Lecture 1

Introduction to Analysis of Algorithms
Motivation

– Procedure vs. Algorithm

– What kind of problems are solved by Algorithms?
  • determine/compare DNA sequences
  • efficiently search (e.g. Google) web pages w/ keywords
  • route data (e.g. email) on the Internet
  • decode data (e.g. banking) for security

– Data Structures & Algorithms

– Repertoire vs. New Algorithms (Techniques)
Motivation cntd

– Efficient (scope of course) vs. Inefficient

– Design algorithms that are
  • fast,
  • uses as little memory as possible, and
  • correct!
Problem: Sorting (from Section 1.1)

Input: Sequence of numbers
\( \langle a_1, a_2, \ldots, a_n \rangle \)

Output: A permutation
\( \Pi = \langle \Pi(1), \Pi(2), \ldots, \Pi(n) \rangle \)

such that
\( a_{\Pi(1)} \leq a_{\Pi(2)} \leq \ldots \leq a_{\Pi(n)} \)
Algorithm: Insertion sort (from Section 1.1)

**Insertion-Sort** (A)

1.  \( \textbf{for } j \leftarrow 2 \text{ to } n \textbf{ do} \)
2.    \( \text{key } \leftarrow A[j]; \)
3.    \( i \leftarrow j - 1; \)
4.    \( \textbf{while } i > 0 \text{ and } A[i] > \text{key} \textbf{ do} \)
5.      \( A[i+1] \leftarrow A[i]; \)
6.      \( i \leftarrow i - 1; \)
7.  \textbf{endwhile} \)
8.  \( A[i+1] \leftarrow \text{key}; \)
9.  \textbf{endfor}
Pseudocode Notation

– Liberal use of English
– Use of indentation for block structure
– Omission of error handling and other details
  • Needed in real programs
Algorithm: Insertion sort

Idea:

- Items sorted in-place
- Items rearranged within array
- At most constant number of items stored outside the array at any time
- Input array A contains sorted output sequence when Insertion-Sort is finished

Incremental approach
Algorithm: Insertion sort

Example: Sample sequence

A=⟨31, 42, 59, 26, 40, 35⟩

Assume first 5 items are already sorted in A[1..5]

A=⟨26, 31, 40, 42, 59, 35⟩

<table>
<thead>
<tr>
<th>26</th>
<th>31</th>
<th>40</th>
<th>42</th>
<th>59</th>
<th>35</th>
</tr>
</thead>
<tbody>
<tr>
<td>26</td>
<td>31</td>
<td>40</td>
<td>42</td>
<td>59</td>
<td>35</td>
</tr>
<tr>
<td>26</td>
<td>31</td>
<td>40</td>
<td>42</td>
<td>42</td>
<td>59</td>
</tr>
<tr>
<td>26</td>
<td>31</td>
<td>40</td>
<td>40</td>
<td>42</td>
<td>59</td>
</tr>
<tr>
<td>26</td>
<td>31</td>
<td>35</td>
<td>40</td>
<td>42</td>
<td>59</td>
</tr>
</tbody>
</table>

35 key
Running Time

• Depends on
  – Input size (e.g., 6 elements vs 60000 elements)
  – Input itself (e.g., partially sorted)

• Usually want upper bound
Kinds of running time analysis:

- **Worst Case** (*Usually*):
  \[ T(n) = \text{max time on any input of size } n \]
- **Average Case** (*Sometimes*):
  \[ T(n) = \text{average time over all inputs of size } n \]
  Assumes statistical distribution of inputs
- **Best Case** (*Rarely*):
  BAD*: Cheat with slow algorithm that works fast on some inputs
  GOOD: Only for showing bad lower bound

*Can modify any algorithm (almost) to have a low best-case running time
  - Check whether input constitutes an output at the very beginning of the algorithm
Running Time

• For Insertion-Sort, what is its \textbf{worst-case time}.
  – Depends on speed of primitive operations
    • Relative speed (on same machine)
    • Absolute speed (on different machines)

