Lecture 1
Introduction to Analysis of Algorithms

View in slide-show mode
Grading

- Midterm: 20%
- Final: 20%
- Classwork: 54%
- Attendance: 6%
Classwork (54% of the total grade)

- Like small exams, covering the most recent material
- There will be 7 classwork sessions
- Thursdays: 17:40 – 19:30?
- Open book (clean and unused). No notes. No slides.
- See the syllabus for details.
Algorithm Definition

- **Algorithm**: A sequence of computational steps that transform the input to the desired output.

- **Procedure vs. algorithm**
  - *An algorithm must halt within finite time with the right output*.

- **Example**:

  a sequence of \( n \) numbers \[\xrightarrow{\text{Sorting Algorithm}}\] \( n \) numbers in sorted permutation of input sequence
Many Real World Applications

- **Bioinformatics**
  - Determine/compare DNA sequences
- **Internet**
  - Manage/manipulate/route data
- **Information retrieval**
  - Search and access information in large data
- **Security**
  - Encode & decode personal/financial/confidential data
- **Computer Aided Design**
  - Minimize human effort in chip-design process
Course Objectives

- Learn basic algorithms & data structures
- Gain skills to design new algorithms
- Focus on efficient algorithms
- Design algorithms that
  - are fast
  - use as little memory as possible
  - are correct!
Outline of Lecture 1

- Study two sorting algorithms as examples
  - Insertion sort: *Incremental* algorithm
  - Merge sort: *Divide-and-conquer*

- Introduction to runtime analysis
  - Best vs. worst vs. average case
  - Asymptotic analysis
Sorting Problem

**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

\[ a_{\Pi(1)} \leq a_{\Pi(2)} \leq \ldots \leq a_{\Pi(n)} \]
Insertion Sort
Insertion Sort: Basic Idea

- Assume input array: A[1..n]
- Iterate j from 2 to n

![Diagram of insertion sort process]

- already sorted
- insert into sorted array
- sorted subarray
Pseudo-code notation

- Objective: Express algorithms to humans in a clear and concise way

- Liberal use of English

- Indentation for block structures

- Omission of error handling and other details

\[\rightarrow \text{needed in real programs}\]
Algorithm: Insertion Sort (from Section 2.2)

Insertion-Sort (A)

1. for $j \leftarrow 2$ to $n$ do
2. \hspace{1em} key $\leftarrow A[j]$;
3. \hspace{1em} $i \leftarrow j - 1$;
4. \hspace{1em} while $i > 0$ and $A[i] >$ key do
5. \hspace{2em} $A[i+1] \leftarrow A[i]$;
6. \hspace{1em} $i \leftarrow i - 1$;
7. \hspace{1em} endwhile
8. \hspace{1em} $A[i+1] \leftarrow key$;
9. endfor
## Algorithm: Insertion Sort

**Insertion-Sort (A)**

1. for $j \leftarrow 2$ to $n$ do
2.     key $\leftarrow A[j]$;
3.     $i \leftarrow j - 1$;
4.     while $i > 0$ and $A[i] > key$ do
5.         $A[i+1] \leftarrow A[i]$;
6.         $i \leftarrow i - 1$;
endwhile
7.     $A[i+1] \leftarrow key$;
endfor

**Loop invariant:**

The subarray $A[1..j-1]$ is always sorted.

Iterate over array elts $j$
Algorithm: Insertion Sort

**Insertion-Sort (A)**

1. for $j \leftarrow 2$ to $n$ do
2.   key $\leftarrow A[j]$;
3.   $i \leftarrow j - 1$;
4.   while $i > 0$ and $A[i] > key$ do
5.     $A[i+1] \leftarrow A[i]$;
6.     $i \leftarrow i - 1$;
7.   endwhile
8.   $A[i+1] \leftarrow key$;
endfor

Shift right the entries in $A[1..j-1]$ that are $> key$
Algorithm: Insertion Sort

