Examples

**Shortest Path**

**Input:** directed graph \( G = (V, E) \), \( s, t \in V \)

**Output:** a shortest path from \( s \) to \( t \) in \( G \)
**Shortest Path**

**Input:** directed graph $G = (V, E)$, $s, t \in V$

**Output:** a shortest path from $s$ to $t$ in $G$
Examples

Shortest Path

**Input:** directed graph $G = (V, E)$, $s, t \in V$

**Output:** a shortest path from $s$ to $t$ in $G$
Examples

Shortest Path

**Input:** directed graph $G = (V, E)$, $s, t \in V$

**Output:** a shortest path from $s$ to $t$ in $G$

Algorithm: Dijkstra’s algorithm . . .
Examples

Shortest Path

**Input:** directed graph $G = (V, E)$ (may have negative edges), $s, t \in V$

**Output:** a shortest path from $s$ to $t$ in $G$
Examples

Shortest Path

**Input:** directed graph $G = (V, E)$ (may have negative edges), $s, t \in V$

**Output:** a shortest path from $s$ to $t$ in $G$
Examples

### Shortest Path

**Input:** directed graph $G = (V, E)$ (may have negative edges), $s, t \in V$

**Output:** a shortest path from $s$ to $t$ in $G$
Examples

**Shortest Path**

**Input:** directed graph $G = (V, E)$ (may have negative edges), $s, t \in V$

**Output:** a shortest path from $s$ to $t$ in $G$

Algorithm = Computer Program?

- Algorithm: “abstract”, can be specified using computer program, English, pseudo-codes or flow charts.

- Computer program: “concrete”, implementation of algorithm, using a particular programming language
Pseudo-Code

Euclidean(a, b)

1: while b > 0 do
2: (a, b) ← (b, a mod b)
3: return a

Python program:

def euclidean(a: int, b: int):
    c = 0
    while b > 0:
        c = b
        b = a % b
        a = c
    return a
Main focus: correctness, running time (efficiency)
Main focus: correctness, running time (efficiency)
Sometimes: memory usage

Why is it important to study the running time (efficiency) of an algorithm?

1. feasible vs. infeasible
2. efficient algorithms: less engineering tricks needed, can use languages aiming for easy programming (e.g., python)
3. fundamental
4. it is fun!
Theoretical Analysis of Algorithms

- Main focus: correctness, running time (efficiency)
- Sometimes: memory usage
- Not covered in the course: engineering side
  - extensibility
  - modularity
  - object-oriented model
  - user-friendliness (e.g., GUI)
  - ...

Why is it important to study the running time (efficiency) of an algorithm?

1. feasible vs. infeasible
2. efficient algorithms: less engineering tricks needed, can use languages aiming for easy programming (e.g., python)
3. fundamental
4. it is fun!
Main focus: correctness, running time (efficiency)
Sometimes: memory usage
Not covered in the course: engineering side
- extensibility
- modularity
- object-oriented model
- user-friendliness (e.g, GUI)
- ...

Why is it important to study the running time (efficiency) of an algorithm?
Theoretical Analysis of Algorithms

- Main focus: correctness, running time (efficiency)
- Sometimes: memory usage
- Not covered in the course: engineering side
  - extensibility
  - modularity
  - object-oriented model
  - user-friendliness (e.g., GUI)
  - ...

- Why is it important to study the running time (efficiency) of an algorithm?
  - feasible vs. infeasible
Theoretical Analysis of Algorithms

- Main focus: correctness, running time (efficiency)
- Sometimes: memory usage
- Not covered in the course: engineering side
  - extensibility
  - modularity
  - object-oriented model
  - user-friendliness (e.g, GUI)
  - ...

Why is it important to study the running time (efficiency) of an algorithm?

1. feasible vs. infeasible
2. efficient algorithms: less engineering tricks needed, can use languages aiming for easy programming (e.g, python)
Theoretical Analysis of Algorithms

- Main focus: correctness, running time (efficiency)
- Sometimes: memory usage
- Not covered in the course: engineering side
  - extensibility
  - modularity
  - object-oriented model
  - user-friendliness (e.g, GUI)
  - ...

Why is it important to study the running time (efficiency) of an algorithm?

1. feasible vs. infeasible
2. efficient algorithms: less engineering tricks needed, can use languages aiming for easy programming (e.g, python)
3. fundamental
Theoretical Analysis of Algorithms

- Main focus: correctness, running time (efficiency)
- Sometimes: memory usage
- Not covered in the course: engineering side
  - extensibility
  - modularity
  - object-oriented model
  - user-friendliness (e.g., GUI)
  - ...

