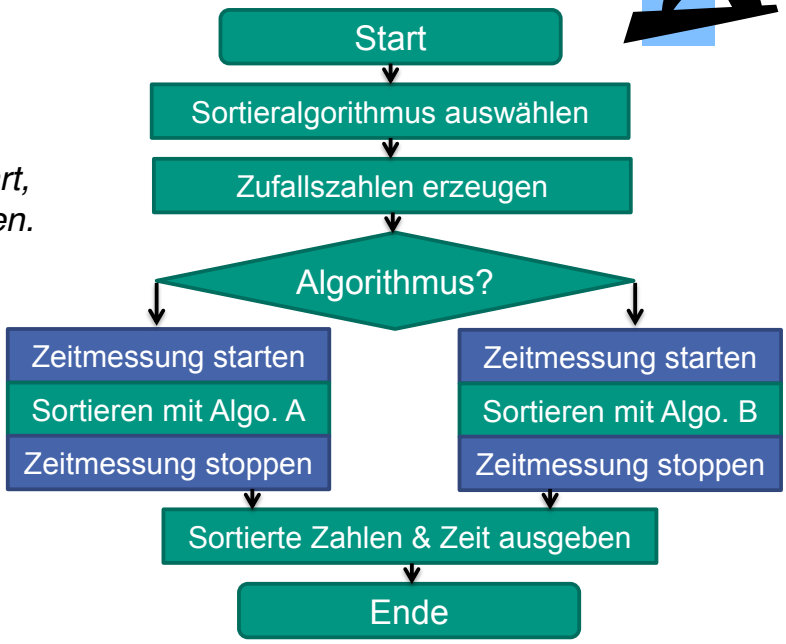
```
C:\WINDOWS\system32\cmd.exe
   1   5  21  65  88  98 123 546 654 987
Drücken Sie eine beliebige Taste . . .
```

Zwischenübung01: Bewertung Alg.

Situationsbeschreibung:
Zwei Forscher treffen sich und möchten sich über die Geschwindigkeit von verschiedenen Sortieralgorithmen austauschen.

Forscher A hat hierzu ein Testprogramm geschrieben, welches er Forscher B als Ablaufdiagramm vorstellt und anschließend zweimal ausführt, um seine Ergebnisse zu zeigen.

Daraufhin Forscher B zu Forscher A …



Link zu Demonstration: http://www.gf-webdesign.de/sortieralgorithmen/

Aufg. 7.03: Laufzeitanalyse Insertion Sort

- In dieser Übung soll die Laufzeit vom "Sortieren durch Einfügen" bzw. Insertion Sort jeweils für den worst- und bestcase bestimmt werden. Hierzu soll anhand des nachstehenden Pseudo-Codes die Gesamtlaufzeit $T(n)$ in beiden Fällen ermittelt und jeweils eine obere asymptotische Schranke bestimmt werden. Dabei nimmt man den Wert n für die Länge des zu sortierenden Arrays A. Noch zu beachten ist, dass c_i die Kosten der Codezeile i darstellt.

```
INSERTIONSORT(A)
1 for j = 2 to laenge(A)
2 do key = A[j]
      //Füge A[j] in die sortierte Folge A[1 .. j - 1] ein.
3     i = j - 1
4     while i > 0 and A[i] > key
5     do A[i + 1] = A[i]
6        i = i - 1
7     A[i + 1] = key
```

Aufg. 7.03: Laufzeitanalyse Insertion Sort Lsg. (1)

- Best-Case Betrachtung — n = Länge des Array
  - Array ist vorsortiert → keine Verschiebungen notwendig
  - Schleife von Zeile 1-7 wird n-1 mal ausgeführt
    - Und Zeile 1 noch einmal zusätzlich wenn die Schleife abgebrochen wird
  - Schleife in Zeile 4 wird sofort abgebrochen, da immer A[i] < key
    - Innere Schleife (Zeile 5 & 6) wird daher nie ausgeführt

