Advanced Algorithms
and Data Structures

Prof. Tapio Elomaa

Course Basics

- A new 7 credit unit course
- Replaces OHJ-2156 Analysis of Algorithms
- We take things a bit further than OHJ-2156

- We will assume familiarity with
  - Necessary mathematics
  - Elementary data structures
Basics cont’d

• There will be 4 hours of lectures per week
• Weekly exercises start in two weeks time

• We will not have a programming exercise this year (unless you demand to have one)
• We might consider organizing a seminar with voluntary presentations (yielding extra points) at the end of the course

Organization & Timetable

• Lectures: Prof. Tapio Elomaa
  – Tue & Thu 2–4 PM in TB223 (& TB219 in PII)
  – Aug. 27 – Dec. 5, 2013
    • Period break Oct. 14–20, 2013

• Exercises: M.Sc. Teemu Heinimäki

• Exam: Fri Dec. 13, 2013 at 9–12 AM
Course Grading

- **Exam**: Maximum of 30 points
- **Weekly exercises** yield extra points
  - 40% of questions answered: 1 point
  - 80% answered: 6 points
  - In between: linear scale (so that decimals are possible)
- Final grading depends on what we agree as course components

Material

- The textbook of the course is
- There is no prepared material, the slides appear in the web as the lectures proceed
- The exam is based on the lectures (i.e., not on the slides only)
Content (Plan)

I  Foundations
II  Sorting (and Order Statistics)
III Data Structures
IV  Advanced Design and Analysis Techniques
V  Advanced Data Structures
VI  Graph Algorithms
VII Selected Topics

I Foundations

The Role of Algorithms in Computing
Getting Started
Growth of Functions
Recurrences
Probabilistic Analysis and Randomized Algorithms
II Sorting (and Order Statistics)

- Heapsort
- Quicksort
- Sorting in Linear Time

III Data Structures

- Elementary Data Structures
- Hash Tables
- Binary Search Trees
- Red-Black Trees
IV Advanced Design and Analysis Techniques

Dynamic Programming
Greedy Algorithms
Amortized Analysis

V Advanced Data Structures

B-Trees
Binomial Heaps
Fibonacci Heaps
VI Graph Algorithms

Elementary Graph Algorithms
Minimum Spanning Trees
Single-Source Shortest Paths
All-Pairs Shortest Paths
Maximum Flow

VII Selected Topics

Part of these could also be student presentation topics
The sorting problem

- **Input**: A sequence of \( n \) numbers 
  \( \langle a_1, a_2, \ldots, a_n \rangle \)
- **Output**: A *permutation* (reordering) 
  \( \langle a'_1, a'_2, \ldots, a'_n \rangle \)
  of the input sequence such that 
  \( a'_1 \leq a'_2 \leq \cdots \leq a'_n \)

- The numbers that we wish to sort are also known as *keys*

---

**Insertion-Sort**(*A*)

1. `for j ← 2 to A.length`
2. `key ← A[j]`
3. // Insert \( A[j] \) into the sorted sequence \( A[1..j-1] \)
4. `i ← j-1`
5. `while i > 0 and A[i] > key`
7. `i ← i-1`
8. `A[i+1] ← key`
Correctness of the Algorithm

- The following loop invariant helps us understand why the algorithm is correct:

At the start of each iteration of the for loop of lines 1–8, the subarray $A[1..j-1]$ consists of the elements originally in $A[1..j-1]$ but in sorted order.
Initialization

- The loop invariant holds before the first loop iteration, when $j = 2$:
  - The subarray, therefore, consists of just the single element $A[1]$
  - It is in fact the original element in $A[1]$
  - This subarray is trivially sorted
  - Therefore, the loop invariant holds prior to the first iteration of the loop

Maintenance

- Each iteration maintains the loop invariant:
  - At this point the value of $A[j]$ is inserted (line 8)
  - The subarray $A[1..j]$ then consists of the elements originally in $A[1..j]$, but in sorted order
Termination

- The condition causing the for loop to terminate is that \( j > A.\text{length} = n \)
- Because each loop iteration increases \( j \) by 1, we must have \( j = n+1 \) at that time
- Substituting \( n+1 \) for \( j \) in the wording of loop invariant, we have that the subarray \( A[1..n] \) consists of the elements originally in \( A[1..n] \), but in sorted order
- \( A[1..n] \) is the entire array

Analysis of insertion sort

- The time taken by the INSERTION-SORT depends on the input:
  - sorting a thousand numbers takes longer than sorting three numbers
- Moreover, the procedure can take different amounts of time to sort two input sequences of the same size
  - depending on how nearly sorted they already are
**Input size**