**Insertion-Sort (A)**

1. for $j \leftarrow 2$ to $n$ do
2. \hspace{1em} key \leftarrow A[j];
3. \hspace{1em} $i \leftarrow j - 1$;
4. \hspace{1em} while $i > 0$ and $A[i] > key$ do
5. \hspace{2em} $A[i+1] \leftarrow A[i]$;
6. \hspace{2em} $i \leftarrow i - 1$;
7. \hspace{1em} endwhile
8. \hspace{1em} $A[i+1] \leftarrow key$;

endfor

Insert key to the correct location

*End of iter $j$: $A[1..j]$ is sorted*
Insertion Sort - Example

Insertion-Sort (A)
1. for j ← 2 to n do
2. key ← A[j];
3. i ← j - 1;
4. while i > 0 and A[i] > key do
5. A[i+1] ← A[i];
6. i ← i - 1;
    endwhile
7. A[i+1] ← key;
endfor
Insertion Sort - Example: Iteration j=2

Insertion-Sort (A)

1. for j ← 2 to n do
2.   key ← A[j];
3.   i ← j - 1;
4.   while i > 0 and A[i] > key do
5.     A[i+1] ← A[i];
6.     i ← i - 1;
   endwhile
7.   A[i+1] ← key;
endfor
Insertion Sort - Example: Iteration $j=3$

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

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

What are the entries at the end of iteration $j=3$?
Insertion Sort - Example: Iteration j=3

**Insertion-Sort** (A)

1. for j ← 2 to n do
2. key ← A[j];
3. i ← j - 1;
4. while i > 0 and A[i] > key do
5. A[i+1] ← A[i];
6. i ← i - 1;
endwhile
7. A[i+1] ← key;
endfor
Insertion Sort - Example: Iteration j=4

Insertion-Sort (A)

1. for j ← 2 to n do
2.   key ← A[j];
3.   i ← j - 1;
4.   while i > 0 and A[i] > key do
5.     A[i+1] ← A[i];
6.     i ← i - 1;
   endwhile
7.   A[i+1] ← key;
endfor
Insertion Sort - Example: Iteration j=5

Insertion-Sort (A)
1. for j ← 2 to n do
2. key ← A[j];
3. i ← j - 1;
4. while i > 0 and A[i] > key do
5. A[i+1] ← A[i];
6. i ← i - 1;
   endwhile
7. A[i+1] ← key;
endfor

What are the entries at the end of iteration j=5?
Insertion Sort - Example: Iteration j=5

**Insertion-Sort (A)**

1. for j ← 2 to n do
2.   key ← A[j];
3.   i ← j - 1;
4.   while i > 0 and A[i] > key do
5.     A[i+1] ← A[i];
6.     i ← i - 1;
endwhile
7.   A[i+1] ← key;
endfor
Insertion Sort - Example: Iteration j=6

Insertion-Sort (A)
1. for j ← 2 to n do
2. key ← A[j];
3. i ← j - 1;
4. while i > 0 and A[i] > key do
5. A[i+1] ← A[i];
6. i ← i - 1;
endwhile
7. A[i+1] ← key;
endfor
Insertion Sort Algorithm - Notes

- **Items sorted in-place**
  - Elements rearranged within array
  - At most constant number of items stored outside the array at any time (e.g. the variable *key*)
  - Input array *A* contains sorted output sequence when the algorithm ends

- **Incremental approach**
  - Having sorted *A*[1..j-1], place *A*[j] correctly so that *A*[1..j] is sorted
Running Time

- Depends on:
  - Input size (e.g., 6 elements vs 6M 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)*
  \[ T(n) = \text{min time on any input of size } n \]
  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 **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 $n \to \infty$
Θ Notation

- Drop low order terms
- Ignore leading constants

  e.g.

\[2n^2 + 5n + 3 = \Theta(n^2)\]

\[3n^3 + 90n^2 - 2n + 5 = \Theta(n^3)\]

- Formal explanations in the next lecture.
• As $n$ gets large, a $\Theta(n^2)$ algorithm runs faster than a $\Theta(n^3)$ algorithm.
Insertion Sort – Runtime Analysis

**Cost** | **Insertion-Sort (A)**
---|---
$c_1$ | 1. for $j \leftarrow 2$ to $n$ do
$c_2$ | 2. key $\leftarrow A[j]$;
$c_3$ | 3. $i \leftarrow j - 1$;
$c_4$ | 4. while $i > 0$ and $A[i] >$ key do
$c_5$ | 5. $A[i+1] \leftarrow A[i]$;
$c_6$ | 6. $i \leftarrow i - 1$;
$c_7$ | 7. $A[i+1] \leftarrow$ key;

endfor

$t_j$: The number of times while loop test is executed for $j$
How many times is each line executed?

