Structure:

TDDB56 has two parts:
- DALG (data structures and algorithms) – IDA
- OPT (optimization) – MAI

They are given in two periods:
- ht1/2006: 11 Fö DALG, 5 Le DALG, 4 Labs DALG
- ht2/2006: 10 Fö OPT, 10 Le OPT, 3 Labs OPT

TDDB57 goes ht1/2006 and consists of:
- DALG-part of TDDB56
- Additional classes: 2 Fö, 1 Le, 1 Lab

Literature:

- DALG TDDB56/TDDB57 (data structures and algorithms)
  - Exercises for lektionerna (.pdf respective homepages)
  - Labkompendium (.pdf respective homepages)

- OPT TDDB56 (optimization)
  - Kaj Holmberg: *Kombinatorisk optimering med linjärprogrammering* (kompendium 2006), Bokakademien
  - Labkompendium (.pdf homepage/labs)

Staff 2006:

**DALG-part:**
- Kursledare, examinator,
  - Jan Maluszynski
  - epost: janma@ida.liu.se
- Kursassistent, lektionsgrupp,
  - Magnus Wahlström
  - epost: magwa@ida.liu.se
- Assistant, lektionsgrupp,
  - David Broman
  - epost: davbr@ida.liu.se
- Kursekreterare: Madeleine Hager
  - epost: madha@ida.liu.se

**OPT-part:**
- Ansvarig: Kaj Holmberg
  - epost: kahol@mai.liu.se
- Kursassistent: to be announced

If mailing: write TDDB56 or TDDB57 in the subject field!
**Examination**

**TDDB56**
- **Kontrollskrivning** (KS) after p1 (not obligatory)
- 3 usual exams covering DALG and OPT:
  - the first exam after p2
    - *you may skip DALG part;*
    - *In this case KS result will be counted as DALG part result*
  - KS result valid only 2006/07

**TDDB57**
- 3 usual exams covering whole TDDB57 material
  - the first exam after p1

---

**Planning – Lectures**

**TDDB56**

**Period 1 (prel):**
1. Organisation, Objectives, Complexity, Big-Oh-notation
2. Analysis of Algorithms
3. Simple Abstract Data Types: Stacks, Queues, Arrays, Lists, etc.
4. Hashing, Skip Lists
5. Binary Search Trees
6. AVL Trees, Multi-way search Trees
7. Priority Queues, Heaps
8. Sorting I
9. Sorting II, Selection
10. Union/Find, ADT Graph
11. Graph Search, Topological sorting
12. TDDB57 Graph algorithms I
13. TDDB57 Graph algorithms II

**TDDB57**

**Planning – Labs**

**Period 1:**
- Experimental evaluation of time complexity
- Hashing (Java)
- Binary Search Trees (Java)
- Quicksort (Java)
  - TDDB57 only: Priority queues and Heapsort (Ada)

**Please register in week 34 through IDAs webReg**

**Period 2 (TDDB56):**
- Simplexmetoden: Övning och implementering.
  - Föden i nätverk: Modellering och lösning med programpaket Netlin.
  - Lokaliseringproblemet: Löst med AMPL samt heuristisk.
### DALG – basics:

**Abstract data types (ADTs)**
- Machine-independent, high-level description
- Of data and operations on them
  - E.g.: Dictionary, Stack, Queue, Priority Queue, Set, ...

**Data structures**: Logical organization of computer memory for storing data

**Algorithms**: High-level description of concrete operations on data structures (to be cont.)

**ADT implemented by suitable data structures and algorithms**

**Program**: Implements algorithms and data structures in some programming language

---

### Algorithm

- **Concrete but machine- and language-independent** specification of all necessary steps (simple operations on data) to compute a solution (output) from a given problem instance (input)
- **Correctness** — the computed output for any given input is as stated in the problem description
- **Algorithm analysis** — time and space consumption, scalability, efficiency, worst case, best case, average case, amortized analysis
- **Algorithmic paradigms** — commonly used problem solving strategies e.g., divide & conquer, dynamic programming, greedy strategy, ...