				Array A			
				1	2	3	4
j	i	key		2	4	6	8
2	1	4		2	4	6	8
3	2	6		2	4	6	8
4	3	8		2	4	6	8

```
INSERTIONSORT(A)
1 for j = 2 to laenge(A)
2 do key = A[j]
3    i = j - 1
4    while i > 0 and A[i] > key
5    do A[i + 1] = A[i]
6       i = i - 1
7    A[i + 1] = key
```

Zeile	Kostenkoeffizient	Zeit (bestcase)
1	c_1	n
2	c_2	n-1
3	c_3	n-1
4	c_4	n-1
5	c_5	0
6	c_6	0
7	c_7	n-1

Aufg. 7.03: Laufzeitanalyse Insertion Sort Lsg. (2)

- Worst-Case Betrachtung
  - Array ist rückwärts sortiert
    - Verschiebungen immer notwendig
  - Nur Änderungen in der inneren Schleife
    - Zeile 4 – 6 neu zu betrachten

				Array A			
				1	2	3	4
j	i	key		8	6	4	2
2	1	6		8	8	4	2
2	0	6		6	8	4	2
3	2	4		6	8	8	2
3	1	4		6	6	8	2
3	0	4		4	6	8	2
4	3	2		4	6	8	8
4	2	2		4	6	6	8
4	1	2		4	4	6	8
4	0	2		2	4	6	8

$$Z_4 = j \qquad \sum_{j=2}^{n} Z_4 = \sum_{j=2}^{n} j = \left(\sum_{j=1}^{n} j\right) - 1$$

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

$$Z_5 = Z_6 = j - 1 \qquad \sum_{j=2}^{n} Z_{5,6} = \sum_{j=2}^{n} (j-1) = \sum_{k=1}^{n-1} (k) =$$

$$(k = j - 1) \qquad = \frac{(n-1) \cdot n}{2} = \frac{n^2 - n}{2}$$

Zeile	Kostenkoeffizient	Zeit (worstcase)
1	c_1	n
2	c_2	n-1
3	c_3	n-1
4	c_4	$(n^2+n)/2 - 1$
5	c_5	$(n^2-n)/2$
6	c_6	$(n^2-n)/2$
7	c_7	n-1

Aufg. 7.03: Laufzeitanalyse Insertion Sort Lsg. (3)

```
INSERTIONSORT(A)
1 for j = 2 to laenge(A)
2 do key = A[j]
     //Füge A[j] in die sortierte Folge A[1 .. j - 1] ein.
3    i = j - 1
4    while i > 0 and A[i] > key
5    do A[i + 1] = A[i]
6       i = i - 1
7    A[i + 1] = key
```

Worstcase:
$T(n) = O(n^2)$

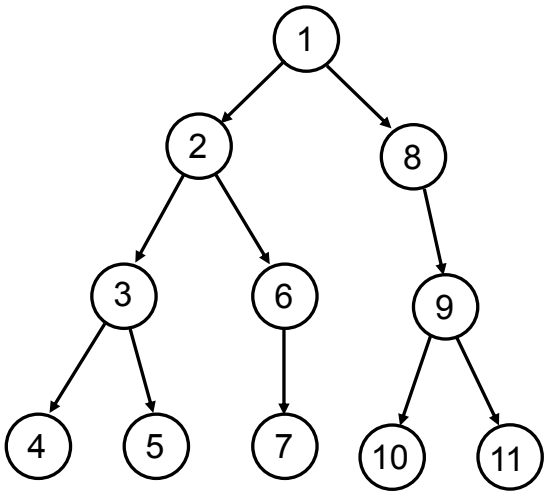
Bestcase:
$T(n) = O(n)$

Zeile	Kostenkoeffizient	Zeit (worstcase)	Zeit (bestcase)
1	c_1	n	n
2	c_2	n-1	n-1
3	c_3	n-1	n-1
4	c_4	$(n^2+n)/2 - 1$	n-1
5	c_5	$(n^2-n)/2$	0
6	c_6	$(n^2-n)/2$	0
7	c_7	n-1	n-1

