Topics:
1: Algorithms, analysing algorithms, Asymptotic notation (for talking about running-times), Sequential Data Structures, Tree data structures, Hashing, Priority Queues, Advanced sorting.
2: Algorithms for searching graphs, applications to graph problems.

Textbooks
For Algorithms and Data Structures (recommended, not required.):
- [GT] *Data Structures and Algorithms in Java*, by Goodrich & Tamassia (4th or 3rd ed), Wiley. Gentle textbook, best for this course (doesn’t have WWW stuff). Java.

If you will not take 3rd year ADS, choose [GT], but don’t rush out to buy a book straight away.

Study advice
1. Education is done with you not to you.
2. You are here because you want to learn the subject.
3. Course consists of:
   - Lectures.
   - Tutorials.
   - Practical work (2 assignments only).
   - Private study.

Deciding not to take an active part in all of these is deciding to under perform at best and fail at worst. It is not possible to coast along and revise just before the exams (unless failure seems like a good idea).

Finally:
- Lectures start at 4.10, keep any eye on the clock and wind down any conversation.
- In lectures either I talk or you talk but not both!
Our Ingredients

Algorithms  Step-by-step procedure (a “recipe”) for performing a task.
Data Structures  Systematic way of organising data and making it accessible in certain ways.

- We are interested in the design and analysis of “good” algorithms and data structures.
- Think about very large systems and the need to have them work within acceptable time.

What you have probably seen already

Data Structures
Arrays, linked lists, stacks, trees.

Algorithm design principles
Recursive algorithms.

Searching and Sorting Algorithms
Linear search and Binary search. Insertion sort, selection sort.

Other prerequisites:
- The ability to reason mathematically, spot a bad argument from a mile off.
- Write down a mathematical argument fluently. It should be a pleasure to read.
- See Note 1 for advice on setting out mathematical reasoning.

Evaluating algorithms

- Correctness
- Efficiency w.r.t.
  - running time,
  - space (=amount of memory used),
  - network traffic,
  - number of times secondary storage is accessed.
- Simplicity

Measuring Running time

The running time of a program depends on a number of factors such as:
1. The input.
2. The running time of the algorithm.
3. The quality of the implementation and the quality of the code generated by the compiler.
4. The machine used to execute the program.

We will rarely be concerned with the implementation quality, the code quality or the machine.
- A given algorithm can be implemented by many different programs (indeed languages).
Example 1: Linear Search in JAVA

```java
public static int linSearch(int[] A, int k) {
    for (int i = 0; i < A.length; i++)
        if (A[i] == k)
            return i;
    return -1;
}
```

This is Java.
- We want to ignore implementation details, so we map this to pseudocode.
- In reality things are the other way round!

Linear Search in Pseudocode

Input: Integer array \( A \), integer \( k \) being searched.
Output: The least index \( i \) such that \( A[i] = k \); otherwise \(-1\).

Algorithm \text{linSearch}(A, k)
1. \text{for } i \leftarrow 0 \text{ to } A.length - 1 \text{ do}
2. \text{if } A[i] = k \text{ then}
3. \text{return } i
4. \text{return } -1

Suppose \( A = \langle 19, 5, 6, 77, 2, 1, 90, 3, 4, 22, 1, 5, 6 \rangle \) and \( k = 1 \).
What happens?

Worst Case Running Time

Assign a size to each possible input.

Definition
The \textit{(worst-case) running time} of an algorithm \( A \) is the function \( T_A : \mathbb{N} \rightarrow \mathbb{N} \) where \( T_A(n) \) is the maximum number of computation steps performed by \( A \) on an input of size \( n \).

Example: \text{linSearch}.
- Suppose the size is the length of the array \( A \).
- Worst-case running time is a linear function of size.

Note:
- Implicit assumption that array entries are of bounded size.
- Otherwise we could take sum of all array entry sizes as measure of input size (plus size of \( k \)).

Average Running Time

In general worst-case seems overly pessimistic.

Definition
The \textit{average running time} of an algorithm \( A \) is the function \( AVT_A : \mathbb{N} \rightarrow \mathbb{R} \) where \( AVT_A(n) \) is the average number of computation steps performed by \( A \) on an input of size \( n \).

Problems with average time
- What precisely does \textit{average} mean? What is meant by an “average” input depends on the application.
- Average time analysis is mathematically very difficult and often infeasible (OK for \text{linSearch}).
Analysis of Algorithms

A nice approach would be to combine:

\[ \text{Worst-Case Analysis} + \text{Experiments} \]

We will aim for this but

- Java’s Garbage Collection hampers the quality of our experiments.

Example 2: Binary Search

Input: Integer array \( A \) in increasing order, integers \( i_1, i_2, k \).
Output: An index \( i \) with \( i_1 \leq i \leq i_2 \) and \( A[i] = k \), if such an \( i \) exists, \(-1\) otherwise.

