Review Informatik 2

What did we learn about Algorithms and Data Structures?
The Big Picture

**Algorithms**

Theory:
- Recursion
- Complexity

Practice
- **Testing**, Code Quality
- **Recursive** Solutions
- Backtracking
- Iteration and **Iterators**
- **Enumeration**, Heuristics, **Dynamic Programming**
- Sorting & Searching
- Graph Algorithms

**Data Structures**

Linked Lists
- Single and Double Links
- Anchors

Trees
- Tree Properties
- Sorted Trees aka Search Trees
- Balanced and Splayed Trees

Maps
- Tree Maps
- Hash Maps
- Index Tables

Graphs
- Edge / Vertex List
- Adjacency Matrix / List
- BFS & DFS
- Floyd, Dijkstra, Prim Algorithms
List of Algorithms

Theoretical Algorithms
• Euclid (GCD)
• Factorial
• (Fibonacci )
• Minimal Partial Sums
• Towers of Hanoi
• Ackermann

Basic Recursion
• Fractals
• 8 Queens / Backtracking

Sorting & Searching
• BinSearch
• TreeSort
• Mergesort
• Hashing

Trees and Graphs
• Pre-/In-/Postfix Traversal
• Prim
• BFS, DFS
• Floyd-Warshall
• Dijkstra
Big-O Complexity

1. An Algorithm can have many implementations (GCD)

2. Complexity is property of algorithm (not implementation)
   • always worst case consideration
   • only fastest growing term of formula,
   • ignore constant factors
   Algorithms with the same complexity may differ noticeably in efficiency.

3. Areas where complexity matters:
   • encryption,
   • data bases,
   • mathematical computations and simulations
Complexity of Iterative Algorithms

- Complexity depends on loops.
- Multiply complexities of nested loops.

**BubbleSort** has two nested loops of length n (worst case).
→ Complexity is $O(n^2)$

**Floyd-Warshals Shortest Path algorithm** has 3 nested loops
→ Complexity is $O(n^3)$

**The length of the outer loop of MergeSort is only** $\log n$
→ Complexity of $O(n \times \log n)$
Complexity of Recursive Algorithms

Use recursive call as basic step.
How much does it cost except for the recursive call?
Determine depth of recursion,
i.e. maximum nesting level during rewriting,
depending on which parameter value?
Multiply step cost by recursion depth.

Fac is called \( n \) times, each step has constant complexity,
\( \rightarrow \) Complexity is \( O(n) \):
\[
fac(n) = \begin{cases} 
1 & \text{if } n=1 \\
 n \cdot fac(n-1) & \text{else}
\end{cases}
\]

Fib is called \( 2n \) times,
\( \rightarrow \) Complexity is \( O(n) \) too:
\[
fib(n) = \begin{cases} 
1 & \text{if } (n=1 | n=2) \\
 fib(n-1) + fib(n-2) & \text{else}
\end{cases}
\]

Ackermann has uncomputable growth
\[
f(x) = \begin{cases} 
 x-10 & \text{if } x>100 \\
 f(f(x+11)) & \text{else}
\end{cases}
\]
Typical Complexities

O(1)  Constant complexity
O(n)  Linear complexity
O(n^k) Polynominal complexity
O(k^n) Exponential complexity

O(log n) Logarithmic complexity

Polynominal complexity or better: practically computable
Exponential complexity: practically incomputable

For larger numbers: log(n) is much better than n, even as sqrt(n)

taken from D.Baily, Java Structures
Computational Power

• Recursion matters:

• **There are functions which can be computed recursively, but not iteratively.**

• Easily transformable: tail recursion
• Not transformable: frame recursion.
  → Emulate recursion by implementing a recursion stack

→ Imperative programming languages support recursive calls.
Typical Recursive Algorithms

• Naturally recursive functions like Factorial, Fibonacci
• Towers of Hanoi
• Fractals
• Backtracking
• Depth-First Traversal of Linked Data Structures
Factorial in LISP

(DEFUN FAC (N)
  (COND ((EQUAL N 1) 1)
        ((MULT N (FAC (MINUS N 1)))))

Evaluate Call of FAC (3) by Rewriting:

→ FAC (3)
→ MULT 3 (FAC (MINUS 3 1))
→ MULT 3 (FAC 2)
→ MULT 3 (MULT 2 (FAC (MINUS 2 1))
→ MULT 3 (MULT 2 (FAC 1))
→ MULT 3 (MULT 2 1)
→ MULT 3 2
→ 6
Towers of Hanoi

Puzzle invented by Edouard Lucas in 1883:

- Put an ordered stack of discs from one peg to another, using a third peg,
- such that all stacks are ordered at all times.