Allgemeine Tiefensuche

- Gegeben ist ein Graph, sowie ein Anfangsknoten im Graphen, gesucht ist ein Knoten mit einer bestimmten Eigenschaft
- Besuche der Reihe nach alle Knoten, die über Kanten erreichbar sind
  - Also: zunächst den ersten vom Anfangsknoten erreichbaren Knoten
  - Dann: den ersten von diesem Knoten aus erreichbaren, und so weiter
  - Wenn kein Knoten mehr erreichbar ist, kehre zum letzten Punkt zurück, an dem es noch weitere Kanten gab und weiter wieder mit Dann
  - Wenn vom Anfangsknoten alle Kanten abgesucht wurden, ist der gesamte erreichbare Teil des Graphen abgesucht

Beispiel:



Tiefensuche

- Die Tiefensuche (Depth First Search – DFS) exploriert den Graphen zuerst in die Tiefe, wenn möglich. Sie untersucht zuerst die Kanten des zuletzt entdeckten Knotens. Erst wenn diese alle abgearbeitet sind, wird zum letzten Knoten zurückgekehrt (eine Art Backtracking).

- DFS für einen geg. Graphen G = (V,E) und Startknoten S:
  - Verwendet eine Einfärbung der Knoten
    - weiß == noch nicht begonnen
    - grau == in Bearbeitung
    - schwarz == abgeschlossen

- Führt die Entdeckungszeit **`starttime[u]`** und die Bearbeitungsendzeit **`endtime[u]`** mit

Tiefensuche Pseudocode

```
DepthFirstSearch( G )
  for alle Knoten u in G
  do farbe[u] = weiss
     vater[u] = NIL
  zeit = 0
  for alle Knoten u in G
  do if farbe[u] == weiss
     then DFS-VISIT( u )

DFS-VISIT( u )
  farbe[u] = grau; zeit = zeit + 1
  startTime[u] = zeit
  for alle Knoten v aus Adj[u]
  do if farbe[v] == weiss
     then vater[v] = u
          DFS-VISIT( v )
  farbe[u] = schwarz; zeit = zeit + 1
  endTime[u] = zeit
```

Für alle Knoten `u` im Graphen `G`

Falls der Knoten `u` noch nicht untersucht wurde

Für alle mit dem aktuellen Knoten `u` verbundenen Knoten (Adjazenz)

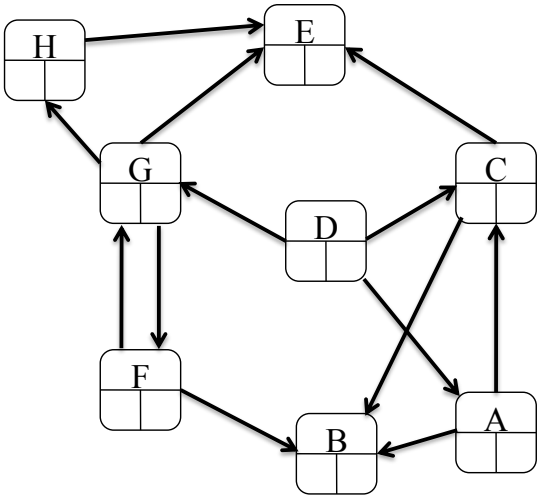
Rekursiver Aufruf

Zwischenübung02: Tiefensuche

Wenden Sie den Pseudocode der Tiefensuche auf den folgenden Graphen an und geben Sie den daraus resultierenden Graphen mit der Entdeckungszeit und der Bearbeitungsendzeit an. Die Knoten werden dabei entsprechend ihrer alphabetischen Reigenfolge gefunden.