Why is it important to study the running time (efficiency) of an algorithm?
1. feasible vs. infeasible
2. efficient algorithms: less engineering tricks needed, can use languages aiming for easy programming (e.g., python)
3. fundamental
4. it is fun!
Outline

1. Syllabus

2. Introduction
   - What is an Algorithm?
   - Example: Insertion Sort
   - Analysis of Insertion Sort

3. Asymptotic Notations

4. Common Running times
Sorting Problem

**Input:** sequence of $n$ numbers $(a_1, a_2, \cdots, a_n)$

**Output:** a permutation $(a'_1, a'_2, \cdots, a'_n)$ of the input sequence such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$

Example:

- **Input:** 53, 12, 35, 21, 59, 15
- **Output:** 12, 15, 21, 35, 53, 59
At the end of $j$-th iteration, the first $j$ numbers are sorted.

iteration 1: 53, 12, 35, 21, 59, 15
iteration 2: 12, 53, 35, 21, 59, 15
iteration 3: 12, 35, 53, 21, 59, 15
iteration 4: 12, 21, 35, 53, 59, 15
iteration 5: 12, 21, 35, 53, 59, 15
iteration 6: 12, 15, 21, 35, 53, 59
Example:

- **Input:** 53, 12, 35, 21, 59, 15
- **Output:** 12, 15, 21, 35, 53, 59

**insertion-sort** \( (A, n) \)

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. \( A[i + 1] \leftarrow key \)
Example:
- Input: 53, 12, 35, 21, 59, 15
- Output: 12, 15, 21, 35, 53, 59

`insertion-sort(A, n)`

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

- $j = 6$
- `key' = 15

12  21  35  53  59  15

\[\uparrow\]
\[i\]
Example:
- Input: 53, 12, 35, 21, 59, 15
- Output: 12, 15, 21, 35, 53, 59

insertion-sort($A, n$)

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:   $A[i + 1] \leftarrow key$

- $j = 6$
- $key = 15$

12 21 35 53 59 59

↑

$i$
Example:

- Input: 53, 12, 35, 21, 59, 15
- Output: 12, 15, 21, 35, 53, 59

**insertion-sort**(A, n)

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} $A[i + 1] \leftarrow key$

- $j = 6$
- key = 15

12  21  35  53  59  59

$\uparrow$

$i$
Example:
- Input: 53, 12, 35, 21, 59, 15
- Output: 12, 15, 21, 35, 53, 59

insertion-sort($A, n$)

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} $A[i + 1] \leftarrow key$

- $j = 6$
- key = 15

12 21 35 53 53 59

$\uparrow$

i
Example:
- Input: 53, 12, 35, 21, 59, 15
- Output: 12, 15, 21, 35, 53, 59

**insertion-sort**($A, n$)

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:   $A[i + 1] \leftarrow key$

- $j = 6$
- $key = 15$

12  21  35  53  53  59  

$\uparrow$

$i$
Example:
- **Input:** 53, 12, 35, 21, 59, 15
- **Output:** 12, 15, 21, 35, 53, 59

**insertion-sort**\((A, n)\)

\[
\begin{align*}
1: & \textbf{for } j \leftarrow 2 \textbf{ to } n \textbf{ do} \\
2: & \quad \text{key } \leftarrow A[j] \\
3: & \quad i \leftarrow j - 1 \\
4: & \quad \textbf{while } i > 0 \text{ and } A[i] > \text{key} \textbf{ do} \\
5: & \quad \quad A[i + 1] \leftarrow A[i] \\
6: & \quad \quad i \leftarrow i - 1 \\
7: & \quad A[i + 1] \leftarrow \text{key}
\end{align*}
\]

- j = 6
- key = 15

\[
\begin{array}{cccccccc}
12 & 21 & 35 & 35 & 53 & 59 \\
\uparrow & i \\
\end{array}
\]
Example:

- **Input**: 53, 12, 35, 21, 59, 15
- **Output**: 12, 15, 21, 35, 53, 59

**insertion-sort(A, n)**

1. **for** $j \leftarrow 2$ to $n$ **do**
2. \hspace{1em} $key \leftarrow A[j]$
3. \hspace{1em} $i \leftarrow j - 1$
4. **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} $A[i + 1] \leftarrow key$