- For a stack of 1:
  move disc from start to dest
- For a stack of 2:
  move disc from start to aux
  move disc from start to dest
  move disc from aux to dest
  *Call this: move 2 from start to dest using aux*
- For a stack of 3:
  move 2 form start to aux using dest
  move disc from start to dest
  move 2 from aux to dest using start

Inductive Approach

• A stack of one can be moved directly to the target pole.
• Apparently, we can move a bigger stack of discs correctly to the target pole if we have an auxiliary pole. We have tested it for 2 and 3.

• Use this idea to first move the top of the stack to the auxiliary pole
• leaving the biggest disc behind to be moved directly to the target. move it to the target.
• Then move the top of the stack back to the origin, leaving the auxiliary pole empty again.
• ... until the stack size is 1.

• move the top of the stack is a recursive application of hanoi to a smaller stack!
• Parameters?
Express Hanoi in Java

```java
public class Hanoi extends Applet {
    private Tower start, aux, dest;
    private int size;
    public Hanoi () {
        this.size = 8;               // replace by reading from ComboBox..
        start = new Tower(size);
        aux = new Tower(0);
        dest = new Tower(0);
    }
    public void start() { move(size, start, dest, aux); }

    private void move(int number, Tower s, Tower d, Tower a) {
        if (number==1) { d.add(s.remove()); repaint(); } else {
            move(n-1,s,a,d);
            move(1,s,d,a);
            move(n-1,a,d,s);
        }
    }
    public void paint(Graphics g) {
        start.display(g, size, 50, 10);
        aux.display(g, size, 100, 10);
        dest.display(g, size, 150, 10);
    }
}
```
Recursive Fun: Fractals

- If you draw a simple shape repeatedly with decreasing size and changing orientation, you get a pattern.
- The simplest way to do that is recursion:
  - draw a pattern
  - spawning a similar pattern of smaller size and modified position and orientation
  - which again spawns another smaller pattern
  - until the size is below a certain limit
- Such patterns are usually called fractals...
- And they are fun doing 😊
Two Fractal Classics

**Pythagoras Tree:**
- Draw a square starting from its base line.
- Then, for a given angle, draw a pair of Pythagoras squares opposite to its base line.
- Continue for each square until too small.

**Sierpinski's Triangle:**
- Take a line between two points.
- Replace it by three lines of half that length, turning $-60^\circ$, $+60^\circ$, $+60^\circ$ left.
- Replace each of these lines again by three, turning $+60^\circ$, $-60^\circ$, $-60^\circ$ for the first line, $-60^\circ$, $+60^\circ$, $+60^\circ$ for the middle line, $+60^\circ$, $-60^\circ$, $-60^\circ$ for the last line.
- Repeat for each line, changing orientation every time until too small.
public void paint (Graphics g) {
    g.clearRect(0,0,500,500);
paintTree(g, double x1, double y1,
            double x2, double y2);
}

private void paintTree(Graphics g, double x1, double x2,
                          double y2){
    // tree defined by base line
    if (length(x1,y1,x2,y2) < minlength) return;
paintSquare(g, x1,y1,x2,y2);
paintTree(g, getLeftBase(x1,y1,x2,y2);
paintTree(g, getRightBase(x1,y1,x2,y2);
}

Note: For painting, you need to truncate the coordinates to int,
but for computing, stick with double to retain precision.
Backtracking

- Backtracking is the idea to search all possible solutions paths (or "configurations"),
- always returning from a dead end to try another path.
- It is the "maze" idea
Backtracking

• Basic concept:
  - decide on a first step
  - try all possible continuations (the same way, recursively)
  - undo the first step and try an alternative.
  - finish when all alternatives have been processed.

• How good is it?
  - Any problem that can be solved in a stepwise process, can be solved by backtracking.

• How bad is it?
  - $O(a^n)$ - a being the max. number of choices per step.
  - Exponential!
  - Plausible?
    How many numbers can be expressed by 6 binary digits? $2^6$ 2 is the number of choices in the corresponding search tree.
Eight Queens

• Very famous again – but not so ingenious....
• Place 8 queens safely on a 8x8 chess board.

