CSE 431/531: Algorithm Analysis and Design (Spring 2020)

Introduction and Syllabus

Lecturer: Shi Li

Department of Computer Science and Engineering
University at Buffalo
Outline

1. Syllabus

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

3. Asymptotic Notations

4. Common Running times
CSE 431/531: Algorithm Analysis and Design

- Course Webpage (contains schedule, policies, homeworks and slides):
  http://www.cse.buffalo.edu/~shil/courses/CSE531/

- Please sign up course on Piazza via link on course webpage
  - announcements, polls, asking/answering questions
CSE 431/531: Algorithm Analysis and Design

- **Time and location:**
  - MoWeFr, 9:00-9:50am
  - Knox 110.

- **Instructor:**
  - Shi Li, shil@buffalo.edu, Davis 328
  - Office hours: TBD via poll
You **should** already have/know:
You should already have/know:

- Mathematical Background
- Reasoning, inductions, probabilities
You should already have/know:

- **Mathematical Background**
  - Reasoning, inductions, probabilities

- **Basic data Structures**
  - Stacks, queues, linked lists
You should already have/know:

- **Mathematical Background**
  - Reasoning, inductions, probabilities

- **Basic data Structures**
  - Stacks, queues, linked lists

- **Some Programming Experience**
  - C, C++, Java or Python
You Will Learn

- Classic algorithms for classic problems
- Sorting, shortest paths, minimum spanning tree, ...
You Will Learn

- Classic algorithms for classic problems
  - Sorting, shortest paths, minimum spanning tree, ···
- How to analyze algorithms
  - Correctness
  - Running time (efficiency)
  - Space requirement (occasionally)
You Will Learn

- Classic algorithms for classic problems
  - Sorting, shortest paths, minimum spanning tree, ···
- How to analyze algorithms
  - Correctness
  - Running time (efficiency)
  - Space requirement (occasionally)
- Meta techniques to design algorithms
  - Greedy algorithms
  - Divide and conquer
  - Dynamic programming
  - ···
You Will Learn

- Classic algorithms for classic problems
  - Sorting, shortest paths, minimum spanning tree, ...

- How to analyze algorithms
  - Correctness
  - Running time (efficiency)
  - Space requirement (occasionally)

- Meta techniques to design algorithms
  - Greedy algorithms
  - Divide and conquer
  - Dynamic programming
  - ...

- NP-completeness
Tentative Schedule (42 Lectures)

See the course webpage.
Textbook (Highly Recommended):

- Algorithm Design, 1st Edition, by Jon Kleinberg and Eva Tardos

Other Reference Books

Reading Before Classes

- Highly recommended: read the correspondent sections from the textbook (or reference book) before classes
  - Sections for each lecture can be found on the course webpage.
- Slides and example problems for recitations will be posted on the course webpage before class
Grading

- 40% for homeworks
  - 6 points $\times$ 5 theory homeworks
  - 10 points for programming homework
- 60% for mid-term + final exams, score for two exams is
  \[
  \max\{M \times 20\% + F \times 40\%, M \times 30\% + F \times 30\%\}
  \]
  \[
  M, F \in [0, 100]
  \]
For Homeworks, You Are Allowed to

- Use course materials (textbook, reference books, lecture notes, etc)
- Post questions on Piazza
- Ask me or TAs for hints
- Collaborate with classmates
  - Think about each problem for enough time before discussions
  - Must write down solutions on your own, in your own words
  - Write down names of students you collaborated with
For Homeworks, You Are Not Allowed to

- Use external resources
  - Can’t Google or ask questions online for solutions
  - Can’t read posted solutions from other algorithm course webpages
- Copy solutions from other students
For Programming Problems

- Need to implement the algorithms by yourself
- Can not copy codes from others or the Internet
- We use Moss (https://theory.stanford.edu/~aiken/moss/) to detect similarity of programs
Late Policy

