Algorithms and Data Structures (INF1)

Lecture 1/15

Hua Lu
Department of Computer Science
Aalborg University
Fall 2007
Agenda

• Course Organization
  ■ Administration
  ■ Session Structure
  ■ Exam
  ■ Tips
• Introduction
• Case Study
• Conclusion
Administration

• Lecturer
  - Hua Lu
    - Email: luhua@cs.aau.dk, Office: 3.2.05

• Homepage
  - http://www.cs.aau.dk/~luhua/courses/ad07/
  - Do check it frequently for notifications and updates!

• Textbook
  - Algorithms and Data Structures: Design, Correctness and Analysis (2nd ed.) by Jeffrey H. Kingston

• Other References
  - Algorithmics: The Spirit of Computing by David Harel
  - Algorithm Design by Jon Kleinberg and Éva Tardos
  - Introduction to Algorithms (2nd ed.) by Thomas H. Cormen et al.
Session Structure

• A total of 15 sessions
  ■ Exercise class + Lecture class
  ■ No exercise class in this session, but you will get exercises in the end of this session

• Exercise class
  ■ ~ 2 hours
  ■ Solve exercises assigned in the previous session
    ◆ Work in groups but select a person to write down compulsory exercises
    ◆ I will come by from group to group during each exercise class
  ■ Get a feeling of exam questions from enough exercises!

• Lecture class
  ■ ~ 2 hours
  ■ Teach specific knowledge
  ■ Assign corresponding exercises
Exam

- Individual and written, but open-book
  - Any electronic devices like laptops and mobile phones are NOT allowed.
  - But you can freely use your copies of slides from the lectures, textbooks, and other course material common to all of us.

- Exam Questions
  - Totally 3 questions
    - 50 points for Q1 + 25 points for Q2 + 25 points for Q3
  - Question 1 consists of 6-8 sub-questions, all of which are objective
    - You just need to pick a choice, or write down only the final answer, without any written reasoning.
  - Both questions 2 and 3 are sort of subjective
    - You need to think actively, and write down your justification together with the solutions (e.g. algorithm design and complexity analysis).
  - Refer to exams of previous years, released on the homepage
Tips

• Your feedback is always welcome
  ■ Esp. when you get unhappy or confused
  ■ Email, phone call, drop by my office…

• Do participate in every lecture

• Play active in every exercise class
  ■ Exercises prepare you for the final exam!
  ■ Make sure you YOURSELF understand all exercises after your team work in each class

• Check the homepage frequently
Agenda

• Course Organization
• Introduction
  ■ Algorithms and Algorithmic Problems
  ■ Data Structures vs. Algorithms
• Case Study
• Conclusion
What is An Algorithm

• The Wiki definition
  - An **algorithm** is a finite list of well-defined instructions for accomplishing some task that, given an initial state, will terminate in a defined end-state.

• What kinds of tasks or problems to tackle?
  - Personal tax calculation and deduction
  - Arrange rooms for lectures
  - Find a best path from my home to AAU
  - Simulate the global climate changes

• Intuitions
  - Different problems have different degrees of difficulty
    - Tax stuff is definitely easier than climate simulation
  - We can solve them using specific
    - procedures, recipes, process descriptions -> algorithms
Problem Solving Procedure

• Analysis
  - Specification
    ✤ Input (Pre-conditions)
    ✤ Output (Post-conditions)

• Design
  - Algorithms
  - Data structures

Covered by this course!

• Implementation
  - Programs in a real programming language

Real-life Problem

Algorithmic Problem

Coding Blueprint

```java
public int compare(int a, int b)
{
    if (a >= b)
        return 1;
    else
        return 0;
}
```
Algorithmic Problem and Solution

An algorithm describes actions on the input instance
- Produces output as function of input

For a single algorithmic problem
- The number of legal input instances can be infinite.
  - Specification example: A sorted, non-descending sequence of natural numbers with non-zero but finite length.
    - 1, 20, 908, 909, 100000, 1000000000.
    - ...
  - There may be zero to many correct algorithms!
Data Structure

• **Organization** of data needed in algorithms
  - Different kinds of data structures are suited to different kinds of applications; specific applications may need corresponding data structures
    - E.g., trees for database indexes, graphs for traffic networks

• A data structure can
  - encapsulate different levels of abstraction
    - Array of integers, array of self-defined data structures
  - have different implementations and representation forms in different programming languages
    - Data structures are key organizing factor in most languages
      - Classes in Object-oriented programming (OOP) languages
    - We in this course will only introduce the abstract forms and operations of basic data structures
      - Will not discuss concrete issues in any real programming language
Data Structures vs. Algorithms

• Choice of data structures
  ■ A primary design consideration at many times
  ■ Determines the algorithms to be used
    ◆ “A well-designed data structure allows a variety of critical operations to be performed using as few resources, both execution time and memory space, as possible.”