- $j = 6$
- $key = 15$

12 21 35 35 53 59
Example:

- Input: 53, 12, 35, 21, 59, 15
- Output: 12, 15, 21, 35, 53, 59

**insertion-sort** \((A, n)\)

1: \textbf{for} \(j \leftarrow 2\) to \(n\) \textbf{do}
2: \hspace{1em} \text{key} \leftarrow A[j]
3: \hspace{1em} i \leftarrow j - 1
4: \hspace{1em} \textbf{while} \(i > 0\) and \(A[i] > \text{key}\) \textbf{do}
5: \hspace{2em} A[i + 1] \leftarrow A[i]
6: \hspace{2em} i \leftarrow i - 1
7: \hspace{1em} A[i + 1] \leftarrow \text{key}

- \(j = 6\)
- \(\text{key} = 15\)

12 \hspace{0.5em} 21 \hspace{0.5em} 21 \hspace{0.5em} 35 \hspace{0.5em} 53 \hspace{0.5em} 59

\[ i \]

\[ \uparrow \]
Example:

- Input: 53, 12, 35, 21, 59, 15
- Output: 12, 15, 21, 35, 53, 59

**insertion-sort**($A, n$)

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

- $j = 6$
- $key = 15$

12 21 21 35 53 59

$\uparrow$

$i$
Example:
- Input: 53, 12, 35, 21, 59, 15
- Output: 12, 15, 21, 35, 53, 59

**insertion-sort**($A, n$)

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.   $A[i + 1] \leftarrow key$

- $j = 6$
- $key = 15$
- 12 15 21 35 53 59
  ▲
  ▲
  ▲
  $i$
Outline

1 Syllabus

2 Introduction
   - What is an Algorithm?
   - Example: Insertion Sort
   - Analysis of Insertion Sort

3 Asymptotic Notations

4 Common Running times
Analysis of Insertion Sort

- Correctness
- Running time
Invariant: after iteration $j$ of outer loop, $A[1..j]$ is the sorted array for the original $A[1..j]$.

- after $j = 1$: $53, 12, 35, 21, 59, 15$
- after $j = 2$: $12, 53, 35, 21, 59, 15$
- after $j = 3$: $12, 35, 53, 21, 59, 15$
- after $j = 4$: $12, 21, 35, 53, 59, 15$
- after $j = 5$: $12, 21, 35, 53, 59, 15$
- after $j = 6$: $12, 15, 21, 35, 53, 59$
Analyzing Running Time of Insertion Sort

Q1: what is the size of input?
Q1: what is the size of input?
A1: Running time as the function of size
Q1: what is the size of input?
A1: Running time as the function of size
possible definition of size:
  - Sorting problem: \# integers,
  - Greatest common divisor: total length of two integers
  - Shortest path in a graph: \# edges in graph
Analyzing Running Time of Insertion Sort

Q1: what is the size of input?
A1: Running time as the function of size
possible definition of size:
- Sorting problem: number of integers,
- Greatest common divisor: total length of two integers
- Shortest path in a graph: number of edges in graph

Q2: Which input?
For the insertion sort algorithm: if input array is already sorted in ascending order, then algorithm runs much faster than when it is sorted in descending order.
Analyzing Running Time of Insertion Sort

- Q1: what is the size of input?
- A1: Running time as the function of size
- possible definition of size:
  - Sorting problem: \# integers,
  - Greatest common divisor: total length of two integers
  - Shortest path in a graph: \# edges in graph

- Q2: Which input?
  - For the insertion sort algorithm: if input array is already sorted in ascending order, then algorithm runs much faster than when it is sorted in descending order.
- A2: Worst-case analysis:
  - Running time for size \( n \) = worst running time over all possible arrays of length \( n \)
Q3: How fast is the computer?
Q4: Programming language?
Q3: How fast is the computer?
Q4: Programming language?
A: They do not matter!
Q3: How fast is the computer?
Q4: Programming language?
A: They do not matter!