- You have 1 “late credit”, using it allows you to submit an assignment solution for three days.
- With no special reasons, no other late submissions will be accepted.
- Mid-Term and Final Exam will be closed-book
- Per Departmental Policy on Academia Integrity Violations, penalty for AI violation is:
  - “F” for the course
  - lose financial support as TA/RA
  - case will be reported to the department and university
• Mid-Term and Final Exam will be closed-book
• Per Departmental Policy on Academia Integrity Violations, penalty for AI violation is:
  • “F” for the course
  • lose financial support as TA/RA
  • case will be reported to the department and university

Questions?
Outline

1 Syllabus

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

3 Asymptotic Notations

4 Common Running times
Outline

1. Syllabus

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

3. Asymptotic Notations

4. Common Running times
What is an Algorithm?

- Donald Knuth: An algorithm is a finite, definite effective procedure, with some input and some output.
What is an Algorithm?

- Donald Knuth: An algorithm is a finite, definite effective procedure, with some input and some output.

- Computational problem: specifies the input/output relationship.

- An algorithm solves a computational problem if it produces the correct output for any given input.
Examples

**Greatest Common Divisor**

**Input:** two integers $a, b > 0$

**Output:** the greatest common divisor of $a$ and $b$

Example:

Input: 210, 270
Output: 30

Algorithm: Euclidean algorithm

$\text{gcd}(270, 210) = \text{gcd}(210, 270 \mod 210) = \text{gcd}(210, 60)$

$\rightarrow (210, 60) \rightarrow (60, 30) \rightarrow (30, 0)$
## Greatest Common Divisor

**Input:** two integers $a, b > 0$

**Output:** the greatest common divisor of $a$ and $b$

### Example:

- Input: 210, 270
- Output: 30
Examples

Greatest Common Divisor

**Input:** two integers $a, b > 0$

**Output:** the greatest common divisor of $a$ and $b$

Example:

- Input: 210, 270
- Output: 30

- Algorithm: Euclidean algorithm
Examples

**Greatest Common Divisor**

**Input:** two integers $a, b > 0$

**Output:** the greatest common divisor of $a$ and $b$

**Example:**

- **Input:** 210, 270
- **Output:** 30

**Algorithm:** Euclidean algorithm

$$\text{gcd}(270, 210) = \text{gcd}(210, 270 \mod 210) = \text{gcd}(210, 60)$$
**Examples**

**Greatest Common Divisor**

**Input:** two integers $a, b > 0$

**Output:** the greatest common divisor of $a$ and $b$

**Example:**
- Input: 210, 270
- Output: 30

- **Algorithm:** Euclidean algorithm
  - $\text{gcd}(270, 210) = \text{gcd}(210, 270 \mod 210) = \text{gcd}(210, 60)$
  - $(270, 210) \rightarrow (210, 60) \rightarrow (60, 30) \rightarrow (30, 0)$
Examples

**Sorting**

**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 \(a'_1 \leq a'_2 \leq \cdots \leq a'_n\)
Examples

Sorting

**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
Examples

Sorting

**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

- Algorithms: insertion sort, merge sort, quicksort, …
### 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 $G$

![Graph Diagram](image-url)
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
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 \( b > 0 \)
2. \( (a, b) \leftarrow (b, a \mod b) \)
3. return \( a \)

**C++ program:**

```cpp
int Euclidean(int a, int b){
    int c;
    while (b > 0){
        c = b;
        b = a % b;
        a = c;
    }
    return a;
}
```
Main focus: correctness, running time (efficiency)
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)
  - ...

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?
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 $n$
2. $key \leftarrow A[j]$
3. $i \leftarrow j - 1$
4. while $i > 0$ and $A[i] > key$
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 \(n\)
2. \(key \leftarrow A[j]\)
3. \(i \leftarrow j - 1\)
4. while \(i > 0\) and \(A[i] > key\)
5. \(A[i + 1] \leftarrow A[i]\)
6. \(i \leftarrow i - 1\)
7. \(A[i + 1] \leftarrow key\)