• Basic idea:
  - You can only have one queen per row (and column), so there must be a queen in each row.
  - So put them on row by row, until you succeed or fail.
  - in case of failure, go back one queen and try another column.

• Complexity:
  - 8 queens, 8 choices per queen
  - \(8^8 = 16,777,216\)
Successful and Failing Configurations

- Successful Configuration 1
- Successful Configuration 2
- Failing Configuration 1
- Failing Configuration 2
The Algorithm

```java
public void placeInRow(int i) {
    for (int h=1; h<=8; h++) {
        if (safe(i,h)) {
            placeQueen(i,h);
            if (i==8) {
                printConfiguration();
            } else {
                placeInRow(i+1);
                removeQueen(i,h);
            }
        }
    }
}
```

Try it by hand for 4 queens, that's only 256 combinations.
Traversing a Linked List

```java
public void printList(Node node) {
    if (node == null) return;
    print(node);
    printList(node.next);
}

public void printListReverse(Node node) {
    if (node == null) return;
    printListReverse(node.next);
    print(node);
}

For a linear list, we normally use iteration rather than recursion.
```
Depth-First Tree Traversal

Infix Traversal

```java
void traverseMidOrder(Node n) {
    if (n==null) return;
    traverseMidOrder(n.left);
    print(n.info);
    traverseMidOrder(n.right);
}
```

Prefix Traversal

```java
void traversePreOrder(Node n) {
    if (n==null) return;
    print(n.info);
    traversePreOrder(n.left);
    traversePreOrder(n.right);
}
```

Postfix Traversal

```java
void traversePostOrder(Node n) {
    if (n==null) return;
    traversePostOrder(n.left);
    traversePostOrder(n.right);
    print(n.info);
}
```
public void TraverseGraph(Graph g) {
    Enumeration<Node> nodes = g.getNodes();
    for (Node n : nodes) n.setWhite();
    for (Node n : nodes)
        if (n.isWhite()) visit(n);
}

private void visit(Node n) {
    n.setGrey();
    Enumeration<Node> successors = n.getSuccessors();
    for (Node succ : successors)
    {   if (succ.isWhite()) visit(succ);
        n.setBlack();
    }
}

/* Depending on the order wanted, print node when marked grey or black */

(c) schmiedecke 09   Inf2- Review
Sorting Algorithms

- BubbleSort
- InsertionSort
- QuickSort
- MergeSort
- TreeSort
- HeapSort
  - ShellSort
  - ShakerSort

http://cg.scs.carleton.ca/~morin/misc/sortalg/
Recursion: Divide and Conquer

- Recursive solution is often "elegant", but complex
- Split it up into subproblems with smaller input set
- combine subset results to final result.
Sorting: MergeSort

Idea:

- Split list in two,
- sort sublists separately,
- merge sublists like a zipper
MergeSort Algorithm

```java
public static void mergeSort(int data[], int temp[], int low, int high) {
    int n = high-low+1;
    int middle = low + n/2;
    int i;
    if (n<2) return;
    // move lower half to temp
    for (i=low; i<middle; i++) temp[i] = data[i];
    mergeSort(temp, data, low, middle-1); // sort lower half
    mergeSort(data, temp, middle, high); // sort upper half
    merge(data, temp, low, middle, high);
}
```

Remember that there is a clever data structure for merge sort, using only ONE extra array.

reminder of hanoi: sort temp using data, sort data using temp....
Greedy Algorithms: Always take as much as you can get.

- Example: Pass out change, using as few coins as possible
- Greedy returns a good result,
- not necessarily the optimum:
  - assume coins of 11, 5 and 1 cents
  - return 15
  - algorithm would yield 11 + 1 + 1 + 1 + 1
  - optimum is 5 + 5 + 5
- But great in complexity (linear) compared to full backtrack (exponential)!
- Greedy looks for a near-optimal solution by combining local optima.
- It is called a heuristic approach – the idea of using good guesses.
Greedily Solvable Problems

• Optimization problems:
  - There is a quality measure for results.
  - The best result is to be determined.
  - Solutions can be constructed incrementally from input values, starting with the empty solution.
  - A suboptimal solution will do.
Prim's Algorithm: Minimal Spanning Tree

```java
int numNodes = 5, graphSize = 1;
int[][] cost = new int[numNodes][numNodes],
    graphEdges = new int[numNodes][numNodes];
boolean[] nodes = { true, false, false, false, false, false };

public void minimalNetwork {
    while (graphSize < numNodes)
        for (int i=1; i<numNodes; i++) {
            if (!nodes[i]) // node not in graph
                // find cheapest edge into graph
            }
            // select cheapest edge of all and add to graph
        }
}
```

(c) schmiedecke 09

Inf2- Review 31
Dynamic Programming: Keep Track of Intermediates

- Optimizing Full Backtracking Solutions,
  - if the search tree is a collapsed tree
  - i.e. some paths are pursued several times.