Important idea: asymptotic analysis
- Focus on growth of running-time as a function, not any particular value.
Asymptotic Analysis: $O$-notation

Informal way to define $O$-notation:

- Ignoring lower order terms
- Ignoring leading constant
Informal way to define $O$-notation:

- Ignoring lower order terms
- Ignoring leading constant

$3n^3 + 2n^2 - 18n + 1028 \Rightarrow 3n^3 \Rightarrow n^3$
Asymptotic Analysis: $O$-notation

Informal way to define $O$-notation:

- Ignoring lower order terms
- Ignoring leading constant

- $3n^3 + 2n^2 - 18n + 1028 \Rightarrow 3n^3 \Rightarrow n^3$
- $3n^3 + 2n^2 - 18n + 1028 = O(n^3)$
Asymptotic Analysis: $O$-notation

Informal way to define $O$-notation:
- Ignoring lower order terms
- Ignoring leading constant

- $3n^3 + 2n^2 - 18n + 1028 \Rightarrow 3n^3 \Rightarrow n^3$
- $3n^3 + 2n^2 - 18n + 1028 = O(n^3)$
- $n^2/100 - 3n + 10 \Rightarrow n^2/100 \Rightarrow n^2$
Asymptotic Analysis: \( O \)-notation

Informal way to define \( O \)-notation:
- Ignoring lower order terms
- Ignoring leading constant

\[
\begin{align*}
3n^3 + 2n^2 - 18n + 1028 &\Rightarrow 3n^3 \Rightarrow n^3 \\
3n^3 + 2n^2 - 18n + 1028 &= O(n^3) \\
n^2/100 - 3n + 10 &\Rightarrow n^2/100 \Rightarrow n^2 \\
n^2/100 - 3n + 10 &= O(n^2)
\end{align*}
\]
Asymptotic Analysis: $O$-notation

- $3n^3 + 2n^2 - 18n + 1028 = O(n^3)$
- $n^2/100 - 3n + 10 = O(n^2)$
Asymptotic Analysis: $O$-notation

- $3n^3 + 2n^2 - 18n + 1028 = O(n^3)$
- $n^2/100 - 3n + 10 = O(n^2)$

$O$-notation allows us to ignore
- architecture of computer
- programming language
- how we measure the running time: seconds or # instructions?
Asymptotic Analysis: $O$-notation

- $3n^3 + 2n^2 - 18n + 1028 = O(n^3)$
- $n^2/100 - 3n + 10 = O(n^2)$

$O$-notation allows us to ignore
- architecture of computer
- programming language
- how we measure the running time: seconds or \# instructions?

To execute $a \leftarrow b + c$:
- program 1 requires 10 instructions, or $10^{-8}$ seconds
- program 2 requires 2 instructions, or $10^{-9}$ seconds
Asymptotic Analysis: $O$-notation

- $3n^3 + 2n^2 - 18n + 1028 = O(n^3)$
- $n^2/100 - 3n + 10 = O(n^2)$

$O$-notation allows us to ignore:
- architecture of computer
- programming language
- how we measure the running time: seconds or # instructions?

To execute $a \leftarrow b + c$:
- program 1 requires 10 instructions, or $10^{-8}$ seconds
- program 2 requires 2 instructions, or $10^{-9}$ seconds
- they only change by a constant in the running time, which will be hidden by the $O(\cdot)$ notation
Asymptotic Analysis: $O$-notation

- Algorithm 1 runs in time $O(n^2)$
- Algorithm 2 runs in time $O(n)$
Algorithm 1 runs in time $O(n^2)$
Algorithm 2 runs in time $O(n)$

Does not tell which algorithm is faster for a specific $n$!
Asymptotic Analysis: $O$-notation

- Algorithm 1 runs in time $O(n^2)$
- Algorithm 2 runs in time $O(n)$
- Does not tell which algorithm is faster for a specific $n$!
- Algorithm 2 will eventually beat algorithm 1 as $n$ increases.
Asymptotic Analysis: \( O \)-notation

- Algorithm 1 runs in time \( O(n^2) \)
- Algorithm 2 runs in time \( O(n) \)

- Does not tell which algorithm is faster for a specific \( n \)!
- Algorithm 2 will eventually beat algorithm 1 as \( n \) increases.

- For Algorithm 1: if we increase \( n \) by a factor of 2, running time increases by a factor of 4
Algorithm 1 runs in time $O(n^2)$
Algorithm 2 runs in time $O(n)$

Does not tell which algorithm is faster for a specific $n$!
Algorithm 2 will eventually beat algorithm 1 as $n$ increases.