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

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

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

1. for *j* ← 2 to *n*
2. \(key \leftarrow A[j]\)
3. \(i \leftarrow j - 1\)
4. while *i* > 0 and \(A[i] > key\)
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* ← 2 to *n*
2.    *key* ← *A*[*j*]
3.    *i* ← *j* − 1
4. while *i* > 0 and *A*[*i*] > *key*
5.    *A*[i + 1] ← *A*[i]
6.    *i* ← *i* − 1
7.  *A*[i + 1] ← *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 \( n \)
2. \( \text{key} \leftarrow A[j] \)
3. \( i \leftarrow j - 1 \)
4. while \( i > 0 \) and \( A[i] > \text{key} \)
   5. \( A[i + 1] \leftarrow A[i] \)
   6. \( i \leftarrow i - 1 \)
7. \( A[i + 1] \leftarrow \text{key} \)

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

12 21 35 53 59 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 $n$
2. \hspace{1em} $key \leftarrow A[j]$
3. \hspace{2em} $i \leftarrow j - 1$
4. \hspace{3em} while $i > 0$ and $A[i] > key$
5. \hspace{4em} $A[i + 1] \leftarrow A[i]$
6. \hspace{5em} $i \leftarrow i - 1$
7. \hspace{6em} $A[i + 1] \leftarrow key$

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

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 \( n \)
2. \( key \leftarrow A[j] \)
3. \( i \leftarrow j - 1 \)
4. while \( i > 0 \) and \( A[i] > key \)
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 \underline{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 $n$
2. \hspace{1em} key \leftarrow A[j]
3. \hspace{1em} $i \leftarrow j - 1$
4. \hspace{1em} while $i > 0$ and $A[i] > key$
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

↑
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 $n$
2. $key \leftarrow A[j]$
3. $i \leftarrow j - 1$
4. while $i > 0$ and $A[i] > key$
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

↑

i
Example:

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

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

1. for *j ← 2 to n*
2.  
   *key ← A[j]*
3.  
   *i ← j – 1*
4.  
   while *i > 0 and A[i] > key*
5.  
   *A[i + 1] ← A[i]*
6.  
   *i ← i – 1*
7.  
   *A[i + 1] ← 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 $n$
2. \hspace{1em} $key \leftarrow A[j]$
3. \hspace{1em} $i \leftarrow j - 1$
4. \hspace{1em} while $i > 0$ and $A[i] > key$
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 15 21 35 53 59

$\uparrow$

$i$
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
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 $j = 6 : 12, 15, 21, 35, 53, 59$
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: \# 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.
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 \( n \)
Analyzing Running Time of Insertion Sort

- Q3: How fast is the computer?
- Q4: Programming language?
Analyzing Running Time of Insertion Sort

- Q3: How fast is the computer?
- Q4: Programming language?
- A: They do not matter!
Analyzing Running Time of Insertion Sort

- 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
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

$3n^3 + 2n^2 - 18n + 1028 \Rightarrow 3n^3 \Rightarrow n^3$

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

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

- $3n^3 + 2n^2 - 18n + 1028 \Rightarrow 3n^3 \Rightarrow n^3$
- $3n^3 + 2n^2 - 18n + 1028 = O(n^3)$
- $n^2/100 - 3n^2 + 10 \Rightarrow n^2/100 \Rightarrow n^2$
Informal way to define $O$-notation:

- Ignoring lower order terms
- Ignoring leading constant

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

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

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

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

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

$O$-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

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

\( O \)-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
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 $n$!
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
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
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 $n$
2. \hspace{1em} key $\leftarrow A[j]$
3. \hspace{1em} $i \leftarrow j - 1$
4. \hspace{1em} while $i > 0$ and $A[i] > key$
5. \hspace{2em} $A[i + 1] \leftarrow A[i]$
6. \hspace{1em} $i \leftarrow i - 1$
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 = $\sum_{n=2}^{j} O(j) = O(\sum_{n=2}^{j} j) = O(n^2 - 1) = O(n^2)$
Asymptotic Analysis of Insertion Sort

Insertion-Sort($A, n$)

1. for $j \leftarrow 2$ to $n$
2. $key \leftarrow A[j]$
3. $i \leftarrow j - 1$
4. while $i > 0$ and $A[i] > key$
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

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

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

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*
2.  
3.  
4. while *i* > 0 and *A*[*i*] > *key*
5.  
6.  
7.  