- Example coin change:
  - If you have found out how to optimally change 7 cents, cache the result!

- What would be the best data structure?
  - An array of size amount – store all precomputed results.

- And complexity goes down by at least halving the exponent.

- ... at the cost of space though, but that is linear ...
The Full Backtracking Algorithm

```java
public static final int[] values =
    { 1, 2, 5, 10, 20, 50, 100, 200 };
public static final String coins =
    {
        "1 cent", "2 cents",..., "2 euros";
    }

// idea: take away one coin and handle the remainder recursively.
//       do so for every type of coin.
public static int numCoins (int amount) {
    int minimum;
    if (amount == 0) return 0;
    for (int i=0; i<coins.length; i++) {
        if (amount >= values[i]) { // try to take away i-th coin
            int possible = 1 + numCoins(amount-values[i]);
            if (minimum > possible) minimum = possible;
        }
    }
    return minimum;
}
```

local variable – no undo required.
The Dynamic Programming Algorithm

```java
public static final int[] values =
    { 1, 2, 5, 10, 20, 50, 100, 200 };
public static final String coins =
    {"1 cent", "2 cents",..., "2 euros"};

public static int numCoins (int amount){
    return numCoins(amount, new int[amount+1]);
} //create cache

public static int numcoins(int amount, int[] cache) {
    int minimum;
    if (amount == 0) return 0;
    if (cache[amount] != 0) return cache[amount];
    for (int i=0; i<coins.length; i++) {
        if (amount >= values[i] {  
            int possible = 1 + numCoins(amount-values[i], cache);
            if minimum > possible) minimum = possible;
        }
    }
    cache[amount] = minimum;
    return minimum;
}
```
Testing, Debugging, Code Quality

- Using JUnit Tests and TestSuites
- Using the Debugger
- Using Refactoring
- Using Style Guides
Data Structures

• Conceptual Data Structures
  - Linear Lists (the most frequent structure)
  - Matrices
  - Trees
  - Graphs
  - Maps
  - ...

• Data Structure Implementations
  - Arrays
  - Linked Structures
  - Combinations
### Example: Tree

#### Conceptual Tree:
- A tree can be used to represent a **hierarchical structure**, e.g. a file system, or a search pattern.
- Then it is conceptually a tree.
- Implementation choices:
  - linked structure
  - 2-dimensional linear structure (e.g. Vector of Vectors)
  - 1-dimensional linear structure (e.g. heap, edge list, vertex list
  - ...

#### Tree Implementation
- A tree can be used to optimize access to a **linear structure**, e.g. a sorted tree or a tree map.
- Then it is conceptually a list or a map.
- Organized, or implemented, as a tree
- With several successive implementation choices, as above.
Linked Lists

• Single- and Double-Linked List
  - With/without Anchor Node

• Operations locate, add, remove, swap
• Techniques for Stack and Queue

• Main Advantage:
  - Flexible

• Main Disadvantage:
  - Random Access requires Traversal.
Single Linked Lists

Basic operations:
- addAfter
- removeAfter
- getNext
- get/setValue
public class Stack {
    private Node top;
    public Element pop() throws EmptyException {
        try {
            Element info = top.getInfo();
            top = top.next();
            return info;
        } catch (NullPointerException e) {
            throw new EmptyException();
        }
    }
}

public class Stack {
    private Node top = new Node(null, null); // Anchor node
    public Element pop() throws EmptyException {
        try {
            return top.removeAfter();
        } catch (NullPointerException e) {
            throw new EmptyException();
        }
    }
}
public class Queue {
    private Node first = new Node(null, null), last;

    public Queue() { last = first; } // start at anchor node

    public void add(Element info) {
        last.addAfter(info);
        last = last.getNext();
    }

    public Element remove() throws EmptyException {
        try {
            return first.removeAfter();
        } catch (NullPointerException e) {
            throw new EmptyException();
        }
    }
}

null

first

null

last

null
Double Linked List

Basic Operations:

- addBefore / addAfter
- removeBefore / removeAfter
- getNext / get Previous
- get/SetInfo
What is a Tree?