For Algorithm 1: if we increase $n$ by a factor of 2, running time increases by a factor of 4

For Algorithm 2: if we increase $n$ by a factor of 2, running time increases by a factor of 2
Asymptotic Analysis of Insertion Sort

insertion-sort(\(A, n\))

1: \textbf{for} \(j \leftarrow 2\) \textbf{to} \(n\) \textbf{do}
2: \hspace{1em} \textit{key} \leftarrow A[j]
3: \hspace{1em} \(i \leftarrow j - 1\)
4: \hspace{1em} \textbf{while} \(i > 0\) \textbf{and} \(A[i] > \textit{key}\) \textbf{do}
5: \hspace{2em} \(A[i + 1] \leftarrow A[i]\)
6: \hspace{2em} \(i \leftarrow i - 1\)
7: \hspace{1em} \(A[i + 1] \leftarrow \textit{key}\)

Worst-case running time for iteration \(j\) of the outer loop?

Answer: \(O(j)\)

Total running time = \(\sum_{j=2}^{n} O(j) = O(n(n+1)/2) = O(n^2)\)
## Asymptotic Analysis of Insertion Sort

### insertion-sort($A, n$)

1. **for** $j \leftarrow 2$ **to** $n$ **do**
2. \hspace{1em} $key \leftarrow A[j]$
3. \hspace{1em} $i \leftarrow j - 1$
4. **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. **end while**
8. **end for**

- Worst-case running time for iteration $j$ of the outer loop?

$$W = \text{Worst-case running time for iteration } j$$

**Answer:**

$$W = O(j)$$

**Total running time:**

$$\sum_{j=2}^{n} W_j = \sum_{j=2}^{n} O(j) = O(\sum_{j=2}^{n} j) = O\left(\frac{n^2}{2}\right) = O\left(n^2\right)$$
Asymptotic Analysis of Insertion Sort

**insertion-sort**\((A, n)\)

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. \(A[i + 1] \leftarrow key\)

- Worst-case running time for iteration \(j\) of the outer loop?
  Answer: \(O(j)\)
Asymptotic Analysis of Insertion Sort

**insertion-sort**(*A, n*)

1. **for** *j* ← 2 to *n* **do**
2.   `key ← A[j]`
3.   `i ← j - 1`
4.   **while** *i* > 0 and *A[i] > key** do
6.     `i ← i - 1`
7.   `A[i + 1] ← key`

- **Worst-case running time for iteration** *j* of the outer loop?
  - Answer: *O*(*j*)

- **Total running time** = \[ \sum_{j=2}^{n} O(j) = O(\sum_{j=2}^{n} j) \]
  \[ = O\left(\frac{n(n+1)}{2} - 1\right) = O(n^2) \]
Computation Model

Random-Access Machine (RAM) model

- Reading and writing: $\mathcal{O}(1)$ time
- Basic operations such as addition, subtraction and multiplication: $\mathcal{O}(1)$ time
- Each integer (word) has $c \log n$ bits, $c \geq 1$ is large enough.
  - Reason: often we need to read the integer $n$ and handle integers within range $[n^c, n^c]$, it is convenient to assume this takes $\mathcal{O}(1)$ time.

- What is the precision of real numbers?
  - Most of the time, we only consider integers.

- Can we do better than insertion sort asymptotically?
  - Yes: merge sort, quicksort and heap sort take $\mathcal{O}(n \log n)$ time.
Computation Model

- Random-Access Machine (RAM) model
- reading and writing $A[j]$ takes $O(1)$ time

Basic operations such as addition, subtraction and multiplication take $O(1)$ time. Each integer (word) has $c \log n$ bits, with $c$ large enough.

Reason: often we need to read the integer $n$ and handle integers within range $[n^c, n^c]$, it is convenient to assume this takes $O(1)$ time.

What is the precision of real numbers? Most of the time, we only consider integers.

Can we do better than insertion sort asymptotically? Yes: merge sort, quicksort and heap sort take $O(n \log n)$ time.
Computation Model

- Random-Access Machine (RAM) model
  - reading and writing $A[j]$ takes $O(1)$ time
- Basic operations such as addition, subtraction and multiplication take $O(1)$ time
Computation Model

- Random-Access Machine (RAM) model
  - reading and writing $A[j]$ takes $O(1)$ time
- Basic operations such as addition, subtraction and multiplication take $O(1)$ time
- Each integer (word) has $c \log n$ bits, $c \geq 1$ large enough
- Reason: often we need to read the integer $n$ and handle integers within range $[-n^c, n^c]$, it is convenient to assume this takes $O(1)$ time.
Random-Access Machine (RAM) model
- reading and writing $A[j]$ takes $O(1)$ time

Basic operations such as addition, subtraction and multiplication take $O(1)$ time

Each integer (word) has $c \log n$ bits, $c \geq 1$ large enough
- Reason: often we need to read the integer $n$ and handle integers within range $[-n^c, n^c]$, it is convenient to assume this takes $O(1)$ time.