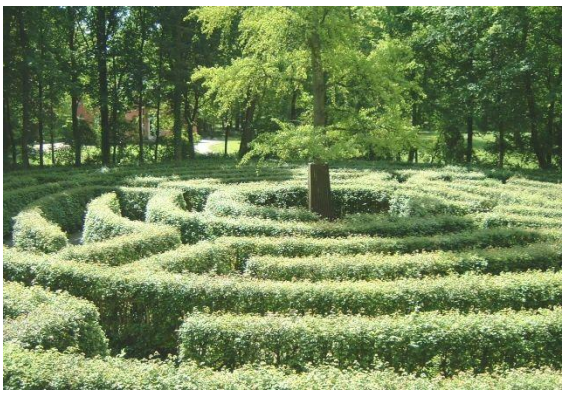
```
DepthFirstSearch( G )
  for alle Knoten u in G
  do farbe[u] = weiss
     vater[u] = NIL
  zeit = 0
  for alle Knoten u in G
  do if farbe[u] == weiss
     then DFS-VISIT( u )

DFS-VISIT( u )
  farbe[u] = grau; zeit = zeit + 1
  startTime[u] = zeit
  for alle Knoten v aus Adj[u]
  do if farbe[v] == weiss
     then vater[v] = u
          DFS-VISIT( v )
  farbe[u] = schwarz; zeit = zeit + 1
  endTime[u] = zeit
```



Tiefensuche: Anwendung auf ein Labyrinth

- Aufgabe: Durchsuchen eines Labyrinths

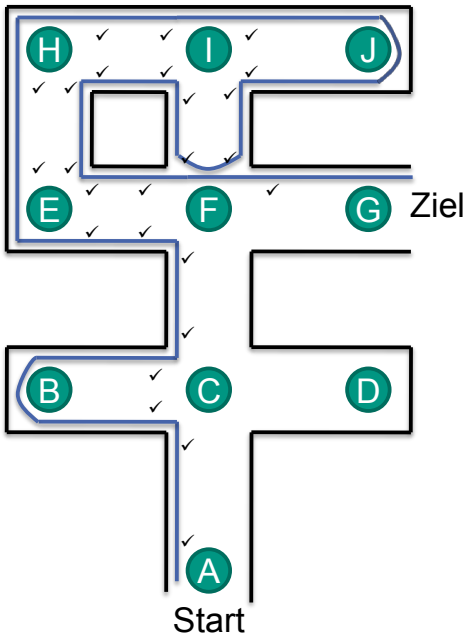


- Präzisierung der Aufgabe:
  - Es soll sichergestellt werden, dass ein Weg vom Start zum Ziel gefunden wird, auch wenn das Labyrinth nicht bekannt ist
  - Die Sichtweite beträgt immer nur ein Feld in jede Richtung
  - Es soll verhindert werden, dass unbemerkt im Kreis gegangen wird
  - Es ist nicht notwendig, den kürzesten Weg zu finden

