The theory side of this course will cover:

- **Mathematical Preliminaries**
  - Asymptotic Growth Rate of Functions
  - Recurrences
  - Induction Proofs
- **Standard ADTs**
  - Elementary Data Structures (Stacks, Queues, Lists)
  - Hash Tables
  - Binary Search Trees
  - Red-Black Trees
  - Disjoint Sets
  - Graphs
- Algorithms associated with these ADTs
- Time Complexity Analysis of these Algorithms

(2.1) Some Sorting Algorithms

A typical problem associated with lists is sorting. Let our list be stored in an array `A` with indices ranging from 1 to \( n = \text{length}[A] \).

We write `A[i..j]` the subarray from index `i` to index `j`. If `i > j` we understand this to be an empty array (length 0.) The full array is then `A[1..n]`.
**Insertion Sort** \( A \)

1. **for** \( i \leftarrow 2 \) **to** \( n \)
2. \( tmp \leftarrow A[i] \)
3. \( i \leftarrow (i - 1) \)
4. **while** \( i > 0 \) \& \( A[i] > tmp \)
5. \( A[i+1] \leftarrow A[i] \)
6. \( i \leftarrow (i - 1) \)
7. \( A[i+1] \leftarrow tmp \)

(Read Pseudo-code Conventions P. 19-20.)

On the \( i \)th iteration of loop 2-7 the subarray \( A[1 \ldots (i-1)] \) is sorted, while \( A[i \ldots n] \) is unsorted. Steps 3-7 insert \( A[i] \) into the sorted subarray \( A[1 \ldots (i-1)] \).

\[ A_1, \ldots, A_{i-1}, A_i, \ldots, A_n \]

**Ex.**

\[
\begin{array}{ccccccccc}
5 & 4 & 7 & 2 & 6 \\
3 & 5 & 4 & 7 & 2 & 6 \\
1 & 3 & 5 & 4 & 7 & 2 & 6 \\
1 & 3 & 4 & 5 & 7 & 2 & 6 \\
1 & 3 & 4 & 5 & 7 & 2 & 6 \\
1 & 2 & 3 & 4 & 5 & 6 & 7 \\
1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\end{array}
\]
2.2 Analysis

We wish to find the run time of this algorithm as a function of the input size \( n \). We should make this analysis as much as possible machine independent.

Assume Step 1 takes time \( c_k \), and that the while loop test (Line 4) executes \( t_j \) times on the \( j \)th execution of loop 1–7. The total run time is then

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

\[
+ c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 (n-1)
\]

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

\[
+ (c_5 + c_6 - c_2 - c_3 - c_7)
\]

We see that \( T(n) \) depends on the numbers \( t_j \), which depend on the particular input list.

In best case, the list is already sorted, so \( t_j = 1 \) (\( 2 \leq j \leq n \)) and

\[
\sum_{i=2}^{n} t_i = n - 1
\]
Therefore

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

A more useful analysis concerns the worst case, which occurs when the list is sorted in reverse order. In this case \( t_j = j \) \( (2 \leq j \leq n) \), so

\[ \sum_{j=2}^{n} t_j = \left( \sum_{j=1}^{n} j \right) - 1 = \frac{n(n+1)}{2} - 1 \]

And hence

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

\[ - (c_2 + c_3 + c_4 + c_7) \]

To analyse the average case we assume all inputs of a given size \( n \) are equally likely. This suggests that, on average, half the elements in \( A[1 \ldots (i-1)] \) are less than \( A[i] \), and half are greater.

Thus, on average \( t_j = \frac{j}{2} \) \( (2 \leq j \leq n) \), so

\[ \sum_{j=2}^{n} t_j = \frac{1}{2} \left( \sum_{j=1}^{n} j \right) - 1 = \frac{1}{4} n^2 + \frac{1}{4} n - \frac{1}{2} \]
AND SO

\[ T(n) = \frac{1}{4} (c_4 + c_5 + c_6) n^4 + (c_1 + c_3 + c_5 - \frac{3}{4} c_4 - \frac{3}{4} c_6 + c_7) n^3 + (-c_2 - c_3 - \frac{c_4}{2} + \frac{c_5}{2} + \frac{c_6}{2} - c_7) n^2 + (c_0 + c_1 + c_5) n + c_0 \]

<table>
<thead>
<tr>
<th>Case</th>
<th>( T(n) )</th>
<th>( \Theta(n) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Best</td>
<td>( an + b )</td>
<td>( \Theta(n) )</td>
</tr>
<tr>
<td>Worst</td>
<td>( cn^2 + dn + e )</td>
<td>( \Theta(n^2) )</td>
</tr>
<tr>
<td>Average</td>
<td>( fn^2 + gn + h )</td>
<td>( \Theta(n^2) )</td>
</tr>
</tbody>
</table>

The constants \( a-h \) depend on the particular computing device used. We seek a measure of running time which is independent of choice of machine.

The required measure is called the **asymptotic growth rate** of \( T(n) \). It is a measure of how \( T(n) \) "scales up" with \( n \).

Consider four algorithms A-D whose run time on input of size \( n \) are

\[
\begin{align*}
A & : n^2 \\
B & : 10n^2 \\
C & : 10n^2 + 2n + 100 \\
D & : 1000n + 10,000
\end{align*}
\]

\( \Theta(n^2) \)
D is superior for large $n$, and worst for small $n$. $A, B, C$ are classified as equivalent. The lower order terms in $C$ are negligible for large $n$, and $A, B$ can be equalized by running $B$ on a machine which is 10 times faster.

Returning to Insertion Sort, since constant $a-h$ (and hence $c_1-c_7$) are not critical, we make no effort to calculate them explicitly. Instead we pick a representative basic operation (sometimes called a parameter) and count the number of times it is executed on inputs of fixed size $n$.

In sorting algorithms it is customary to count the number of array comparisons performed, i.e. line 4 of Insertion Sort.

Exercise:
Show that Insertion Sort does $n-1$, $n(n-1)/2$, $n(n-1)/4$ comparisons in best, worst, and average cases, on input arrays of length $n$. 