What is the precision of real numbers?
Computation Model

- Random-Access Machine (RAM) model
- reading and writing $A[j]$ takes $O(1)$ time
- Basic operations such as addition, subtraction and multiplication take $O(1)$ time
- Each integer (word) has $c \log n$ bits, $c \geq 1$ large enough
- Reason: often we need to read the integer $n$ and handle integers within range $[-n^c, n^c]$, it is convenient to assume this takes $O(1)$ time.
- What is the precision of real numbers?
  Most of the time, we only consider integers.
Computation Model

- Random-Access Machine (RAM) model
  - reading and writing $A[j]$ takes $O(1)$ time
- Basic operations such as addition, subtraction and multiplication take $O(1)$ time
- Each integer (word) has $c \log n$ bits, $c \geq 1$ large enough
  - Reason: often we need to read the integer $n$ and handle integers within range $[-n^c, n^c]$, it is convenient to assume this takes $O(1)$ time.
- What is the precision of real numbers?
  - Most of the time, we only consider integers.
- Can we do better than insertion sort asymptotically?
Computation Model

- Random-Access Machine (RAM) model
  - reading and writing $A[j]$ takes $O(1)$ time
- Basic operations such as addition, subtraction and multiplication take $O(1)$ time
- Each integer (word) has $c \log n$ bits, $c \geq 1$ large enough
  - Reason: often we need to read the integer $n$ and handle integers within range $[-n^c, n^c]$, it is convenient to assume this takes $O(1)$ time.
- What is the precision of real numbers?
  - Most of the time, we only consider integers.
- Can we do better than insertion sort asymptotically?
  - Yes: merge sort, quicksort and heap sort take $O(n \log n)$ time
Outline

1 Syllabus

2 Introduction
   - What is an Algorithm?
   - Example: Insertion Sort
   - Analysis of Insertion Sort

3 Asymptotic Notations

4 Common Running times
Asymptotically Positive Functions

**Def.** \( f : \mathbb{N} \rightarrow \mathbb{R} \) is an **asymptotically positive function** if:

\[ \exists n_0 > 0 \text{ such that } \forall n > n_0 \text{ we have } f(n) > 0 \]
Asymptotically Positive Functions

**Def.** $f : \mathbb{N} \rightarrow \mathbb{R}$ is an asymptotically positive function if:

- $\exists n_0 > 0$ such that $\forall n > n_0$ we have $f(n) > 0$

- In other words, $f(n)$ is positive for large enough $n$. 
Asymptotically Positive Functions

Def. $f : \mathbb{N} \rightarrow \mathbb{R}$ is an asymptotically positive function if:

- \( \exists n_0 > 0 \) such that \( \forall n > n_0 \) we have \( f(n) > 0 \)

- In other words, \( f(n) \) is positive for large enough \( n \).

- \( n^2 - n - 30 \)
Asymptotically Positive Functions

Def. $f : \mathbb{N} \rightarrow \mathbb{R}$ is an asymptotically positive function if:

- $\exists n_0 > 0$ such that $\forall n > n_0$ we have $f(n) > 0$

- In other words, $f(n)$ is positive for large enough $n$.
- $n^2 - n - 30$ Yes
Asymptotically Positive Functions

**Def.** \( f : \mathbb{N} \to \mathbb{R} \) is an **asymptotically positive function** if:

- \( \exists n_0 > 0 \) such that \( \forall n > n_0 \) we have \( f(n) > 0 \)

In other words, \( f(n) \) is positive for large enough \( n \).

- \( n^2 - n - 30 \) Yes
- \( 2^n - n^{20} \)
Asymptotically Positive Functions

**Def.** $f : \mathbb{N} \rightarrow \mathbb{R}$ is an **asymptotically positive function** if:

- $\exists n_0 > 0$ such that $\forall n > n_0$ we have $f(n) > 0$

- In other words, $f(n)$ is positive for large enough $n$.

- $n^2 - n - 30$  Yes

- $2^n - n^{20}$  Yes
Def. $f : \mathbb{N} \to \mathbb{R}$ is an asymptotically positive function if:

- $\exists n_0 > 0$ such that $\forall n > n_0$ we have $f(n) > 0$

In other words, $f(n)$ is positive for large enough $n$.