- The time taken by an algorithm grows with the size of the input
- Traditional to describe the *running time* of a program as a function of the *size of its input*
- For many problems, such as sorting most natural measure for input size is the number of items in the input—i.e., the array size $n$

- For, e.g., multiplying two integers, the best measure is the total number of bits needed to represent the input in binary notation
- Sometimes, more appropriate to describe the size with two numbers rather than one
- E.g., if the input to an algorithm is a graph, the input size can be described by the numbers of vertices and edges in it
# Running time

- Running time of an algorithm on an input:
  - The number of primitive operations (“steps”) executed
- Step as machine-independent as possible
- For the moment:
  - Constant amount of time to execute each line of pseudocode
  - We assume that each execution of the $i$th line takes time $c_i$, where $c_i$ is a constant

## INSERTION-SORT($A$)

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
<th>Cost</th>
<th>Times</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$j \leftarrow 2$ to $A.length$</td>
<td>$c_1$</td>
<td>$n$</td>
</tr>
<tr>
<td>2</td>
<td>$key \leftarrow A[j]$</td>
<td>$c_2$</td>
<td>$n-1$</td>
</tr>
<tr>
<td>3</td>
<td>// Insert $A[j]$ into the sorted sequence $A[1..j]$</td>
<td>$0$</td>
<td>$n-1$</td>
</tr>
<tr>
<td>4</td>
<td>$i \leftarrow j-1$</td>
<td>$c_4$</td>
<td>$n-1$</td>
</tr>
<tr>
<td>5</td>
<td>while $i &gt; 0$ and $A[i] &gt; key$</td>
<td>$c_5$</td>
<td>$\sum_{j=2}^{n} t_j$</td>
</tr>
<tr>
<td>6</td>
<td>$A[i+1] \leftarrow A[i]$</td>
<td>$c_6$</td>
<td>$\sum_{j=2}^{n} (t_j - 1)$</td>
</tr>
<tr>
<td>7</td>
<td>$i \leftarrow i-1$</td>
<td>$c_7$</td>
<td>$\sum_{j=2}^{n} (t_j - 1)$</td>
</tr>
<tr>
<td>8</td>
<td>$A[i+1] \leftarrow key$</td>
<td>$c_8$</td>
<td>$n-1$</td>
</tr>
</tbody>
</table>
• \( t_j \) denotes the number of times the `while` loop test in line 5 is executed for that value of \( j \)
• When a `for` or `while` loop exits in the usual way, the test is executed one time more than the loop body
• Comments are not executable statements, so they take no time
• Running time of the algorithm is the sum of those for each statement executed

• To compute \( T(n) \), the running time of INSERTION-SORT on an input of \( n \) values,
  – we sum the products of the cost and times columns, obtaining
  \[
  T(n) = c_1 n + c_2 (n - 1) + c_4 (n - 1) + c_5 \sum_{j=2}^{n} t_j \\
  + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8 (n - 1)
  \]
Best case

- The best case occurs if the array is already sorted
- For each $j = 2, 3, \ldots, n$, we then find that $A[i] < key$ in line 5 when $i$ has its initial value of $j-1$
- Thus $t_j = 1$ for $j = 2, 3, \ldots, n$, and the best-case running time is

$$T(n) = c_1 n + c_2 (n-1) + c_4 (n-1) + c_5 (n-1) + c_8 (n-1)$$
$$= (c_1 + c_2 + c_4 + c_5 + c_8) n - (c_2 + c_4 + c_5 + c_8)$$

Worst case

- We can express this as $an + b$ for constants $a$ and $b$ that depend on the statement costs $c_i$
- It is a linear function of $n$
- The worst case results when the array is in reverse sorted order — in decreasing order
- We must compare each element $A[j]$ with each element in the entire sorted subarray $A[1..j-1]$, and so $t_j = j$ for $j = 2, 3, \ldots, n$
• Note that
\[ \sum_{j=2}^{n} j = \frac{n(n + 1)}{2} - 1 \]
and
\[ \sum_{j=2}^{n} (j - 1) = \frac{n(n - 1)}{2} \]
by the summation of an arithmetic series
\[ \sum_{j=1}^{n} j = \frac{n(n + 1)}{2} \]

• The worst-case running time of INSERTION-SORT is
\[
T(n) = c_1 n + c_2 (n - 1) + c_4 (n - 1) + c_5 \left( \frac{n(n + 1)}{2} - 1 \right) + c_6 \left( \frac{n(n - 1)}{2} \right) + c_7 \left( \frac{n(n - 1)}{2} \right) + c_8 (n - 1)
\]
\[
= \left( \frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2} \right) n^2 + (c_1 + \cdots + c_8) n - (c_2 + \cdots + c_8)
\]
We can express this worst-case running time as $an^2 + bn + c$ for constants $a$, $b$, and $c$ that depend on the statement costs $c_i$.