- Set of nodes
- Each node has many successors (children)
- but max. one predecessor (father)

→ There is one root node
→ Nodes without successors are called leaves
# Tree Properties

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>node degree</td>
<td>number of children</td>
</tr>
<tr>
<td>tree degree (arity)</td>
<td>maximum node degree</td>
</tr>
<tr>
<td>path</td>
<td>sequence of edges from a node to another in the same subtree</td>
</tr>
<tr>
<td>path length</td>
<td>number of edges in the path</td>
</tr>
<tr>
<td>node level</td>
<td>path length to root</td>
</tr>
<tr>
<td>node height</td>
<td>longest path length to al leaf</td>
</tr>
<tr>
<td>tree height</td>
<td>height of root</td>
</tr>
<tr>
<td>full node</td>
<td>node degree = tree degree</td>
</tr>
<tr>
<td>full tree</td>
<td>leaves only on last level, and all internal nodes full</td>
</tr>
<tr>
<td>complete tree</td>
<td>full tree with some of the rightmost leaves removed</td>
</tr>
<tr>
<td>balanced tree (different defn's)</td>
<td>level of all leaves differs by max. 1 – or full tree with some leaves removed</td>
</tr>
</tbody>
</table>
Sorted Trees for Information Retrieval

 Searching almost trivial:
   If smaller, go left, otherwise go right.
   Search complexity: log(n)

 Adding: extension of searching - add as leaf.

 Removing: Requires a little restructuring for middle nodes –
   replace by predecessor, append right subtree to predecessor.
Adding a Node

for identical values, locate returns a subtree: append new node right of predecessor
Removing a Node

- **move up solitary subtree**
- **move up predecessor node**
Tree Balancing

- **Purpose:** keep depth at log n (lookup complexity)

- **AVL trees:**
  - watch for unbalance, rebalance if required

- **Splay trees:**
  - restructure after every insertion:
    - make new entry root
    - keeps recent entries at the top

- **Basic restructuring operation:** "Rotation"
<table>
<thead>
<tr>
<th>Case</th>
<th>Diagram 1</th>
<th>Diagram 2</th>
<th>Diagram 3</th>
<th>Diagram 4</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Left Left Case</strong></td>
<td><img src="image1" alt="Diagram" /></td>
<td><img src="image2" alt="Diagram" /></td>
<td><img src="image3" alt="Diagram" /></td>
<td><img src="image4" alt="Diagram" /></td>
</tr>
<tr>
<td><strong>Right Right Case</strong></td>
<td><img src="image5" alt="Diagram" /></td>
<td><img src="image6" alt="Diagram" /></td>
<td><img src="image7" alt="Diagram" /></td>
<td><img src="image8" alt="Diagram" /></td>
</tr>
<tr>
<td><strong>Left Right Case</strong></td>
<td><img src="image9" alt="Diagram" /></td>
<td><img src="image10" alt="Diagram" /></td>
<td><img src="image11" alt="Diagram" /></td>
<td><img src="image12" alt="Diagram" /></td>
</tr>
<tr>
<td><strong>Right Left Case</strong></td>
<td><img src="image13" alt="Diagram" /></td>
<td><img src="image14" alt="Diagram" /></td>
<td><img src="image15" alt="Diagram" /></td>
<td><img src="image16" alt="Diagram" /></td>
</tr>
</tbody>
</table>

Rotation Operations for Rebalancing
BinTree Array Implementation: The HEAP Data Structure

The HEAP data structure:

- Array representation of a binary tree, optimal for complete trees (i.e. almost full trees)
- heap[0] contains the root
- heap[1] and heap[2] contain the children of heap[0]
- etc.
- $heap[(i-1)/2]$ contains the father of $heap[i]$ (integer division)
- $heap[2i+1]$ and $heap[2i+2]$ contain the children of $heap[i]$
What is a Map?

• Associative data structure

• Complex information:
  - record of data
  - unique key value for identification
  - e.g. number plate of a car

• Task:
  - Store data so that it can be retrieved by means of the key:

  Who is the owner of 'HZ-AX 1'?

