CS2351
Data Structures

Lecture 1: Getting Started
About this lecture

• Study some sorting algorithms
  - Insertion Sort
  - Selection Sort
  - Merge Sort
• Show why these algorithms are correct
• Analyze the efficiency of the algorithms
The Sorting Problem

Input: A list of $n$ numbers
Output: Arrange the numbers in increasing order

Remark: Sorting has many applications.
E.g., if the list is already sorted, we can search a number in the list faster
Insertion Sort

• Operates in $n$ rounds
• At the $k^{th}$ round,

Swap towards left side; Stop until seeing an item with a smaller value.

Question: Why is this algorithm correct?
Selection Sort

- Operates in \( n \) rounds
- At the \( k^{th} \) round,
  - Find minimum item after \((k-1)^{th}\) position
  - Let’s call this minimum item \( X \)
  - Insert \( X \) at \( k^{th} \) position in the list

**Question:** Why is this algorithm correct?
Divide and Conquer

- Divide a big problem into smaller problems
  - solve smaller problems separately
  - combine the results to solve original one
- This idea is called **Divide-and-Conquer**
- Smart idea to solve complex problems *(why?)*
- Can we apply this idea for sorting?
Divide-and-Conquer for Sorting

• What is a smaller problem?
  ➔ E.g., sorting fewer numbers
  ➔ Let’s divide the list to two shorter lists

• Next, solve smaller problems (how?)

• Finally, combine the results
  ➔ “merging” two sorted lists into a single sorted list (how?)
Merge Sort

- The previous algorithm, using divide-and-conquer approach, is called Merge Sort.
- The key steps are summarized as follows:
  Step 1. Divide list to two halves, A and B
  Step 2. Sort A using Merge Sort
  Step 3. Sort B using Merge Sort
  Step 4. Merge sorted lists of A and B

Question: Why is this algorithm correct?
Analyzing the Running Times

• Which of previous algorithms is the best?
• Compare their running time on a computer
  - But there are many kinds of computers !!!

Standard assumption: Our computer is a RAM (Random Access Machine), so that
  - each arithmetic (such as +, −, ×, ÷), memory access,
    and control (such as conditional jump, subroutine call, return) takes constant amount of time
Analyzing the Running Times

• Suppose that our algorithms are now described in terms of RAM operations
  ➔ we can count # of each operation used
  ➔ we can measure the running time!

• Running time is usually measured as a function of the input size
  - E.g., n in our sorting problem
Insertion Sort (Running Time)

Below is a pseudo-code for Insertion Sort:

$\text{Insertion-Sort}(A)$
$\begin{align*}
1. & \quad \text{for } j = 2 \text{ to } \text{length}[A] \{ \\
1.1 & \quad \text{Compare } A_j \text{ with } A_{j-1}, A_{j-2}, \ldots \\
& \quad \quad \text{until getting } A_x \text{ smaller than } A_j; \\
1.2 & \quad \text{Insert } A_j \text{ after } A_x; \\
\} \\
\end{align*}$

Note: Steps 1.1 and 1.2 can be described in terms of RAM operations. Can you do that?
Insertion Sort (Running Time)

- Let $T(n)$ denote the running time of insertion sort, on an input of size $n$
- Suppose $t_j$ denotes the number of comparisons in round $j$
- By combining terms, we have
  \[ T(n) = c_1(n-1) + c_2 \sum t_j \]
- The values of $t_j$ are dependent on the input (not the input size)
Insertion Sort (Running Time)

- **Best Case:**
  The input list is sorted, so that all $t_j = 1$
  Then, $T(n) = c_1(n-1) + c_2(n-1)$
  \[= Kn + c \rightarrow \text{linear function of } n\]

- **Worst Case:**
  The input list is sorted in *decreasing* order, so that all $t_j = j-1$
  Then, $T(n) = K_1n^2 + K_2n + K_3$
  \[\rightarrow \text{quadratic function of } n\]
Worst-Case Running Time

- In our course (and in most CS research), we concentrate on worst-case time
- Some reasons for this:
  1. Gives an upper bound of running time
  2. Worst case occurs fairly often

Remark: Some people also study average-case running time (they assume input is drawn according to some distribution)
Practice Implementation

• When we understand completely about an algorithm, it is easy to implement it using any programming language (such as C, C++, java)

• Can we write Insertion Sort in C?
**Merge Sort** (Running Time)

The following is a partial pseudo-code for Merge Sort.

```java
MERGE-SORT(A, p, r)
1  if p < r
2      then q ← ⌊(p + r)/2⌋
3      MERGE-SORT(A, p, q)
4      MERGE-SORT(A, q + 1, r)
5      MERGE(A, p, q, r)
```

The subroutine MERGE(A,p,q,r) is missing.

Can you complete it?

Hint: Create a temp array for merging
Merge Sort (Running Time)

• Let $T(n)$ denote the running time of merge sort, on an input of size $n$.
• Suppose we know that $\text{Merge( )}$ of two lists of total size $n$ runs in $c_1n$ time.
• Then, we can write $T(n)$ as:
  \[ T(n) = 2T(n/2) + c_1n + c_2 \quad \text{when } n > 1 \]
  \[ T(n) = c_3 \quad \text{when } n = 1 \]
• Solving the recurrence, we have
• $T(n) = K_1 n \log n + K_2 n + K_3$.
Which Algorithm is Faster?

• Unfortunately, we still cannot tell
  - since constants in running times are unknown

• But we do know that if $n$ is VERY large, worst-case time of Merge Sort must be smaller than that of Insertion Sort

• Merge Sort is asymptotically faster than Insertion Sort
Practice Implementation

• Can we write Merge Sort in C?

• How about Selection Sort?
  - What is its running time in terms of n?
  - Can we write Selection Sort in C?