- It is a **quadratic function** of $n$.
- The **rate of growth**, or **order of growth**, of the running time really interests us.
- We consider only the leading term of a formula ($an^2$); the lower-order terms are relatively insignificant for large values of $n$.

We also ignore the leading term's coefficient, constant factors are less significant than the rate of growth in determining computational efficiency for large inputs.

- For insertion sort, we are left with the factor of $n^2$ from the leading term.
- We write that insertion sort has a worst-case running time of $\Theta(n^2)$
  (“theta of $n$-squared”)
2.3 Designing algorithms

- Insertion sort is an incremental approach: having sorted \( A[1..j-1] \), we insert \( A[j] \) into its proper place, yielding sorted subarray \( A[1..j] \)
- Let us examine an alternative design approach, known as “divide-and-conquer”
- We design a sorting algorithm whose worst-case running time is much lower
- The running times of divide-and-conquer algorithms are often easily determined

The divide-and-conquer approach

- Many useful algorithms are recursive:
  - to solve a problem, they call themselves to deal with closely related subproblems
- These algorithms typically follow a divide-and-conquer approach:
  - Break the problem into subproblems that resemble the original problem but are smaller,
  - Solve the subproblems recursively,
  - Combine these solutions to create a solution to the original problem.
The paradigm involves three steps at each level of the recursion:

1. **Divide** the problem into a number of subproblems that are smaller instances of the same problem
2. **Conquer** the subproblems by solving them recursively
   - If the sizes are small enough, just solve the subproblems in a straightforward manner
3. **Combine** the solutions to the subproblems into the solution for the original problem

### The merge sort algorithm

- **Divide**: Divide the \( n \)-element sequence into two subsequences of \( n/2 \) elements each
- **Conquer**: Sort the two subsequences recursively using merge sort
- **Combine**: Merge the two sorted subsequences to produce the sorted answer
  - Recursion “bottoms out” when the sequence to be sorted has length 1: a sequence of length 1 is already in sorted order
The key operation is the merging of two sorted sequences in the “combine” step.

We call auxiliary procedure \textsc{Merge}(A, p, q, r), where A is an array and p, q, and r are indices such that \( p \leq q < r \).

The procedure assumes that the subarrays \( A[p..q] \) and \( A[q+1..r] \) are in sorted order and merges them to form a single sorted subarray that replaces the current subarray \( A[p..r] \).

\begin{algorithm}
\textbf{Merge}(A, p, q, r)
\begin{enumerate}
\item \( n_1 \leftarrow q-p+1 \)
\item \( n_2 \leftarrow r-q \)
\item Let \( L[1..n_1+1] \) and \( R[1..n_2+1] \) be new arrays
\item for \( i \leftarrow 1 \) to \( n_1 \)
\item \( L[i] \leftarrow A[p+i-1] \)
\item for \( j \leftarrow 1 \) to \( n_2 \)
\item \( R[j] \leftarrow A[q+j] \)
\item \( L[n_1+1] \leftarrow \infty \)
\item \( R[n_2+1] \leftarrow \infty \)
\item \( i \leftarrow 1 \)
\item \( j \leftarrow 1 \)
\item for \( k \leftarrow p \) to \( r \)
\item if \( L[i] \leq R[j] \)
\item \( A[k] \leftarrow L[i] \)
\item \( i \leftarrow i+1 \)
\item else \( A[k] \leftarrow R[j] \)
\item \( j \leftarrow j+1 \)
\end{enumerate}
\end{algorithm}
• Line 1 computes the length $n_1$ of the subarray $A[p..q]$; similarly for $n_2$ and $A[q+1..r]$ on line 2
• Line 3 creates arrays $L$ and $R$ (“left” /“right”), of lengths $n_1+1$ and $n_2+1$, respectively
  – the extra position will hold the sentinel
• The for loop of lines 4–5 copies $A[p..q]$ into $L[1..n_1]$; lines 6–7 copy $A[q+1..r]$ into $R[1..n_2]$
• Lines 8–9 put the sentinels at the ends of $L$ and $R$

• Lines 10–17 perform the $r-p+1$ basic steps by maintaining the following loop invariant:
  – At the start of each iteration of the for loop of lines 12–17, $A[p..k-1]$ contains the $k-p$
    smallest elements of $L[1..n_1+1]$ and $R[1..n_2+1]$, in sorted order
  – Moreover, $L[i]$ and $R[j]$ are the smallest elements of their arrays that have not been
    copied back into $A
MERGE($A$, 9, 12, 16)
• The needed $r-p+1$ iterations of the last for loop have been executed:
  • $A[9..16]$ is sorted, and
  • the two sentinels in $L$ and $R$ are the only two elements in these arrays that have not been copied into $A$