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 $G$

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

**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 $G$
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: $(a, b) \leftarrow (b, a \mod b)$
3: **return** $a$

Python program:

```python
def euclidean(a: int, b: int):
    c = 0
    while b > 0:
        c = b
        b = a % b
        a = c
    return a
```
Theoretical Analysis of Algorithms

- Main focus: correctness, running time (efficiency)

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
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)
  - ...
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?
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
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: 
3: 
4: **while** $i > 0$ **and** $A[i] > key$ **do**
5: 
6: 
7: $A[i + 1] \leftarrow key$

- $j = 6$
- $key = 15$
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{2em} i \leftarrow i - 1
7: \hspace{1em} A[i + 1] \leftarrow key

- \( j = 6 \)
- \( key = 15 \)

12 21 35 53 59 59 59

\( i \)

\( i \)
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{1cm} $key \leftarrow A[j]$
3. \hspace{1cm} $i \leftarrow j - 1$
4. **while** $i > 0$ and $A[i] > key$ **do**
5. \hspace{2cm} $A[i + 1] \leftarrow A[i]$
6. \hspace{2cm} $i \leftarrow i - 1$
7. \hspace{2cm} $A[i + 1] \leftarrow key$

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

12 \hspace{0.5cm} 21 \hspace{0.5cm} 35 \hspace{0.5cm} 53 \hspace{0.5cm} 59 \hspace{0.5cm} 59$

\[ \uparrow \hspace{2cm} i \]
**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{2em} $i \leftarrow i - 1$
7. \hspace{1em} $A[i + 1] \leftarrow key$

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

12 21 35 53 53 59

$\uparrow$

\hspace{1em} $i$
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

↑

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

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

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

\(j = 6\)
\(key = 15\)

12 21  \underline{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. $A[i + 1] \leftarrow key$

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

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
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
Correctness of Insertion Sort


  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
Q1: what is the size of input?

- Sorting problem: # integers,
- Greatest common divisor: total length of two integers
- Shortest path in a graph: # edges in graph

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.

Worst-case analysis:
- Running time for size $n$ is the worst running time over all possible arrays of length $n$. 
Q1: what is the size of input?
A1: Running time as the function of size
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
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.
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 $n = \text{worst running time over all possible arrays of length } n$
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: \( \mathcal{O} \)-notation

Informal way to define \( \mathcal{O} \)-notation:

- Ignoring lower order terms
- Ignoring leading constant
Asymptotic Analysis: $O$-notation

Informal way to define $O$-notation:

- Ignoring lower order terms
- Ignoring leading constant

$3n^3 + 2n^2 - 18n + 1028 \Rightarrow 3n^3 \Rightarrow n^3$
Informal way to define $O$-notation:

- Ignoring lower order terms
- Ignoring leading constant

\[ 3n^3 + 2n^2 - 18n + 1028 \Rightarrow 3n^3 \Rightarrow n^3 \]

\[ 3n^3 + 2n^2 - 18n + 1028 = O(n^3) \]
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

- $3n^3 + 2n^2 - 18n + 1028 \Rightarrow 3n^3 \Rightarrow n^3$
- $3n^3 + 2n^2 - 18n + 1028 = O(n^3)$
- $n^2/100 - 3n + 10 \Rightarrow n^2/100 \Rightarrow n^2$
- $n^2/100 - 3n + 10 = O(n^2)$
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 $O(n)$
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 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.
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: 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} $A[i + 1] \leftarrow key$

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

Total running time = $\sum_{j=2}^{n} O(j) = O(n^2)$
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?
Asymptotic Analysis of Insertion Sort

```plaintext
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)
```
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**
5:        *A*[*i* + 1] ← *A*[*i*]
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 = $\sum_{j=2}^{n} O(j) = O(\sum_{j=2}^{n} j)$
  = $O(\frac{n(n+1)}{2} − 1) = O(n^2)$
Computation Model

Random-Access Machine (RAM) model

- Reading and writing a[i] takes $O(1)$ time.

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

Each integer (word) has $c \log n$ bits, where $c$ is large enough.

Reason: Often, we need to read the integer $n$ and handle integers within the 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.

Each integer (word) has $c \log n$ bits, $c$ 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

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
- 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.
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?
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
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 $f(n) > 0$
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 \( n \).
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 $n$.

- $n^2 - n - 30$
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 $n$.
- $n^2 - n - 30$ \quad Yes

We only consider asymptotically positive functions.
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 \( n \).
- \( n^2 - n - 30 \) \( \text{Yes} \)
- \( 2^n - n^{20} \)
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 $n$.
- $n^2 - n - 30 \quad \text{Yes}$
- $2^n - n^{20} \quad \text{Yes}$
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 \( n \).

- \( n^2 - n - 30 \) \( \quad \) Yes
- \( 2^n - n^{20} \) \( \quad \) Yes
- \( 100n - n^2 / 10 + 50 \) \( \quad \) ?
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
- \( 2^n - n^{20} \) Yes
- \( 100n - n^2/10 + 50? \) No
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
- $2^n - n^{20}$ 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 $n$. 

![Graph showing $f(n) = O(g(n))$ and $cg(n)$](image)
**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,

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

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

$$3n^2 + 2n \leq c(n^2 - 10n)$$
\textbf{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><strong>Comparison Relations</strong></td>
<td>$\leq$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>