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

Answer: *O*(*j*)

Total running time = \[\sum_{j=2}^{n} O(j) = O\left(\sum_{j=2}^{n} j\right)\]

= \[O\left(\frac{n(n+1)}{2} - 1\right) = O(n^2)\]
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, where $c \geq 1$ is large enough.

Reason: often we need to read the integer $n$ and handle integers within range $[-nc, nc]$, 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.
Random-Access Machine (RAM) model
- reading and writing $A[j]$ 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
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
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?
  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.
- 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
Remember to sign up for Piazza.

Questions?
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. \( f : \mathbb{N} \to \mathbb{R} \) 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 $n$. 
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 \( n \).

- \( n^2 - n - 30 \)
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 \( n \).

- \( n^2 - n - 30 \) \hspace{1cm} 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. \( f : \mathbb{N} \to \mathbb{R} \) 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
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
- $2^n - n^{20} \quad$ Yes
- $100n - n^2/10 + 50 \quad$?
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
- $100n - n^2/10 + 50$?   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.
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 \( n \).
- \( n^2 - n - 30 \) \hspace{1cm} Yes
- \( 2^n - n^{20} \) \hspace{1cm} Yes
- \( 100n - n^2/10 + 50 \) \hspace{1cm} No

- We only consider asymptotically positive functions.
- Why not (everywhere-)positive functions? Answer: for the sake of convenience.
$O$-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 $n$. 
\textbf{O-Notation: Asymptotic Upper Bound}

\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 \( c > 0 \) 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,

\[
3n^2 + 2n - c(n^2 - 10n) = 3n^2 + 2n - 4(n^2 - 10n)
\]

\[
= -n^2 + 40n \leq 0.
\]

\[
3n^2 + 2n \leq c(n^2 - 10n)
\]
**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 $n$.

- $3n^2 + 2n \in O(n^2 - 10n)$
- $3n^2 + 2n \in O(n^3 - 5n^2)$
- $n^{100} \in O(2^n)$
- $n^3 \notin O(10n^2)$
**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 $n$.

- $3n^2 + 2n \in O(n^2 - 10n)$
- $3n^2 + 2n \in O(n^3 - 5n^2)$
- $n^{100} \in O(2^n)$
- $n^3 \notin O(10n^2)$

<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>
Conventions

- 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 “$f(n) \in O(g(n))$”
- $3n^2 + 2n = O(n^3 - 10n)$
- $3n^2 + 2n = O(n^2 + 5n)$
- $3n^2 + 2n = O(n^2)$
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.
\( \Omega \)-Notation: Asymptotic Lower Bound

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

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

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

\[
\Omega(g(n)) = \{ \text{function } f : \exists c > 0, n_0 > 0 \text{ such that } f(n) \geq cg(n), \forall n \geq n_0 \}.
\]
Ω-Notation: Asymptotic Lower Bound

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

\textbf{Ω-Notation}  For a function \( g(n) \),
\[ \Omega(g(n)) = \{ \text{function } f : \exists c > 0, n_0 > 0 \text{ such that } f(n) \geq cg(n), \forall n \geq n_0 \}. \]

- In other words, \( f(n) \in \Omega(g(n)) \) if \( f(n) \geq cg(n) \) for some \( c \) and large enough \( n \).
**Ω-Notation: Asymptotic Lower Bound**

**Ω-Notation** For a function $g(n)$,

$$
Ω(g(n)) = \{ \text{function } f : \exists c > 0, n_0 > 0 \text{ such that } f(n) \geq cg(n), \forall n \geq n_0 \}. 
$$
Again, we use “=” instead of ∈.