---

### Solving algorithmic problems

**An algorithmic problem:**
- **Legal input-data**
  - Example:
    - Results of an exam
    - A query: student's name
  - **Expected output**
    - In our case: *yes* or *no*

- What kind of data, which operations on data? → **ADT reuse standard ADTs (to be discussed here)**
- How the data is represented in the computer? → **Data structure**
- How the operations are implemented? → **Algorithm(s)**
Example cont.

- Step 1 – choose suitable Abstract Data Type
  - domain and operations:
    - we choose ADT dictionary:
  - Domain:
    - K: linearly ordered set of keys
    - here: students names with alphabetic order
  - Domain I: information (here: "yes" or "no")
  - Dictionary D: set of elements (key, info)
  - Operations on a dictionary:
    - LookUp(D, key) returns info
    - …details not visible for the user

Example cont.

- Step 2: Implement ADT as a data structure
  - choose representation of data in the computer memory
  - write algorithms for the operations of the ADT
  - Re-use standard ADT’s: Step 2 can be omitted
  - Step 3: Use the operations of the ADT to describe the algorithm solving the problem

In our example LookUp(D, key) returns info

Algorithms will be described in pseudocode

Pseudocode in course book § 1.9.2

- Control flow
  - if then [else ...]
  - while ... do
  - repeat ... until...
  - for ... do...
  - Indentation replaces braces
- Method declaration
  - Algorithm method (arg [], arg ...)
  - Input ...
  - Output ...
- Structured Data
  - array/table ...

We take some freedom

The Random Access Machine (RAM) Model

- A CPU
  - An potentially unbounded bank of memory cells, each of which can hold an arbitrary number or character

- Memory cells are numbered and accessing any cell in memory takes unit time.
**Example:**
Implementing ADT Dictionary with data structure array/table

```plaintext
function LookUpTable (table T[1..n], key k):bool
for i from 1 to n do
  if T[i] = k then return true
  if T[i] > k then return false
return false
```

..."scope" indicated by indentation!

**Different implementation**

Dictionary ADT represented as an ordered, singly linked list L.

The LookUp operation is realized by ListSearch:

```plaintext
function ListSearch (pointer List, key k): boolean
  pointer P ← List
  while P ≠ nil and Key(P) ≤ k do
    if Key(P) = k then return true
    P ← Next(P)
  return false
```

**Discussion**

ADT tells what to do: set of operations on data.

To describe how to do that, we:
- choose a data structure (representation in memory)
- construct algorithms for the ADT operations

The same ADT can be implemented:
- with different data structures
- with different algorithms

The algorithms depend on the data structure chosen. Choose the most efficient solution. What is efficiency?

**Analysis of Algorithms...**

What to analyze:
- correctness
- termination
- efficiency → yes!

Time/space efficiency

- growth rate
- worst case, expected case, amortized
- analysis techniques for iterative algorithms
- analysis techniques for recursive algorithms

**Mathematical background**

[Goodrich/Tamassia 4.2]

[Goodrich/Tamassia 4.1, 4.2.3]
Principles of Algorithm Analysis

An algorithm should work for (input) data of any size. (Example LookUpTable: input size is the size of the table.) Show the resource (time/memory) used as an increasing function of input size.

Focus on the worst case performance. Ignore constant factors

• analysis should be machine-independent;
• more powerful computers introduce speed-up by constant factors.

Study scalability / asymptotic behaviour for large problem sizes:

• ignore lower-order terms, focus on dominating terms.

How to compare time/space efficiency?

Study execution time (or memory usage) as a function of the size of input data:

• When are they “equally” effective?

• When is one better than the other?

