CS4311
Design and Analysis of Algorithms

Lecture 1: Getting Started
About this lecture

• Study a few simple algorithms for sorting
  - Insertion Sort
  - Selection Sort
  - Merge Sort
• Show why these algorithms are correct
• Try to analyze the efficiency of these algorithms (how fast they run)
The Sorting Problem

• Input: A list of \( n \) numbers
• Output: Arrange the numbers in increasing order

Remark: Sorting has many applications. As we have seen before, if the list is already sorted, we can search a number in the list faster.
Insertion Sort

• Operates in n rounds
• At the k\textsuperscript{th} round,
  - Pick up the k\textsuperscript{th} element (let’s call it X)
  - Compare X with the elements on its left, starting with the (k-1)\textsuperscript{th} element, the (k-2)\textsuperscript{th} element, and so on, until we see some element (let’s call it Y) not larger than X
  - Insert X after Y in the list
    • if no Y is found, insert X at the beginning of the list

Question: Why is this algorithm correct?
Selection Sort

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

Question: Why is this algorithm correct?
Divide and Conquer

• A good idea to solve a complicated problem is: Divide it into smaller problems, and see if we can combine the result of the smaller problems to solve the original one.
• This idea is called **Divide-and-Conquer**.
• Can we apply this idea for sorting?

• Suppose we don’t know how to sort n numbers, but we know how to sort a fewer of them (say, n/2 numbers). Can this help?
Merge Sort

• **Observation:** If we have two sorted lists $A$ and $B$, we can “merge” them into a single sorted list (how?)

• **Merge Sort** applies the divide-and-conquer idea to sort a list as follows:

  Step 1. Divide the list into two halves, $A$ and $B$
  Step 2. Sort $A$ using Merge Sort (solving a smaller problem now)
  Step 3. Sort $B$ using Merge Sort
  Step 4. Merge the sorted lists of $A$ and $B$
Analyzing the Running Times

• Which of previous algorithms is the best?
• Compare the time a computer needs to run these algorithms
  - But there are many kinds of computers !!!
• Let us assume our computer is a RAM (random access machine), so that
  - each arithmetic (such as $+$, $-$, $\times$, $\div$), memory read/write, and control (such as conditional jump, subroutine call, return) takes constant amount of time
Analyzing the Running Times

• Now, suppose that our algorithms are described in terms of these arithmetic/memory/control operations.

• Then given an input, its running time can be analyzed!

• One more point: we normally want to know the trend of how our algorithm performs on different input... The running time is usually a function of the input size (e.g., \( n \) in our sorting problem)
Insertion Sort (Running Time)

The following is a pseudo-code for Insertion Sort. Note that each line requires constant basic operations.

```
int Insertion-Sort(A)
1 for j ← 2 to length[A]
2   do 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
6            do A[i + 1] ← A[i]
7                i ← i - 1
8         A[i + 1] ← key
```

cost   times
---   ---
c1   n

c2   n - 1

c3   n - 1

c4   n - 1

c5   \( \sum_{j=2}^{n} t_j \)

c6   \( \sum_{j=2}^{n} (t_j - 1) \)

c7   \( \sum_{j=2}^{n} (t_j - 1) \)

c8   n - 1

\( t_j = \# \) of times key is compared at round j
Insertion Sort (Running Time)

• Let $T(n)$ denote the running time of insertion sort, on an input of size $n$

• By combining terms, we have

$$T(n) = c_1n + (c_2+c_4+c_8)(n-1) + c_5 \sum t_j + (c_6+c_7) \sum (t_j - 1)$$

• 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, \( T(n) = c_1 n + (c_2+c_4+c_5+c_8)(n-1) \)
\[ = Kn + c \quad \Rightarrow \text{linear function of } n \]

• **Worst Case:**
The input list is sorted in decreasing order, so that all \( t_j = j-1 \)
Then, \( T(n) = K_1 n^2 + K_2 n + K_3 \)
\[ \Rightarrow \text{quadratic function of } n \]
Worst-Case Running Time

The analysis of most algorithms in our course (and in fact, in algorithm research) concentrates on worst-case running time.

Some reasons for this:
1. Gives an upper bound of running time
2. Worst case occurs fairly often in some problem

Remark: Some people also study the average case running time (assume input is drawn randomly).
Try this at home

• Can you write down the pseudo-code for Selection Sort similarly?

• What is its running time in the worst case?
Merge Sort (Running Time)

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

```
MERGE-SORT(A, p, r)
1    if p < r
2       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: You will need to create a temporary array [Solution: textbook page 29]
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 $c_1 n$ time
• Then, we can write $T(n)$ as:
  $$T(n) = 2T(n/2) + c_1 n + c_2 \quad \text{when } n > 1$$
  $$T(n) = c_3 \quad \text{when } n = 1$$
• 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
  - Because the constants in the running times are unknown

• However, we do know that if $n$ is sufficiently large, worst-case running time of Merge Sort must become smaller than that of Insertion Sort

• We say: Merge Sort is asymptotically faster than Insertion Sort