<table>
<thead>
<tr>
<th># times</th>
<th>Insertion-Sort (A)</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>1. for j ← 2 to n do</td>
</tr>
<tr>
<td>n−1</td>
<td>2. key ← A[j];</td>
</tr>
<tr>
<td>n−1</td>
<td>3. i ← j - 1;</td>
</tr>
<tr>
<td>k₄</td>
<td>4. while i &gt; 0 and A[i] &gt; key do</td>
</tr>
<tr>
<td>k₅</td>
<td>5. A[i+1] ← A[i];</td>
</tr>
<tr>
<td>k₆</td>
<td>6. i ← i - 1;</td>
</tr>
<tr>
<td>n−1</td>
<td>7. A[i+1] ← key;</td>
</tr>
<tr>
<td></td>
<td><strong>endfor</strong></td>
</tr>
</tbody>
</table>

\[
k_4 = \sum_{j=2}^{n} t_j \\
k_5 = \sum_{j=2}^{n} (t_j - 1) \\
k_6 = \sum_{j=2}^{n} (t_j - 1)
\]
Insertion Sort – Runtime Analysis

- Sum up costs:

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

- What is the **best case** runtime?

- What is the **worst case** runtime?
**Question:** If $A[1...j]$ is already sorted, $t_j = ?$

**Insertion-Sort (A)**

1. **for** $j \leftarrow 2$ **to** $n$ **do**
2. \hspace{0.5cm} **key** $\leftarrow A[j]$;
3. \hspace{0.5cm} $i \leftarrow j - 1$;
4. \hspace{0.5cm} **while** $i > 0$ **and** $A[i] > \text{key}$ **do**
5. \hspace{1.5cm} $A[i+1] \leftarrow A[i]$;
6. \hspace{1.5cm} $i \leftarrow i - 1$;
7. \hspace{1.5cm} **endwhile**
8. \hspace{0.5cm} $A[i+1] \leftarrow \text{key}$;
9. **endfor**

$A = [2, 4, 5, 6, 1, 3]$ is sorted. $j = 6$, $\text{key} = 6$.

$t_j = 1$
Insertion Sort – Best Case Runtime

- Original function:

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

- Best-case: Input array is already sorted

\[
t_j = 1 \text{ for all } j
\]

\[
T(n) = (c_1 + c_2 + c_3 + c_4 + c_7)n - (c_2 + c_3 + c_4 + c_7)
\]
Q: If A[j] is smaller than every entry in A[1..j-1], t_j = ?

**Insertion-Sort (A)**

1. for j ← 2 to n do
2.   key ← A[j];
3.   i ← j - 1;
4.   while i > 0 and A[i] > key do
5.     A[i+1] ← A[i];
6.     i ← i - 1;
7.   endwhile
8.   A[i+1] ← key;
9. endfor
Insertion Sort – Worst Case Runtime

- **Worst case**: The input array is reverse sorted
  \[ t_j = j \text{ for all } j \]

- **After derivation, worst case runtime**:
  \[
  T(n) = \frac{1}{2} (c_4 + c_5 + c_6) n^2 + \\
  (c_1 + c_2 + c_3 + \frac{1}{2} (c_4 c_5 c_6) + c_7) n (c_2 + c_3 + c_4 + c_7)
  \]
Insertion Sort – Asymptotic Runtime Analysis