• The opposite direction sometimes
  ■ Certain algorithms require particular data structures
    ◆ E.g., shortest path algorithm needs priority queues

Data Structures
How do we organize the data to be handled?

Algorithms
Describe the process by which data is handled.

Duality
Criteria for Algorithms

• Correctness
  - We want algorithms that do the right job!
  - To be correct is a MUST!!!
  - Wrong algorithms do not always cause disasters, but they CAN
    - Tax calculation and deduction
    - Space shuttle control

• Efficiency / Complexity
  - How much memory space does an algorithm need?
  - How long time does an algorithm need to produce what we want?
  - Good or bad algorithm
  - May depend on inputs
    - Best case, worst case, average case

• Both aspects may depend on data structures used
Agenda

• Course Organization
• Introduction
• Case Study
  ■ Linear Search
  ■ Binary search
• Conclusion
Case Study on Search

• Why do we use search?
  - Search sequence is simple yet very important
  - Widely used in so many applications ranging from primitive sequences to highly complicated ones
    - Phonebook search
    - DNA search
    - Fingerprint search

• This is a warm-up case study that
  - Involves simple integer sequence only
  - Gives you real examples of some basic concepts
    - Algorithms, efficiency
  - Prepares you for further knowledge to be taught in this course
  - Ideas behind can be applied to more complicated cases
Specification of Search Problem

**INPUT**
- A sequence of \( n \) (\( n > 0 \)) integers
  - \( A[1..n] \) – an array of integers
- A single integer
  - \( q \)

\[ a_1, a_2, a_3, \ldots, a_n; \ q \]

**OUTPUT**
- An index of the found integer or \( 0 \)
  - \( j \) such that \( A[j] = q \);
  - 0 if \( q \) is not found

\[ j \]

**Input Instances**

\[ 2 \ 5 \ 4 \ 10 \ 11; \ 5 \]

\[ 2 \ 5 \ 4 \ 10 \ 11; \ 9 \]

**Output**

\[ 2 \]

\[ 0 \]
Linear Search Algorithm

**INPUT:** A[1..n] – an array of integers, q – an integer.
**OUTPUT:** an index \( j \) such that \( A[j] = q \); 0, if \( \forall j \ (1 \leq j \leq n): A[j] \neq q \)

\[
j \leftarrow 1 \\
\textbf{while } j \leq n \textbf{ and } A[j] \neq q \\
\hspace{1cm} \textbf{do } j++ \\
\textbf{if } j \leq n \textbf{ then return } j \\
\textbf{else return } 0
\]

- The algorithm uses a *brute-force* algorithm design technique — scans the input array sequentially.
  - Linear search
- The code is written in an unambiguous format
  - Pseudo-code
  - INPUT and OUTPUT are clearly specified
Pseudo-code

• Description of algorithms
  ■ A *compact, informal but complete* high-level description
  ■ “It allows the designer to focus on the logic of the algorithm without being distracted by details of language syntax.”

• A mixture of code (using syntax building blocks from programming language) and English
  ■ Control structures (*if then else*, *while* and *for* loops)
  ■ Assignment (←, or :=)
  ■ Array element access: A[i]

• You are allowed to choose your own pseudo-code style
  ■ Two extremes
    ✤ A near-exact imitation of a real programming language
    ✤ A description approaching a formatted prose
  ■ Keep yours somewhere in-between, clean and readable!
  ■ If you use LaTeX, various packages are available for formatting algorithms
Analysis on Linear Search

**INPUT:** A[1..n] – an array of integers, q – an integer.
**OUTPUT:** an index j such that A[j] = q; 0, if \( \forall j (1 \leq j \leq n): A[j] \neq q \)

\[ j \leftarrow 1 \]
\[ \text{while } j \leq n \text{ and } A[j] \neq q \]
\[ \quad \text{do } j++ \]
\[ \text{if } j \leq n \text{ then return } j \]
\[ \text{else return } 0 \]

- Efficiency as a function of input size
  - Running time: determined how element checks in the while-loop
  - Space used: array size
- Best case: A[1]=q, running time=1, i.e., *constant* time
- Worst case: A[n]=q, running time=n, i.e., *linear* time
- Average case: q is expected to be found in the middle, running time=\( n/2 \), still *linear* time
Considerations

• Can we do better?
  ■ Linear time is not be fast enough if the size of array A is huge
  ■ E.g., treat student number as an integer and store all of AAU students in an array
    ◆ Find the student record with a given student number
    ◆ Need to search about 10 thousand records!
    ◆ This definitely is not a huge enough example. Think about CPR numbers of all legal residents in Denmark!