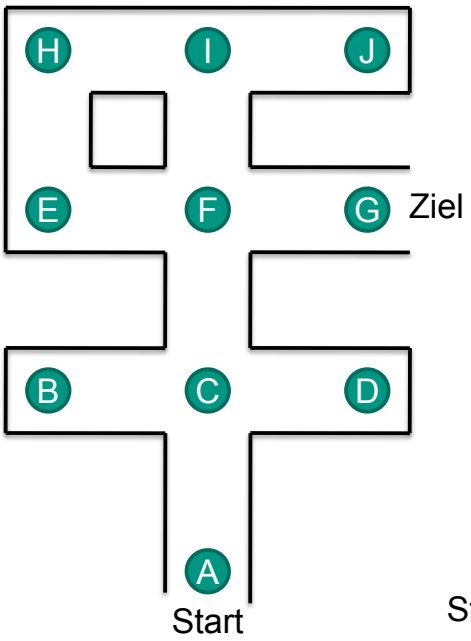
Umsetzung des Tiefensuch-Algorithmus

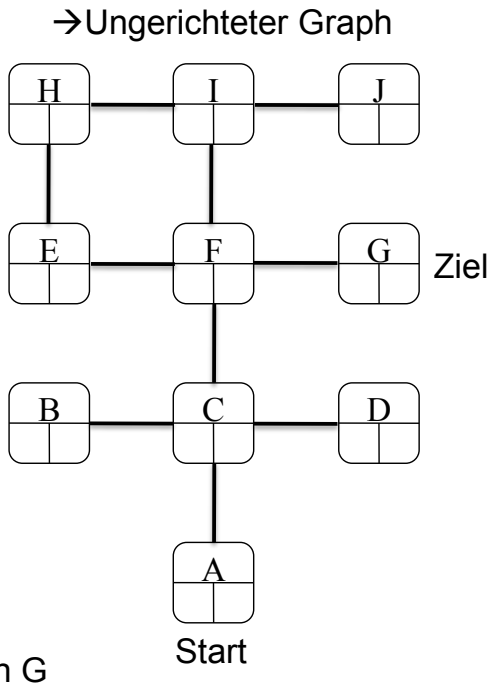
Lösung mit Hilfe von Markierungen an den Kreuzungen:

- Sackgasse: Umdrehen. (Knoten hat keinen Nachfolger)
- Kreuzung: Beim Betreten, Haken an die Wand malen (Entdeckungszeit). Außerdem:
  - Nicht im Kreis laufen! Hat der Gang, durch den man gekommen ist, eben seinen ersten Haken bekommen und sind weitere Haken an der Kreuzung vorhanden: Zweiten Haken malen und umdrehen. (Knoten ist nicht weiß)
  - Sonst: Unerkundete Gänge suchen! Falls Gänge ohne Markierungen vorhanden: Davon den Ersten von links nehmen, Haken an die Wand malen. (nächsten weißen Knoten untersuchen)
  - Sonst: Zurückgehen: Den Gang nehmen, der nur einen Haken hat. (zurück zum vorherigen Knoten)



Umsetzung des Tiefensuch-Algorithmus (1)



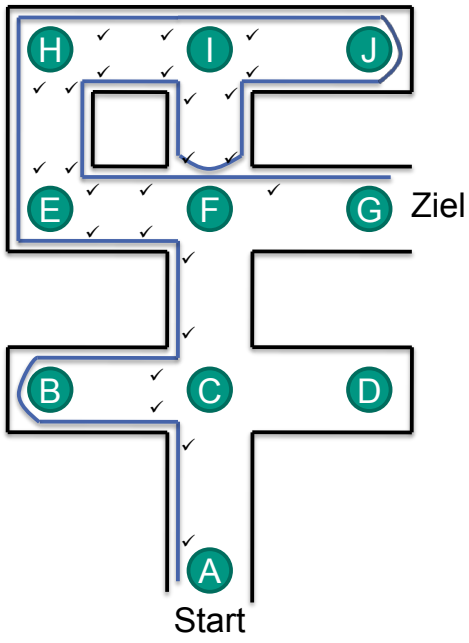


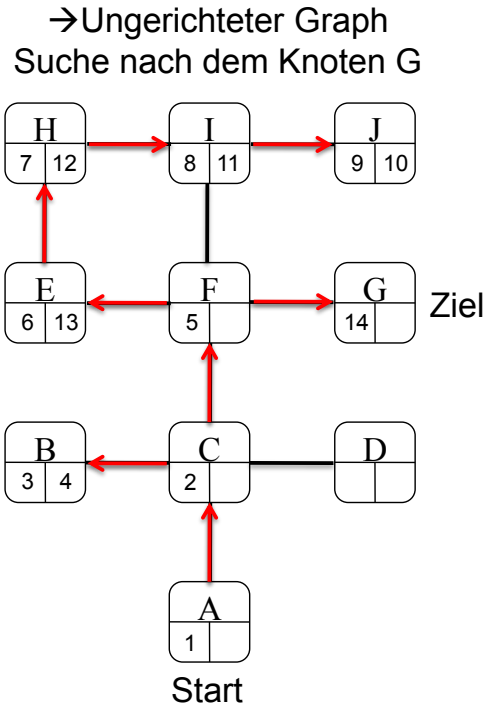
Startknoten A

Gesuchter Knoten G

Gesucht: Ein Weg von A nach G

Umsetzung des Tiefensuch-Algorithmus (2)





Aufg. 7.04: Tiefensuche (1)