• It's all about information retrieval.
• How to implement an efficient lookup?
Example: Electronic Patient Records

- Patient Records
- Kept in a data base to efficiently find, compare and evaluate patient information
- So a patient record would be a line in a database table
- Unique, identifying key: insurance number

<table>
<thead>
<tr>
<th>insuranceNo</th>
<th>dateofbirth</th>
<th>name</th>
<th>firstname</th>
<th>bloodgroup</th>
<th>inscardread</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEK2233445</td>
<td>1900-02-08</td>
<td>Wassermann</td>
<td>Eugen</td>
<td>0+</td>
<td>1988-05-05</td>
</tr>
<tr>
<td>BKK9953678</td>
<td>1999-07-02</td>
<td>Sandmann</td>
<td>Horst</td>
<td>AB-</td>
<td>2005-01-01</td>
</tr>
<tr>
<td>TKK1702198</td>
<td>1986-06-06</td>
<td>Schnnemann</td>
<td>Jupp</td>
<td>A0</td>
<td>2007-01-06</td>
</tr>
</tbody>
</table>
Natural Associative Data Structures

- dictionaries
- symbol tables
- administrative structures
- lists of utilities (hotels, tools, lectures)
- guest lists
- ....

- you can store them in an array or list, BUT
- in all of these cases, indices would "feel artificial"
- so you should at least hide them (ADT)
Basic Map Operations

- A map entry is a pair (key, content)
- e.g. (Benni, 0177-7777777)
- so they basic operations are
  - add(key, content)
  - retrieve(key) – returns content
  - remove(key) – may return content
  - contains(key) – returns boolean
- i.e., if you have the key, you can retrieve the content.
Map Implementations

Map implementations aim at efficient entry and lookup:

If keys are ordered (i.e. sortable)
• Sorted Linear List → for lookup only
• Sorted Tree → "TreeMap"

For any kind of keys
• Hash Map

Java Collection API contains:
• interface Map
• class TreeMap
• class HashMap
Information Retrieval by Key

- Either:
  - keep the table SORTED
    (and splayed or balanced, if it is a a tree)

- Or:
  - use key hashing to store PatientRecords
Hash Maps

- Find an computable immediate mapping from key to index
- \( \rightarrow \) hash code
- Results in constant access time! \( O(1) \)

Hash Functions

- Using the index of the first letter is a simple hash function.
- Is it a good one?
  - criterion: even distribution over table
  - so, not so good...
  - but prime table length helps.

- Better hash functions on Strings:
  - sum of all letter indeces modulo length
  - sum of weighted characters (higher powers of two)
  - sum of weighted selected characters.

- In Java, hashCode() is an Object function
  - override it for your own types, if you like.
Handling Hash Clashes

- What if the hash slot is already full?

- **Open Addressing:**
  - Rehash by adding an offset and try again
  - Constant offset
  - Double Hashing: computed offset.

- **Problem:** clustering
- **Best distribution with prime length tables**

- **External Chaining**
  - let each entry be the head of a **chain** of hash-isomorphic entries
Implementing a Hash Table

- either list of records
- handle hash clashes by open addressing

- or list of chainable entries
- handle hash clashes by chaining
Index Lookup

- "Find all patients named "Schmiedecke"."
- Key sorting is not helpful, because we are looking for a record field!
- For a list (including hash table):
  - create an index (table) for the name field of the PatientRecord
  - and keep it sorted
  - You can create independant indexes for all record fields!

Patient Schmiedecke
Patient Jeffers
Patient Andresen
Patient Schmiedecke

0 1 2
1 2 1
2 0 3
3 3

index for "name"

so the only data moved during sorting are small integers
Table as List of Columns

<table>
<thead>
<tr>
<th>TKK170286</th>
<th>17/02/1986</th>
<th>Sandmann</th>
<th>Sepp</th>
<th>01/05/2007</th>
</tr>
</thead>
<tbody>
<tr>
<td>BKK887506</td>
<td>01/06/1999</td>
<td>Schneemann</td>
<td>Jutta</td>
<td>12/07/2005</td>
</tr>
<tr>
<td>BEK775534</td>
<td>12/12/1944</td>
<td>Wassermann</td>
<td>Horst</td>
<td>01/06/2007</td>
</tr>
</tbody>
</table>

- Table implemented as *list of columns*.
- Each column has its *data type*.
- Each column may have a *name*.
- A PatientRecord is a set of column values with identical index.
  → If you have looked up a column entry, e.g. "Schneemann", its *index* gives access to the entire PatientRecord.
Graph: Definition

- Structure of nodes (or vertices) and edges.
- directed or undirected
- possibly weighted:
  - vertices have numerical attributes
Optimized Graph Data Structures