- $4n^2 = \Omega(n - 10)$
- $3n^2 - n + 10 = \Omega(n^2 - 20)$
ω-Notation: Asymptotic Lower Bound

- Again, we use “=” instead of ∈.
  - $4n^2 = \Omega(n - 10)$
  - $3n^2 - n + 10 = \Omega(n^2 - 20)$

<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>$\geq$</td>
<td></td>
</tr>
</tbody>
</table>
\(\Omega\)-Notation: Asymptotic Lower Bound

- Again, we use “=” instead of \(\varepsilon\).
  - \(4n^2 = \Omega(n - 10)\)
  - \(3n^2 - n + 10 = \Omega(n^2 - 20)\)

<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>(\geq)</td>
<td></td>
</tr>
</tbody>
</table>

**Theorem** \(f(n) = O(g(n)) \iff g(n) = \Omega(f(n))\).
\( \Theta \)-Notation: Asymptotic Tight Bound

\( \Theta \)-Notation  For a function \( g(n) \),

\[
\Theta(g(n)) = \{ \text{function } f : \exists c_2 \geq c_1 > 0, n_0 > 0 \text{ such that } c_1 g(n) \leq f(n) \leq c_2 g(n), \forall n \geq n_0 \}. 
\]
**Θ-Notation: Asymptotic Tight Bound**

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

$$\Theta(g(n)) = \{ \text{function } f : \exists c_2 \geq c_1 > 0, n_0 > 0 \text{ such that }\]

$$c_1 g(n) \leq f(n) \leq c_2 g(n), \forall n \geq n_0 \}.$$  

- $f(n) = \Theta(g(n))$, then for large enough $n$, we have  
  “$f(n) \approx g(n)$”.
**Θ-Notation: Asymptotic Tight Bound**

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

$$\Theta(g(n)) = \{ \text{function } f : \exists c_2 \geq c_1 > 0, n_0 > 0 \text{ such that}$$

$$c_1 g(n) \leq f(n) \leq c_2 g(n), \forall n \geq n_0 \}.$$

- $f(n) = \Theta(g(n))$, then for large enough $n$, we have 
  "$f(n) \approx g(n)$".
**Θ-Notation: Asymptotic Tight Bound**

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

\[
\Theta(g(n)) = \{ \text{function } f : \exists c_2 \geq c_1 > 0, n_0 > 0 \text{ such that } c_1 g(n) \leq f(n) \leq c_2 g(n), \forall n \geq n_0 \}.
\]

- \( 3n^2 + 2n = \Theta(n^2 - 20n) \)
**Θ-Notation:** Asymptotic Tight Bound

**Θ-Notation** For a function $g(n)$,

$$\Theta(g(n)) = \{ \text{function } f : \exists c_2 \geq c_1 > 0, n_0 > 0 \text{ such that } c_1 g(n) \leq f(n) \leq c_2 g(n), \forall n \geq n_0 \}.$$ 

- $3n^2 + 2n = \Theta(n^2 - 20n)$
- $2^{n/3} + 100 = \Theta(2^{n/3})$
**Θ-Notation**  For a function $g(n)$,

$$\Theta(g(n)) = \{ \text{function } f : \exists c_2 \geq c_1 > 0, n_0 > 0 \text{ such that } c_1 g(n) \leq f(n) \leq c_2 g(n), \forall n \geq n_0 \}.$$ 

- $3n^2 + 2n = \Theta(n^2 - 20n)$
- $2^{n/3} + 100 = \Theta(2^{n/3})$

<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>$\geq$</td>
<td>$=$</td>
</tr>
</tbody>
</table>
**Θ-Notation: Asymptotic Tight Bound**

**Θ-Notation** For a function $g(n)$,

$\Theta(g(n)) = \{ \text{function } f : \exists c_2 \geq c_1 > 0, n_0 > 0 \text{ such that} \]

$$c_1g(n) \leq f(n) \leq c_2g(n), \forall n \geq n_0 \}.$$

- $3n^2 + 2n = \Theta(n^2 - 20n)$
- $2^{n/3+100} = \Theta(2^{n/3})$

<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>$\geq$</td>
<td>$=$</td>
</tr>
</tbody>
</table>

**Theorem** $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$. 
<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>$\geq$</td>
<td>$=$</td>
</tr>
</tbody>
</table>
Asymptotic Notations

<table>
<thead>
<tr>
<th></th>
<th>$O$</th>
<th>$\Omega$</th>
<th>$\Theta$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comparison Relations</td>
<td>$\leq$</td>
<td>$\geq$</td>
<td>$=$</td>
</tr>
</tbody>
</table>