- $n^2 - n - 30$ \hspace{1cm} Yes
- $2^n - n^{20}$ \hspace{1cm} Yes
- $100n - n^2 / 10 + 50$?
Asymptotically Positive Functions

**Def.** \( f : \mathbb{N} \rightarrow \mathbb{R} \) is an *asymptotically positive function* if:

\[ \exists n_0 > 0 \text{ such that } \forall n > n_0 \text{ we have } f(n) > 0 \]

- In other words, \( f(n) \) is positive for large enough \( n \).
- \( n^2 - n - 30 \) \hspace{1cm} Yes
- \( 2^n - n^{20} \) \hspace{1cm} Yes
- \( 100n - n^2 / 10 + 50 \) \hspace{1cm} No
Def. $f : \mathbb{N} \rightarrow \mathbb{R}$ is an asymptotically positive function if:

- $\exists n_0 > 0$ such that $\forall n > n_0$ we have $f(n) > 0$

In other words, $f(n)$ is positive for large enough $n$.

- $n^2 - n - 30$ Yes
- $2^n - n^{20}$ Yes
- $100n - n^2/10 + 50?$ No

We only consider asymptotically positive functions.
**O-Notation: Asymptotic Upper Bound**

**O-Notation**  For a function $g(n)$,

$$O(g(n)) = \{ \text{function } f : \exists c > 0, n_0 > 0 \text{ such that } f(n) \leq cg(n), \forall n \geq n_0 \}.$$
**O-Notation: Asymptotic Upper Bound**

**O-Notation** For a function $g(n)$,

$$O(g(n)) = \{ \text{function } f : \exists c > 0, n_0 > 0 \text{ such that } f(n) \leq cg(n), \forall n \geq n_0 \}.$$

- In other words, $f(n) \in O(g(n))$ if $f(n) \leq cg(n)$ for some $c > 0$ and every large enough $n$. 
**O-Notation: Asymptotic Upper Bound**

**O-Notation** For a function \( g(n) \),

\[
O(g(n)) = \{ \text{function } f : \exists c > 0, n_0 > 0 \text{ such that } f(n) \leq cg(n), \forall n \geq n_0 \}.
\]

- In other words, \( f(n) \in O(g(n)) \) if \( f(n) \leq cg(n) \) for some \( c > 0 \) and every large enough \( n \).
**O-Notation: Asymptotic Upper Bound**

**O-Notation** For a function $g(n)$,

$$O(g(n)) = \{ \text{function } f : \exists c > 0, n_0 > 0 \text{ such that } f(n) \leq cg(n), \forall n \geq n_0 \}.$$ 

- In other words, $f(n) \in O(g(n))$ if $f(n) \leq cg(n)$ for some $c > 0$ and every large enough $n$.
- $3n^2 + 2n \in O(n^2 - 10n)$
**O-Notation: Asymptotic Upper Bound**

**O-Notation** For a function $g(n)$,

$$O(g(n)) = \{ \text{function } f : \exists c > 0, n_0 > 0 \text{ such that } f(n) \leq cg(n), \forall n \geq n_0 \}.$$  

- In other words, $f(n) \in O(g(n))$ if $f(n) \leq cg(n)$ for some $c > 0$ and every large enough $n$.
- $3n^2 + 2n \in O(n^2 - 10n)$

**Proof.**

Let $c = 4$ and $n_0 = 50$, for every $n > n_0 = 50$, we have,

$$3n^2 + 2n - c(n^2 - 10n) = 3n^2 + 2n - 4(n^2 - 10n)$$

$$= -n^2 + 42n \leq 0.$$  

$$3n^2 + 2n \leq c(n^2 - 10n)$$
**O-Notation**  For a function \( g(n) \),

\[
O(g(n)) = \{ \text{function } f : \exists c > 0, n_0 > 0 \text{ such that } f(n) \leq cg(n), \forall n \geq n_0 \}. 
\]

- In other words, \( f(n) \in O(g(n)) \) if \( f(n) \leq cg(n) \) for some \( c \) and large enough \( n \).
- \( 3n^2 + 2n \in O(n^2 - 10n) \)
**O-Notation**  For a function \(g(n)\),

\[
O(g(n)) = \{ \text{function } f : \exists c > 0, n_0 > 0 \text{ such that } f(n) \leq cg(n), \forall n \geq n_0 \}.
\]