• Derived from mathematical models:

• Edge list
  - num. of vertices
  - num of edges
  - sequence of vertex pairs
  - space complexity $2 + 2|E|$

```
6 11 1 2 1 3 3 1 4 1 3 4 3 6 5 3 5 5 6 5 6 2 6 4
```

• Vertex list
  - num of vertices
  - num of edges
  - sequence: num edges, seq. end vertices
  - space complexity $2 + |V| + |E|$

```
6 11 2 2 3 0 3 1 4 6 1 1 2 3 5 3 2 4 5
```
Adjacency Matrix

• Immediate Consequence of
  - Directed Graph definition:
  - $E \subseteq V \times V$
  - Weighted Graph definition:
    - $f : V \times V \rightarrow \mathbb{N}$

• Idea:
  - $|V| \times |V|$ matrix $A$
  - each entry $A[i,j]$ represents edge from $i$ to $j$
  - either boolean, or weight.

• Discussion
  - intuitive structure
  - space complexity $|V|^2$ - only good for "dense" graphs
  - undirected graphs require only $\frac{1}{2}$ matrix (triangle)
Adjacency List

- Linked vertex list
- Dynamic structure – easily extendable
Important Graph Algorithms

• Traversal (visiting ALL vertices)
  - Breadth-First Search (BFS)
  - Depth-First Search (DFS)
  - Cycle Detection (using DFS)
  - Minimal Spanning Tree: Prim's Algorithm (Greedy)

• Paths
  - Topological Sorting / Scheduling (using DFS)
  - Transitive Closure: Floyd-Warshall-Algorithm
  - Shortest Paths: Dijkstra's Algorithm
  - Maximum Flow: Ford-Fulkerson Algorithm
  - Travelling Salesman (NP complete)
DFS-Applications

• Spanning Tree:
  - during DFS, record all edges leading to white nodes.

• Cycles:
  - during DFS, record all edges leading to grey nodes.
  - if there are none, the graph is acyclic

• Scheduling:
  - during DFS, put nodes into a list when they get black.
  - correct only for acyclic graphs – there is no schedule for cyclical graphs!
Shortest Paths
Floyd Algorithm

Question: What is the shortest distance between any A and B?
• Similar to Warshall, but use weighted graph (distance matrix)
• Instead of asserting a path (boolean), compute and enter minimal path length

class Graph {
    private int [][] dist; // adjacency
    public Graph(List edgelist) {
        // similar
    }

    public void computeShortPath() { // modifying the matrix
        for (int mid=0; mid<numnodes; mid++) // must be outer loop
            for (int i=0; i<numnodes; i++)
                for (int j=0; j<numnodes; j++)
                    dist[i][j] =
                        min(dist[i][j], (dist[i][mid] + dist[mid][j]));
    }
}
Shortest Paths
Dijkstra's Algorithm

Restricted Question: **Shortest paths from one start node.**

Domain: weighted, connected, directed graphs.

Greedy pattern,

Modification of Prim's algorithm:

*cheapest node* is node with minimal distance from start node.
Dijkstra Implementation

int numnodes = 100;
int [][] dist; // distance matrix, initialize!
boolean [] T = new boolean[numnodes]; // member of T
int numT = 1;
T[0] = true; // start node
int path[]; // distance from start node
path[0] = 0; // initialize others to 99999

void dijkstra() {
    while (numT < numnodes) { // until all nodes in T
        int candidate = -77, // invalid index
            newpath = 99999;
        for (int i=0; i<numnodes; i++) {
            if (T[i]) { // node in T
                for (int j=0; j<numnodes; j++) {
                    if (!T[j] // node not in T
                        && path[i]+dist[i][j] < newpath) {
                        path[j] =path[i]+dist[i][j];
                        newpath = path[j];
                        candidate = j;
                    }
                }
            }
        }
        if (candidate>0) T[candidate] = true; // new member
        numT++;
    }
}
Travelling Salesman

Question: Minimal tour meeting all nodes.

Sounds simple, but amazingly:
Is NP complete → practically unsolvable:
   no algorithm with $O(n^k)$ for any $k$!
• $(n-1)!$ possibilities
• impossible to iterate through them...

• So you need to live with heuristics which find routes,
e.g. not longer than 2x the minimum.
• Consider literature for discussion ([Lang])
That's it!
All we did in just 70 slides!

WOW 😊