- Der folgende Pseudocode beschreibt die Tiefensuche:

```
DepthFirstSearch( G )
  for alle Knoten u in G
  do farbe[u] = weiss
     vater[u] = NIL
  zeit = 0
  for alle Knoten u in G
  do if farbe[u] == weiss
     then DFS-VISIT( u )

DFS-VISIT( u )
  farbe[u] = grau; zeit = zeit + 1
  startTime[u] = zeit
  for alle Knoten v aus Adj[u]
  do if farbe[v] == weiss
     then vater[v] = u
          DFS-VISIT( v )
  farbe[u] = schwarz; zeit = zeit + 1
  endTime[u] = zeit
```

a) Gegeben ist die folgende Adjazenzmatrix. Wenden Sie darauf den Algorithmus der Tiefensuche an und zeichnen Sie den daraus resultierenden Suchbaum (Startknoten A). Markieren Sie dabei (mit einer fortlaufenden Zahl) im jeweiligen Knoten den Zeitpunkt des Entdeckens des Knotens und den Zeitpunkt, wann der Knoten vollständig bearbeitet worden ist. Knoten mit einem kleineren Buchstaben (A ist kleiner als B) werden dabei zuerst gefunden.

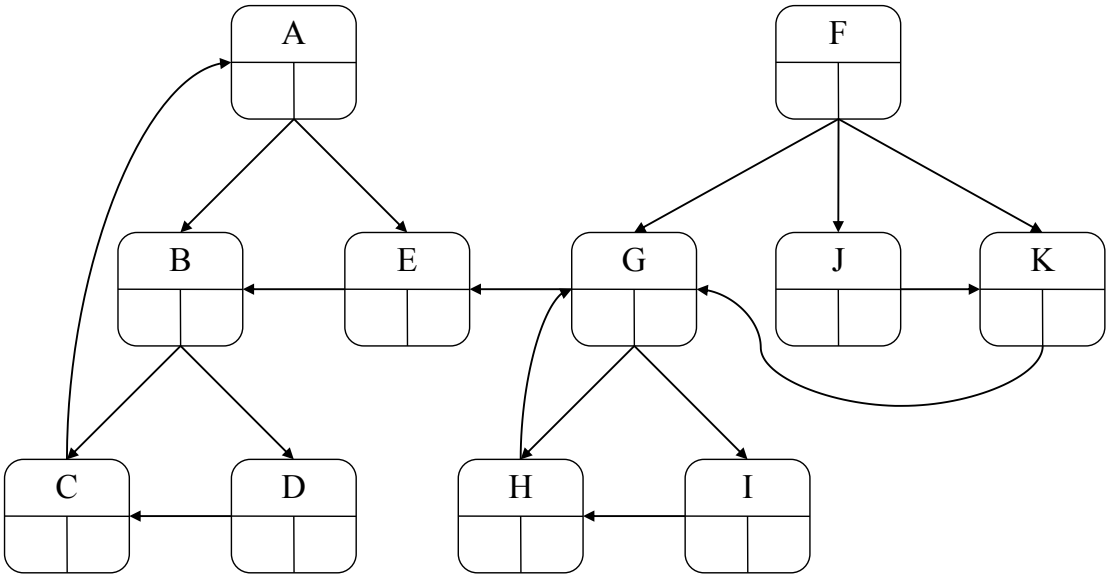
Aufg. 7.04: Tiefensuche (2)

a) Gegeben ist die folgende Adjazenzmatrix:

	A	B	C	D	E	F	G	H	I	J	K
A	0	1	0	0	1	0	0	0	0	0	0
B	0	0	1	1	0	0	0	0	0	0	0
C	1	0	0	0	0	0	0	0	0	0	0
D	0	0	1	0	0	0	0	0	0	0	0
E	0	1	0	0	0	0	0	0	0	0	0
F	0	0	0	0	0	0	1	0	0	1	1
G	0	0	0	0	1	0	0	1	1	0	0
H	0	0	0	0	0	0	1	0	0	0	0
I	0	0	0	0	0	0	0	1	0	0	0
J	0	0	0	0	0	0	0	0	0	0	1
K	0	0	0	0	0	0	1	0	0	0	0