- In other words, \(f(n) \in O(g(n))\) if \(f(n) \leq cg(n)\) for some \(c\) and large enough \(n\).
- \(3n^2 + 2n \in O(n^2 - 10n)\)
- \(3n^2 + 2n \in O(n^3 - 5n^2)\)
**O-Notation**  For a function $g(n)$,

$$O(g(n)) = \{ \text{function } f : \exists c > 0, n_0 > 0 \text{ such that } f(n) \leq cg(n), \forall n \geq n_0 \}.$$ 

- In other words, $f(n) \in O(g(n))$ if $f(n) \leq cg(n)$ for some $c$ and large enough $n$.
- $3n^2 + 2n \in O(n^2 - 10n)$
- $3n^2 + 2n \in O(n^3 - 5n^2)$
- $n^{100} \in O(2^n)$
**O-Notation**  For a function $g(n)$,

$$O(g(n)) = \{ \text{function } f : \exists c > 0, n_0 > 0 \text{ such that } f(n) \leq cg(n), \forall n \geq n_0 \}.$$

- In other words, $f(n) \in O(g(n))$ if $f(n) \leq cg(n)$ for some $c$ and large enough $n$.
- $3n^2 + 2n \in O(n^2 - 10n)$
- $3n^2 + 2n \in O(n^3 - 5n^2)$
- $? \ n^{100} \in O(2^n)$
- $? \ \sin n \in O(1/2)$
O-Notation  For a function $g(n)$, 

$$O(g(n)) = \{ \text{function } f : \exists c > 0, n_0 > 0 \text{ such that } f(n) \leq cg(n), \forall n \geq n_0 \}.$$ 

- In other words, $f(n) \in O(g(n))$ if $f(n) \leq cg(n)$ for some $c$ and large enough $n$.

- $3n^2 + 2n \in O(n^2 - 10n)$
- $3n^2 + 2n \in O(n^3 - 5n^2)$
- $? n^{100} \in O(2^n)$
- $? \sin n \in O(1/2)$
- $? n^3 \notin O(10n^2)$
**O-Notation**  For a function $g(n)$,

\[ O(g(n)) = \{ \text{function } f : \exists c > 0, n_0 > 0 \text{ such that } f(n) \leq cg(n), \forall n \geq n_0 \} . \]

- In other words, $f(n) \in O(g(n))$ if $f(n) \leq cg(n)$ for some $c$ and large enough $n$.
- $3n^2 + 2n \in O(n^2 - 10n)$
- $3n^2 + 2n \in O(n^3 - 5n^2)$
- $n^{100} \in O(2^n)$
- $\sin n \in O(1/2)$
- $n^3 \notin O(10n^2)$

<table>
<thead>
<tr>
<th>Asymptotic Notations</th>
<th>$O$</th>
<th>$\Omega$</th>
<th>$\Theta$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comparison Relations</td>
<td>$\leq$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
We use \( f(n) = O(g(n)) \) to denote \( f(n) \in O(g(n)) \)
Conventions

- We use \( f(n) = O(g(n)) \) to denote \( f(n) \in O(g(n)) \)
- \( 3n^2 + 2n = O(n^3 - 10n) \)
- \( 3n^2 + 2n = O(n^2 + 5n) \)
- \( 3n^2 + 2n = O(n^2) \)

Analogy: Mike is a student. 
A student is Mike.
Conventions

- We use \( f(n) = O(g(n)) \) to denote \( f(n) \in O(g(n)) \)
- \( 3n^2 + 2n = O(n^3 - 10n) \)
- \( 3n^2 + 2n = O(n^2 + 5n) \)
- \( 3n^2 + 2n = O(n^2) \)

“=” is asymmetric! Following equalities are wrong:
- \( O(n^3 - 10n) = 3n^2 + 2n \)
- \( O(n^2 + 5n) = 3n^2 + 2n \)
- \( O(n^2) = 3n^2 + 2n \)
Conventions

- We use \( f(n) = O(g(n)) \) to denote \( f(n) \in O(g(n)) \)
- \( 3n^2 + 2n = O(n^3 - 10n) \)
- \( 3n^2 + 2n = O(n^2 + 5n) \)
- \( 3n^2 + 2n = O(n^2) \)

“=” is asymmetric! Following equalities are wrong:
- \( O(n^3 - 10n) = 3n^2 + 2n \)
- \( O(n^2 + 5n) = 3n^2 + 2n \)
- \( O(n^2) = 3n^2 + 2n \)

Analogy: Mike is a student. A student is Mike.