COMP 3711H
Lecture 1: Introduction
Computational Problems and Algorithms

- A **computational problem** is a specification of the desired input-output relationship.
- An **instance** of a problem is all the inputs needed to compute a solution to the problem.
- An **algorithm** is a well defined computational procedure that transforms inputs into outputs, achieving the desired input-output relationship.
- A **correct algorithm** halts with the correct output for every input instance. We then say that the algorithm **solves** the problem.
Example of a Problem and an Instance

Computational Problem: Sorting

- **Input:** Sequence of $n$ numbers $\langle a_1, \cdots, a_n \rangle$.
- **Output:** Permutation (reordering)

$$\langle a'_1, a'_2, \cdots, a'_n \rangle$$

such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$. 
Example of a Problem and an Instance

Computational Problem: Sorting
- **Input**: Sequence of \( n \) numbers \( \langle a_1, \cdots, a_n \rangle \).
- **Output**: Permutation (reordering)

\[ \langle a'_1, a'_2, \cdots, a'_n \rangle \]

such that \( a'_1 \leq a'_2 \leq \cdots \leq a'_n \).

Instance of Problem Sorting
- **Input**: Permutation

\[ \langle 8, 3, 6, 7, 1, 2, 9 \rangle \]

- **Output**: Permutation (reordering)

\[ \langle 1, 2, 3, 6, 7, 8, 9 \rangle \]
Example of Algorithm: Insertion Sort

In-Place Sort: uses only a fixed amount of storage beyond that needed for the data.
Example of Algorithm: Insertion Sort

In-Place Sort: uses only a fixed amount of storage beyond that needed for the data.

Pseudocode:

A is an array of numbers

for $j = 2$ to length($A$)
{
    key = $A[j]$;
    $i = j - 1$;
    while ($i \geq 1$ and $A[i] >$ key)
    {
        $i = i - 1$;
    }
    $A[i + 1] = $ key;
}
Example of Algorithm: Insertion Sort

In-Place Sort: uses only a fixed amount of storage beyond that needed for the data.

Pseudocode:
$A$ is an array of numbers

for $j = 2$ to length($A$)
{
  key = $A[j]$;
  $i = j - 1$;
  while ($i \geq 1$ and $A[i] > key$)
  {
    $i = i - 1$;
  }
  $A[i + 1] = key$;
}

Pause:
How does it work?
Insertion Sort: an Incremental Approach

To sort an array of length $n$: $n$ steps

$i$th step sorts the array of the first $i$ items by inserting $i$th item properly into sorted array of the first $i - 1$ items (created in previous step)
Insertion Sort: an Incremental Approach

To sort an array of length $n$: $n$ steps

$i$th step sorts the array of the first $i$ items by inserting $i$th item properly into sorted array of the first $i - 1$ items (created in previous step)

Example: Sort $A = \langle 6, 3, 2, 4 \rangle$ with Insertion Sort.
Insertion Sort: an Incremental Approach

To sort an array of length $n$: $n$ steps

$i$th step sorts the array of the first $i$ items by inserting $i$th item properly into sorted array of the first $i - 1$ items (created in previous step)

**Example:** Sort $A = \langle 6, 3, 2, 4 \rangle$ with Insertion Sort.

Step 1: $\langle 6, 3, 2, 4 \rangle$
**Insertion Sort**: an Incremental Approach

To sort an array of length $n$: $n$ steps

$i$th step sorts the array of the first $i$ items by inserting $i$th item properly into sorted array of the first $i - 1$ items (created in previous step)

**Example**: Sort $A = \langle 6, 3, 2, 4 \rangle$ with Insertion Sort.

Step 1: $\langle 6, 3, 2, 4 \rangle$

Step 2: $\langle 3, 6, 2, 4 \rangle$
Insertion Sort: an Incremental Approach

To sort an array of length $n$: $n$ steps

$i$th step sorts the array of the first $i$ items by inserting $i$th item properly into sorted array of the first $i - 1$ items (created in previous step)

**Example:** Sort $A = \langle 6, 3, 2, 4 \rangle$ with Insertion Sort.

Step 1: $\langle 6, 3, 2, 4 \rangle$  
Step 2: $\langle 3, 6, 2, 4 \rangle$  
Step 3: $\langle 2, 3, 6, 4 \rangle$
Insertion Sort: an Incremental Approach

To sort an array of length \( n \): \( n \) steps

\( i \)th step sorts the array of the first \( i \) items by inserting \( i \)th item properly into sorted array of the first \( i - 1 \) items (created in previous step)

**Example:** Sort \( A = \langle 6, 3, 2, 4 \rangle \) with Insertion Sort.

Step 1: \( \langle 6, 3, 2, 4 \rangle \)  
Step 2: \( \langle 3, 6, 2, 4 \rangle \)  
Step 3: \( \langle 2, 3, 6, 4 \rangle \)  
Step 4: \( \langle 2, 3, 4, 6 \rangle \)
Analyzing Algorithms