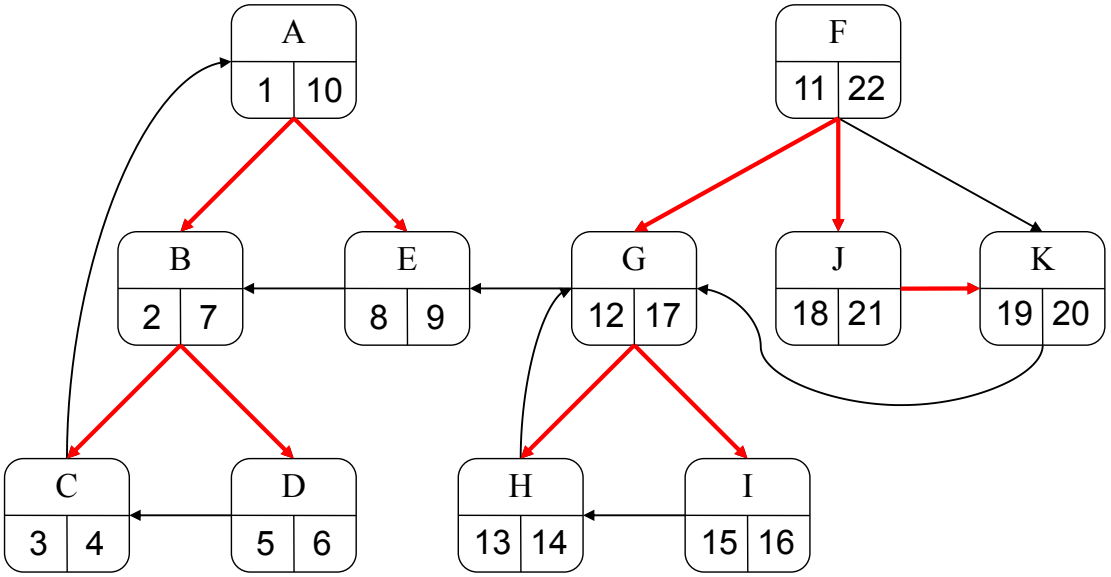
Aufg. 7.04: Tiefensuche Lsg. (1)

- Umsetzung in einen Graph



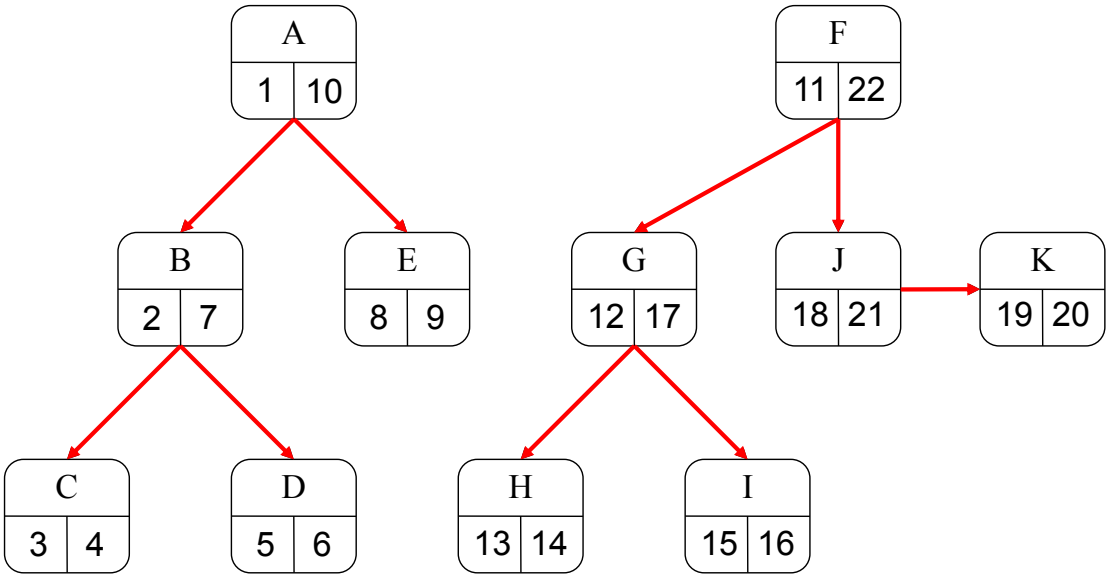
Aufg. 7.04: Tiefensuche Lsg. (2)