**Insertion-Sort** (A)

1. **for** \( j \leftarrow 2 \) **to** \( n \) **do**
2. \( \text{key} \leftarrow A[j]; \)
3. \( i \leftarrow j - 1; \) \( \Theta(1) \)
4. **while** \( i > 0 \) **and** \( A[i] > \text{key} \) **do**
5. \( A[i+1] \leftarrow A[i]; \) \( \Theta(1) \)
6. \( i \leftarrow i - 1; \)
7. **endwhile**
8. \( A[i+1] \leftarrow \text{key}; \) \( \Theta(1) \)
9. **endfor**
Asymptotic Runtime 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* $\Theta(j/2)$
    
    $$T(n) = \sum_{j=2}^{n} \Theta(j/2) = \sum_{j=2}^{n} \Theta(j) = \Theta(n^2)$$

  - Often, average case not much better than worst case

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

Input array A

Divide

sort this half

Conquer

sort this half

Combine

merge two sorted halves
**Merge-Sort** (A, p, r)

if \( p = r \) then return;
else

\[ q \leftarrow \lfloor \frac{p+r}{2} \rfloor; \quad \text{(Divide)} \]

Merge-Sort (A, p, q);

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

Merge (A, p, q, r);

endif

- Call **Merge-Sort**(A,1,n) to sort A[1..n]
- Recursion bottoms out when subsequences have length 1
Merge Sort: Example

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

if \(p = r\) then
  return
else
  \(q \leftarrow \lfloor (p+r)/2 \rfloor\)

  \[ \text{Merge-Sort} \ (A, p, q) \]
  \[ \text{Merge-Sort} \ (A, q+1, r) \]

  Merge\((A, p, q, r)\)
endif
How to merge 2 sorted subarrays?

- HW: Study the pseudo-code in the textbook (Sec. 2.3.1)
- What is the complexity of this step? $\Theta(n)$
Merge Sort: Correctness

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

if p = r then
    return
else
    q ← ⌊ (p+r)/2 ⌋
    Merge-Sort (A, p, q)
    Merge-Sort (A, q+1, r)

endif

**Base case:** p = r
→ Trivially correct

**Inductive hypothesis:** MERGE-SORT is correct for any subarray that is a strict (smaller) *subset* of A[p, r].

**General Case:** MERGE-SORT is correct for A[p, r].
→ From inductive hypothesis and correctness of *Merge*. 

Cevdet Aykanat and Mustafa Ozdal
Computer Engineering Department, Bilkent University
Merge-Sort: Complexity

\[
\text{Merge-Sort} \ (A, p, r) \quad \rightarrow \quad T(n)
\]

\[
\text{if } p = r \text{ then}
\]

\[
\rightarrow \quad \Theta(1)
\]

\[
\text{return}
\]

\[
\rightarrow \quad \Theta(1)
\]

\[
\text{else}
\]

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

\[
\rightarrow \quad \Theta(1)
\]

\[
\text{Merge-Sort} \ (A, p, q) \quad \rightarrow \quad T(n/2)
\]

\[
\text{Merge-Sort} \ (A, q+1, r) \quad \rightarrow \quad T(n/2)
\]

\[
\text{Merge}(A, p, q, r) \quad \rightarrow \quad \Theta(n)
\]

\[
\text{endif}
\]
Merge Sort – Recurrence

- Describe a function recursively in terms of itself
- To analyze the 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 to solve for T(n)?

\[ T(n) = \begin{cases} 
\Theta(1) & \text{if } n=1 \\
2T(n/2) + \Theta(n) & \text{otherwise}
\end{cases} \]

- Generally, we will assume \( T(n) = \Theta(1) \) for sufficiently small \( n \).
- The recurrence above can be rewritten as:

\[ T(n) = 2T(n/2) + \Theta(n) \]

- How to solve this recurrence?
Solve Recurrence: \( T(n) = 2T \left( \frac{n}{2} \right) + \Theta(n) \)
Solve Recurrence: \( T(n) = 2T \left( \frac{n}{2} \right) + \Theta(n) \)
Solve Recurrence: \( T(n) = 2T \left( \frac{n}{2} \right) + \Theta(n) \)
Merge Sort Complexity

- **Recurrence:**
  \[ T(n) = 2T(n/2) + \Theta(n) \]

- **Solution to recurrence:**
  \[ T(n) = \Theta(n \log n) \]
Conclusions: **Insertion Sort vs. Merge Sort**

- \( \Theta(n\log n) \) 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.