• We can do better if the input array is sorted!
  ■ Binary search
  ■ Idea behind
    ◆ Divide-and-conquer, one of the key design techniques
Binary Search

- We can safely ignore some array elements without missing what we are looking for.
  - The order of the array ensures those ignored elements are not qualified.

\[
\begin{array}{cccccc}
  l & j-1 & j & j+1 & \cdots & r \\
  \cdots & ? & \cdots & \cdots & \cdots & \cdots \\
\end{array}
\]

\textbf{INPUT:} \( A[1..n] \) – a sorted (non-descending) array of integers, \( q \) – an integer.
\textbf{OUTPUT:} an index \( j \) such that \( A[j] = q \); 0, if \( \forall j (1 \leq j \leq n): A[j] \neq q \)

\begin{verbatim}
left ← 1
right ← n
do
  j ← ⌊(left+right)/2⌋
  if A[j]=q then return j
  else if A[j]>q then right ← j-1
  else left ← j+1
while left <= right
return 0
\end{verbatim}
Analysis on Binary Search

• How many times the loop is executed:
  ■ With each execution the difference between left and right is cut in half
    ◆ Initially the difference is \( n \)
    ◆ The loop stops when the difference becomes 0
  ■ How many times do you have to cut \( n \) in half to get 1?
    ◆ \( \log n \): better than the brute-force approach (\( n \))
Example of Binary Search

Input: array \(a\), \(q=36\)  \(\text{(Note indexes here start from 0!)}\)

1. \((0+15)/2=7;\ a[7]=19;\) too small; search 8..15
2. \((8+15)/2=11;\ a[11]=32;\) too small; search 12..15
3. \((12+15)/2=13;\ a[13]=37;\) too large; search 12..12
4. \((12+12)/2=12;\ a[12]=32;\) too small; search 13..12...but 13>12, so quit: \(36\) not found
Agenda

• Course Organization
• Introduction
• Case Study
• Conclusion
  ■ Lecture Topics
  ■ Course Goals
  ■ Next Lecture
Lecture Topics

• Algorithm Basics (2)
  ■ Correctness
  ■ Complexity

• Fundamental Data Structures
  ■ Linear structures (2) + sorting (2)
  ■ Complex structures and their algorithms (6)

• Algorithm Design Techniques (2)
  ■ Routines or templates that can be followed and reused in solving your own problems
  ■ Disclose basic and important design techniques (like brute-force and divide-and-conquer)
  ■ Examples
  ■ Applications
Lecture Schedule

- **Lecture 2**: Correctness of algorithms.
- **Lecture 3**: Complexity of algorithms.

- **Lecture 4**: Data abstraction, Linear structures (I).
- **Lecture 5**: Linear structures (II).

- **Lecture 6**: Sorting (I).
- **Lecture 7**: Sorting (II).

- **Lecture 8**: Trees (I).
- **Lecture 9**: Trees (II).

- **Lecture 10**: Algorithm design techniques (I).
- **Lecture 11**: Algorithm design techniques (II).

- **Lecture 12**: Graphs (I).
- **Lecture 13**: Graphs (II).

- **Lecture 14**: Shortest paths in graphs.
- **Lecture 15**: Minimum spanning trees.
The Goals of This Course

- Main points to learn from this course
  - To be able to *think “algorithmically”*, to get the spirit of how algorithms are designed
  - To be equipped with a *toolbox* of *classical* data structures and algorithms
  - To learn a number of algorithm design *techniques* (like divide-and-conquer)
  - To learn reason (in a formal way) about the *correctness* and the *efficiency* of algorithms
Next Lecture

• Correctness of algorithms
  ■ Partial correctness vs. total correctness
  ■ Pre- and post-conditions
  ■ Assertions and invariants
  ■ Recursion vs. iteration
  ■ Mathematical induction