Algorithm binarySearch(\( A, k, i_1, i_2 \))
1. if \( i_2 < i_1 \) then return \(-1\)
2. else
3. \( j \leftarrow \left\lfloor \frac{i_1 + i_2}{2} \right\rfloor \)
4. if \( k = A[j] \) then
5. return \( j \)
6. else if \( k < A[j] \) then
7. return binarySearch(\( A, k, i_1, j - 1 \))
8. else
9. return binarySearch(\( A, k, j + 1, i_2 \))

Running-time of Binary search

Input array with \( n = i_2 - i_1 + 1 \) (the number of items in the region we search).

- Do at most a constant \( c \) amount of work.
- If \( k \) found done else recurse on array of size about \( n/2 \).
- Do a constant \( c \) amount of work.
- If \( k \) found done else recurse on array of size about \( n/2^2 \).
  ...
- Do a constant \( c \) amount of work.
- If \( k \) found done else recurse on array of size about \( n/2^r \).

Base case: \( n/2^r = 1 \), i.e., \( r = \log(n) \). Then one more call.
Total work done (time) no more than
\[ c(\log(n) + 2). \]

Better than linSearch?

\[
T_{\text{linSearch}}(n) = 10n + 10,
T_{\text{binarySearch}}(n) = 1000\log(n) + 1000.
\]
**lg n versus n**

Put

\[ m = \lg n. \]

By definition

\[ n = 2^m. \]

Now:

\[ m \to m + 1 \quad n \to 2n \]
\[ m \to m + 5 \quad n \to 32n \]
\[ m \to m + 10 \quad n \to 1024n \]
\[ m \to m + c \quad n \to 2^cn \]

---

**Some Statistics**

Jan 2008 on a DICE machine.

<table>
<thead>
<tr>
<th>size</th>
<th>wc linS</th>
<th>avc linS</th>
<th>wc binS</th>
<th>avc binS</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>( \leq 1 \text{ ms} )</td>
<td>( \leq 1 \text{ ms} )</td>
<td>( \leq 1 \text{ ms} )</td>
<td>( \leq 1 \text{ ms} )</td>
</tr>
<tr>
<td>100</td>
<td>( \leq 1 \text{ ms} )</td>
<td>( \leq 1 \text{ ms} )</td>
<td>( \leq 1 \text{ ms} )</td>
<td>( \leq 1 \text{ ms} )</td>
</tr>
<tr>
<td>1000</td>
<td>( \leq 1 \text{ ms} )</td>
<td>( \leq 1 \text{ ms} )</td>
<td>( \leq 1 \text{ ms} )</td>
<td>( \leq 1 \text{ ms} )</td>
</tr>
<tr>
<td>100000</td>
<td>( \leq 1 \text{ ms} )</td>
<td>( \leq 1 \text{ ms} )</td>
<td>( \leq 1 \text{ ms} )</td>
<td>( \leq 1 \text{ ms} )</td>
</tr>
<tr>
<td>200000</td>
<td>( \leq 1 \text{ ms} )</td>
<td>( \leq 1 \text{ ms} )</td>
<td>( \leq 1 \text{ ms} )</td>
<td>( \leq 1 \text{ ms} )</td>
</tr>
<tr>
<td>400000</td>
<td>3 ms</td>
<td>1 ms</td>
<td>1 ms</td>
<td>1 ms</td>
</tr>
<tr>
<td>600000</td>
<td>3 ms</td>
<td>1.3 ms</td>
<td>1 ms</td>
<td>1 ms</td>
</tr>
<tr>
<td>800000</td>
<td>3 ms</td>
<td>1.5 ms</td>
<td>1 ms</td>
<td>1 ms</td>
</tr>
<tr>
<td>1000000</td>
<td>5 ms</td>
<td>2.1 ms</td>
<td>1 ms</td>
<td>1 ms</td>
</tr>
<tr>
<td>2000000</td>
<td>7 ms</td>
<td>3.7 ms</td>
<td>1 ms</td>
<td>1 ms</td>
</tr>
<tr>
<td>4000000</td>
<td>12 ms</td>
<td>6.9 ms</td>
<td>1 ms</td>
<td>1 ms</td>
</tr>
<tr>
<td>6000000</td>
<td>24 ms</td>
<td>11.6 ms</td>
<td>1 ms</td>
<td>1 ms</td>
</tr>
<tr>
<td>8000000</td>
<td>24 ms</td>
<td>15.6 ms</td>
<td>1 ms</td>
<td>1 ms</td>
</tr>
</tbody>
</table>

200 repetitions for each size.

---

**Why not just do experiments?**

- Consider sorting arrays of the integers 1, 2, \ldots, 100 held in some order.
- Just take a 1% sample of all possible inputs.
- How many experiments?

\[ 99! = 9332621544394415268169923885626670049071596826438 \]
\[ 16214685929638952175999322991560894146397615651 \]
\[ 82862536979208272237582511852109168640000000000000 \]

Assume algorithm can sort \( 10^{50} \) instances per second(!).

How long do we need to wait?

\[ \frac{99!}{60 \times 60 \times 24 \times 366 \times 10^{50}} \approx 2.951269209 \times 10^{98} \text{ years.} \]

Be seeing you!