- Anwendung der Tiefensuche



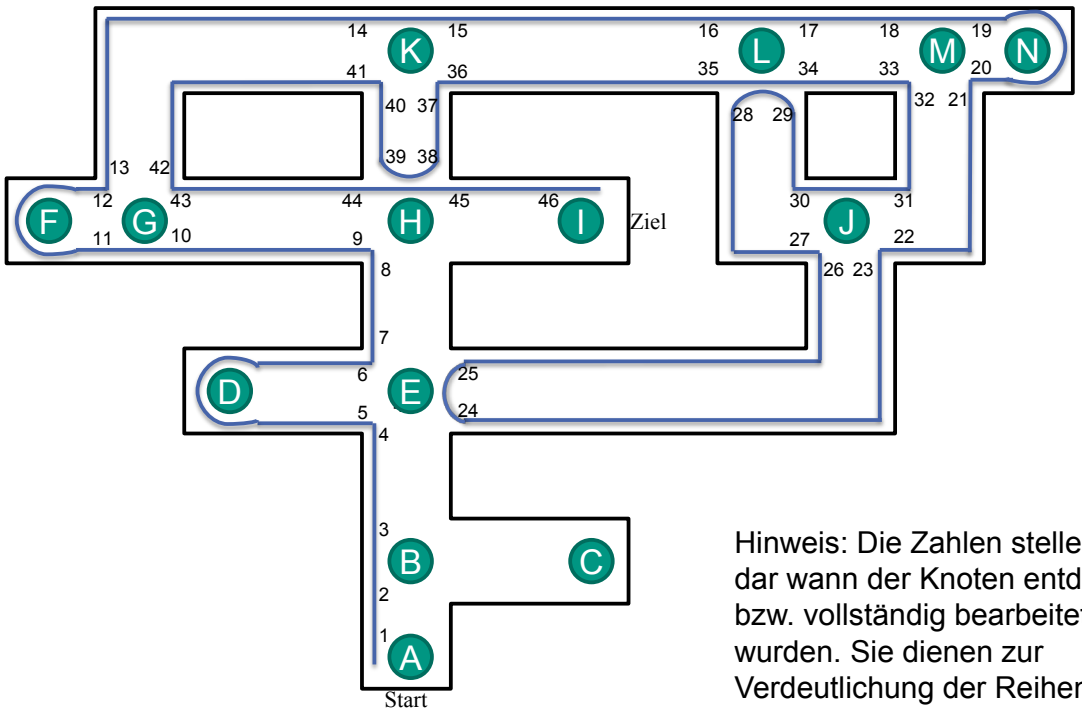
Aufg. 7.04: Tiefensuche Lsg. (3)

- Tiefensuchwald



Aufg. 7.04: Tiefensuche Lsg. (4)

b) Tiefensuche im Labyrinth



Hinweis: Die Zahlen stellen nicht dar wann der Knoten entdeckt bzw. vollständig bearbeitet wurden. Sie dienen zur Verdeutlichung der Reihenfolge.

Ausblick

- Wie kann man einen optimalen Weg finden, um n Städte in einer Rundreise nacheinander zu besuchen?
- Wie ist ein Rechner in Hardware aufgebaut?
- Wie werden meine Daten innerhalb des Rechners verarbeitet?
- Welche Möglichkeiten gibt es, um die Verarbeitung im Rechner zu beschleunigen?

Probeklausur am 18. Juli 2012 um 09:45 Uhr im Benz Hörsaal

Vielen Dank für Ihre Aufmerksamkeit

Tobias Schwalb & Timo Sandmann & Stephan Werner
Karlsruher Institut für Technology (KIT) – ITIV
tobias.schwalb@kit.edu
timo.sandmann@kit.edu
stephan.werner@kit.edu