Predict resource utilization

1. **Memory** (space complexity)
2. **Running time** (time complexity)
Analyzing Algorithms

Predict resource utilization

1. **Memory** (space complexity)
2. **Running time** (time complexity)

**Remark:** Depends on model of computation, e.g., *sequential vs. parallel* or *internal memory vs. external memory.*

In this class we usually assume *sequential* and *internal memory.*
Analyzing Algorithms (Continued)

**Running time:** the number of *primitive operations* used to solve the problem.
Analyzing Algorithms (Continued)

**Running time:** the number of primitive operations used to solve the problem.

**Primitive operations:**
e.g., addition, multiplication, comparisons. In more advanced models could be page faults or Map/Reduce calls.
Analyzing Algorithms (Continued)

**Running time:** the number of *primitive operations* used to solve the problem.

**Primitive operations:**
e.g., addition, multiplication, comparisons. In more advanced models could be page faults or Map/Reduce calls

**Running time:** depends on problem instance, often we find an upper bound: $F(\text{input size})$
Analyzing Algorithms (Continued)

**Running time:** the number of **primitive operations** used to solve the problem.

**Primitive operations:**
e.g., addition, multiplication, comparisons. In more advanced models could be page faults or Map/Reduce calls

**Running time:** depends on problem instance, often we find an upper bound: $F(\text{input size})$

**Input size:** rigorous definition given later.
1. **Sorting:** number of items to be sorted
2. **Multiplication:** number of bits, number of digits.
3. **Graphs:** number of vertices and edges.
Three Types of Algorithmic Analyses

**Best Case:** constraints on the input, other than size, resulting in the fastest possible running time.
Three Types of Algorithmic Analyses

**Best Case:** constraints on the input, other than size, resulting in the fastest possible running time.

**Worst Case:** constraints on the input, other than size, resulting in the slowest possible running time.

Example. In the worst case *Quicksort* runs in $\Theta(n^2)$ time on an input of $n$ keys.
Three Types of Algorithmic Analyses

**Best Case:** constraints on the input, other than size, resulting in the fastest possible running time.

**Worst Case:** constraints on the input, other than size, resulting in the slowest possible running time.

Example. In the worst case Quicksort runs in $\Theta(n^2)$ time on an input of $n$ keys.

**Average Case:** average running time over every possible type of input (usually involve probabilities of different types of input).

Example. In average case Quicksort runs in $\Theta(n \log n)$ time on an input of $n$ keys. All $n!$ inputs of $n$ keys are considered equally likely.
Three Types of Algorithmic Analyses

**Best Case**: constraints on the input, other than size, resulting in the fastest possible running time.

**Worst Case**: constraints on the input, other than size, resulting in the slowest possible running time. Example. In the worst case *Quicksort* runs in $\Theta(n^2)$ time on an input of $n$ keys.

**Average Case**: average running time over every possible type of input (usually involve probabilities of different types of input). Example. In average case *Quicksort* runs in $\Theta(n \log n)$ time on an input of $n$ keys. All $n!$ inputs of $n$ keys are considered equally likely.

**Remark**: All cases are relative to the algorithm under consideration.
Three Analyses of Insertion Sorting
Three Analyses of Insertion Sorting


The number of comparisons needed is equal to

$$1 + 1 + 1 + \cdots + 1 = n - 1 = \Theta(n).$$
Three Analyses of Insertion Sorting


The number of comparisons needed is equal to

\[
1 + 1 + 1 + \cdots + 1 = n - 1 = \Theta(n).
\]


The number of comparisons needed is equal to

\[
1 + 2 + \cdots + (n - 1) = \frac{n(n - 1)}{2} = \Theta(n^2).
\]
Three Analyses of Insertion Sorting


The number of comparisons needed is equal to

\[
1 + 1 + 1 + \cdots + 1 = n - 1 = \Theta(n).
\]


The number of comparisons needed is equal to

\[
1 + 2 + \cdots + (n - 1) = \frac{n(n - 1)}{2} = \Theta(n^2).
\]

**Average Case:**  \( \Theta(n^2) \) assuming that each of the \( n! \) instances are equally likely.
Further thoughts on algorithm design

- *Algorithm Design*, as taught in this class, is mainly about designing algorithms that have small big $O()$ running times.

- "All other things being equal", $O(n \log n)$ algorithms will run more quickly than $O(n^2)$ ones and $O(n)$ algorithms will beat $O(n \log n)$ ones.

- Being able to do good algorithm design lets you identify the hard parts of your problem and deal with them effectively.

- Too often, programmers try to solve problems using brute force techniques and end up with slow complicated code! A few hours of abstract thought devoted to algorithm design could have speeded up the solution substantially and simplified it.
Final Note

Note: After algorithm design one can continue on to Algorithm tuning which would further concentrate on improving algorithms by cutting cut down on the constants in the big $O()$ bounds. This needs a good understanding of both algorithm design principles and efficient use of data structures.

In this course we will not go further into algorithm tuning. For a good introduction, see Chapter 9 in Programming Pearls, 2nd ed by Jon Bentley.