Big-Oh Notation (§4.2.5)

Given functions \( f(n), g(n) \)

\( f(n) \) is \( O(g(n)) \) if there are positive constants \( c \) and \( n_0 \) such that

\[ f(n) \leq cg(n) \quad \text{for} \; n \geq n_0 \]

Example: \( 4n + 2 \) is \( O(n) \)

- \( 4n + 2 \leq cn \)
- \( (c - 4) n \geq 2 \)
- \( n \geq 2(c - 4) \)
- Pick \( c = 5 \) and \( n_0 = 2 \)

But also \( n \) is \( O(4n+2) \)

### Comparison with simple math function:

<table>
<thead>
<tr>
<th>( n )</th>
<th>( \log_{10} n )</th>
<th>( n \log_{10} n )</th>
<th>( n \log_{10}^2 n )</th>
<th>( n^2 )</th>
<th>( 2^n )</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>16</td>
<td>4</td>
<td>16</td>
<td>64</td>
<td>256</td>
<td>6.5( \times 10^4 )</td>
</tr>
<tr>
<td>64</td>
<td>6</td>
<td>64</td>
<td>384</td>
<td>4096</td>
<td>1.84( \times 10^{19} )</td>
</tr>
</tbody>
</table>

1.84\( \times 10^{19} \) \( \mu \text{sec} = 2.14 \times 10^8 \) days = 5845 centuries
Big-Oh Example

Example:
the function $n^2$ is not $O(n)$
- $n^2 \leq cn$
- $n \leq c$
- The above inequality cannot be satisfied since $c$ must be a constant

Big-Oh and Growth Rate

"$f(n)$ is $O(g(n))$" means the growth rate of $f(n)$ is no more than the growth rate of $g(n)$

We can use the big-Oh notation to rank functions according to their growth rate

<table>
<thead>
<tr>
<th>$g(n)$ grows more</th>
<th>$f(n)$ is $O(g(n))$</th>
<th>$g(n)$ is $O(f(n))$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Same growth</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Types of growth comparison:

- $f$, $g$ growing functions from natural numbers to positive real numbers
- $f$ is (in) $O(g)$ iff there exist $c > 0, n_0 \geq 1$ such that $f(n) \leq c g(n)$ for all $n \geq n_0$
  Intuition: Apart from constant factors, $f$ grows at most as quickly as $g$
- $f$ is (in) $\Omega(g)$ iff there exist $c > 0, n_0 \geq 1$ such that $f(n) \geq c g(n)$ for all $n \geq n_0$
  Intuition: Apart from constant factors, $f$ grows at least as quickly as $g$
  $\Omega$ is the converse of $O$, i.e. $f$ is in $\Omega(g)$ iff $g$ is in $O(f)$
- $f$ is (in) $\Theta(g)$ iff $f(n) \in O(g(n))$ and $g(n) \in O(f(n))$
  Intuition: Apart from constant factors, $f$ grows exactly as quickly as $g$
Big-Oh Rules

- If \( f(n) \) a polynomial of degree \( d \), then \( f(n) \) is \( O(n^d) \), i.e.,
  1. Drop lower-order terms
  2. Drop constant factors
- Use the smallest possible class of functions
  - Say “2n is \( O(n) \)” instead of “2n is \( O(n^2) \)”
- Use the simplest expression of the class
  - Say “3n + 5 is \( O(n) \)” instead of “3n + 5 is \( O(3n) \)”

Examples: Prove or disprove...

- Assuming algorithm \( f \) has an execution time behaviour \( T(f) \) as listed below, e.g., \( T(f) = (n+1)^2 \) ...

\[
\begin{align*}
(n+1)^2 &\in O(n^3) \\
(n-1)^3 &\in O(n^2) \\
3^n - 1 &\in O(2^n) \\
\sqrt{n^5} &\in O(n^2)
\end{align*}
\]

To check growth rate?

To check \( f \in O(g) \), \( f \in \Omega(g) \), \( f \in \Theta(g) \), analyze

\[
l = \lim_{n \to \infty} \frac{f(n)}{g(n)}
\]

- \( f \in O(g) \) iff \( l \leq \infty \)
- \( f \in \Omega(g) \) iff \( l > 0 \)
- \( f \in \Theta(g) \) iff \( 0 < l < \infty \)