Trivial Facts on Comparison Relations

- $f \leq g \iff g \geq f$
- $f = g \iff f \leq g \text{ and } f \geq g$
- $f \leq g \text{ or } f \geq g$
<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>$\geq$</td>
<td>$=$</td>
</tr>
</tbody>
</table>

### Trivial Facts on Comparison Relations

- $f \leq g \iff g \geq f$
- $f = g \iff f \leq g$ and $f \geq g$
- $f \leq g$ or $f \geq g$

### Correct Analogies

- $f(n) = O(g(n)) \iff g(n) = \Omega(f(n))$
- $f(n) = \Theta(g(n)) \iff f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$
<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>$\geq$</td>
<td>$=$</td>
</tr>
</tbody>
</table>

**Trivial Facts on Comparison Relations**

- $f \leq g \iff g \geq f$
- $f = g \iff f \leq g \text{ and } f \geq g$
- $f \leq g \text{ or } f \geq g$

**Correct Analogies**

- $f(n) = O(g(n)) \iff g(n) = \Omega(f(n))$
- $f(n) = \Theta(g(n)) \iff f(n) = O(g(n)) \text{ and } f(n) = \Omega(g(n))$

**Incorrect Analogy**

- $f(n) = O(g(n)) \text{ or } g(n) = O(f(n))$
Incorrect Analogy

- \( f(n) = O(g(n)) \) or \( g(n) = O(f(n)) \)
Incorrect Analogy

- $f(n) = O(g(n))$ or $g(n) = O(f(n))$

\[
\begin{align*}
  f(n) &= n^2 \\
  g(n) &= \begin{cases} 
  1 & \text{if } n \text{ is odd} \\
  n^3 & \text{if } n \text{ is even}
\end{cases}
\end{align*}
\]
Recall: Informal way to define $O$-notation

- ignoring lower order terms: $3n^2 - 10n - 5 \rightarrow 3n^2$
- ignoring leading constant: $3n^2 \rightarrow n^2$
Recall: Informal way to define $O$-notation

- ignoring lower order terms: $3n^2 - 10n - 5 \rightarrow 3n^2$
- ignoring leading constant: $3n^2 \rightarrow n^2$
- $3n^2 - 10n - 5 = O(n^2)$
Recall: Informal way to define $O$-notation

- ignoring lower order terms: $3n^2 - 10n - 5 \rightarrow 3n^2$
- ignoring leading constant: $3n^2 \rightarrow n^2$
- $3n^2 - 10n - 5 = O(n^2)$
- Indeed, $3n^2 - 10n - 5 = \Omega(n^2), 3n^2 - 10n - 5 = \Theta(n^2)$
Recall: Informal way to define $O$-notation

- ignoring lower order terms: $3n^2 - 10n - 5 \rightarrow 3n^2$
- ignoring leading constant: $3n^2 \rightarrow n^2$
- $3n^2 - 10n - 5 = O(n^2)$
- Indeed, $3n^2 - 10n - 5 = \Omega(n^2), 3n^2 - 10n - 5 = \Theta(n^2)$

In the formal definition of $O(\cdot)$, nothing tells us to ignore lower order terms and leading constant.
Recall: Informal way to define $O$-notation

- ignoring lower order terms: $3n^2 - 10n - 5 \to 3n^2$
- ignoring leading constant: $3n^2 \to n^2$
- $3n^2 - 10n - 5 = O(n^2)$
- Indeed, $3n^2 - 10n - 5 = \Omega(n^2), 3n^2 - 10n - 5 = \Theta(n^2)$
- In the formal definition of $O(\cdot)$, nothing tells us to ignore lower order terms and leading constant.
- $3n^2 - 10n - 5 = O(5n^2 - 6n + 5)$ is correct, though weird
Recall: Informal way to define $O$-notation

- ignoring lower order terms: $3n^2 - 10n - 5 \rightarrow 3n^2$
- ignoring leading constant: $3n^2 \rightarrow n^2$
- $3n^2 - 10n - 5 = O(n^2)$
- Indeed, $3n^2 - 10n - 5 = \Omega(n^2), 3n^2 - 10n - 5 = \Theta(n^2)$