• Asymptotic analysis
  – Ignore machine-dependent constants
  – Look at growth of $T(n)$ as $n \rightarrow \infty$
\( \Theta \) Notation

• Drop low order terms
• Ignore leading constants

E.g. \( 3n^3 + 90n^2 - 2n + 5 = \Theta(n^3) \)
• As $n$ gets large a $\Theta(n^2)$ algorithm runs faster than a $\Theta(n^3)$ algorithm
Running Time Analysis of Insertion-Sort

• Sum up costs:

\[ T(n) = c_1 n + c_2 (n - 1) + c_3 (n - 1) + c_4 \sum_{j=2}^{n} t_j + c_5 \sum_{j=2}^{n} (t_j - 1) + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 (n - 1) \]

• The best case (sorted order):

\[ T(n) = (c_1 + c_2 + c_3 + c_4 + c_7)n - (c_2 + c_3 + c_4 + c_7) \]

• The worst case (reverse sorted order):

\[ T(n) = \frac{1}{2} (c_4 + c_5 + c_6)n^2 + (c_1 + c_2 + c_3 + \frac{1}{2} (c_4 + c_5 + c_6) + c_7)n - (c_2 + c_3 + c_4 + c_7) \]
Running Time Analysis of Insertion-Sort

- **Worst-case (input reverse sorted)**
  - Inner loop is $\Theta(j)$
    \[
    T(n) = \sum_{j=2}^{n} \Theta(j) = \Theta\left(\sum_{j=2}^{n} j\right) = \Theta(n^2)
    \]

- **Average case (all permutations equally likely)**
  - Inner loop is $\Theta(j/2)$
    \[
    T(n) = \sum_{j=2}^{n} \Theta\left(j/2\right) = \sum_{j=2}^{n} \Theta\left(j\right) = \Theta(n^2)
    \]
    
    Often, average case not much better than worst case

- **Is this a fast sorting algorithm?**
  - Yes, for small $n$. No, for large $n.$
Algorithm: Merge-Sort

• Basic Step: Merge 2 sorted lists of total length $n$ in $\Theta(n)$ time

• Example:

\[
\begin{array}{cccc}
2 & 3 & 7 & 8 \\
1 & 4 & 5 & 6 \\
\end{array}
\}
\begin{array}{cccc}
1 & 2 & 3 & 4 & 5 & 6 & \ldots \\
\end{array}
\]
Recursive Algorithm:

**Merge-Sort** \((A,p,r)\)

\[T(n)\]

if \(p = r\) then return; \((\Theta(1))\)

else

\[q \leftarrow \lfloor (p+r)/2 \rfloor;\]

: Divide \((\Theta(1))\)

Merge-Sort(A,p,q);

: Conquer \((T(n/2))\)

Merge-Sort(A,q+1,r);

: Conquer \((T(n/2))\)

Merge(A,p,q,r);

: Combine \((\Theta(n))\)

endif

• Call **Merge-Sort**\((A,1,n)\) to sort \(A[1..n]\)

• Recursion bottoms up when subsequences have length 1
Recurrence (for Merge-Sort) - From Section 1.3

• Describes a function recursively in terms of itself
• Describes performance of recursive algorithms

For Merge-Sort

\[ T(n) = \begin{cases} 
\Theta(1) & \text{if } n=1 \\
2T(n/2) + \Theta(n) & \text{otherwise}
\end{cases} \]
• How do we find a good upper bound on $T(n)$ in closed form?

• Generally, will assume $T(n) = \text{Constant (}$Θ$(1))$ for sufficiently small $n$

• For **Merge-Sort** write the above recurrence as

$$T(n) = 2 \cdot T(n/2) + \Theta(n)$$

• Solution to the recurrence

$$T(n) = \Theta(n \log n)$$
Conclusions (from Section 1.3)

• $\Theta(n \log n)$ grows more slowly than $\Theta(n^2)$

Therefore Merge-Sort beats Insertion-Sort in the worst case

• In practice, Merge-Sort beats Insertion-Sort for $n > 30$ or so.