In the formal definition of $O(\cdot)$, nothing tells us to ignore lower order terms and leading constant.
- $3n^2 - 10n - 5 = O(5n^2 - 6n + 5)$ is correct, though weird
- $3n^2 - 10n - 5 = O(n^2)$ is the most natural since $n^2$ is the simplest term we can have inside $O(\cdot)$. 
Notice that $O$ denotes asymptotic upper bound

- $n^2 + 2n = O(n^3)$ is correct.
- The following sentence is correct: the running time of the insertion sort algorithm is $O(n^4)$.
- We say: the running time of the insertion sort algorithm is $O(n^2)$ and the bound is tight.
Notice that $O$ denotes asymptotic upper bound

- $n^2 + 2n = O(n^3)$ is correct.
- The following sentence is correct: the running time of the insertion sort algorithm is $O(n^4)$.
- We say: the running time of the insertion sort algorithm is $O(n^2)$ and the bound is tight.
- We do not use $\Omega$ and $\Theta$ very often when we talk about running times.
Exercise

For each pair of functions \( f, g \) in the following table, indicate whether \( f \) is \( O, \Omega \) or \( \Theta \) of \( g \).

<table>
<thead>
<tr>
<th>( f )</th>
<th>( g )</th>
<th>( O )</th>
<th>( \Omega )</th>
<th>( \Theta )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( n^3 - 100n )</td>
<td>( 5n^2 + 3n )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( 3n - 50 )</td>
<td>( n^2 - 7n )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( n^2 - 100n )</td>
<td>( 5n^2 + 30n )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \log_2 n )</td>
<td>( \log_{10} n )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \log^{10} n )</td>
<td>( n^{0.1} )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( 2^n )</td>
<td>( 2^{n/2} )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \sqrt{n} )</td>
<td>( n^{\sin n} )</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Exercise

For each pair of functions $f, g$ in the following table, indicate whether $f$ is $O$, $\Omega$ or $\Theta$ of $g$.

<table>
<thead>
<tr>
<th></th>
<th>$f$</th>
<th>$g$</th>
<th>$O$</th>
<th>$\Omega$</th>
<th>$\Theta$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$n^3 - 100n$</td>
<td>$5n^2 + 3n$</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>$3n - 50$</td>
<td>$n^2 - 7n$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$n^2 - 100n$</td>
<td>$5n^2 + 30n$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$\log_2 n$</td>
<td>$\log_{10} n$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$\log_{10} n$</td>
<td>$n^{0.1}$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$2^n$</td>
<td>$2^{n/2}$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$\sqrt{n}$</td>
<td>$n^{\sin n}$</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Exercise

For each pair of functions $f, g$ in the following table, indicate whether $f$ is $O$, $\Omega$ or $\Theta$ of $g$.

<table>
<thead>
<tr>
<th>$f$</th>
<th>$g$</th>
<th>$O$</th>
<th>$\Omega$</th>
<th>$\Theta$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n^3 - 100n$</td>
<td>$5n^2 + 3n$</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>$3n - 50$</td>
<td>$n^2 - 7n$</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>$n^2 - 100n$</td>
<td>$5n^2 + 30n$</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>$\log_2 n$</td>
<td>$\log_{10} n$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\log^{10} n$</td>
<td>$n^{0.1}$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$2^n$</td>
<td>$2^{n/2}$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\sqrt{n}$</td>
<td>$n^{\sin n}$</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Exercise

For each pair of functions \( f, g \) in the following table, indicate whether \( f \) is \( O, \Omega \) or \( \Theta \) of \( g \).

<table>
<thead>
<tr>
<th>( f )</th>
<th>( g )</th>
<th>( O )</th>
<th>( \Omega )</th>
<th>( \Theta )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( n^3 - 100n )</td>
<td>( 5n^2 + 3n )</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>( 3n - 50 )</td>
<td>( n^2 - 7n )</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>( n^2 - 100n )</td>
<td>( 5n^2 + 30n )</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>( \log_2 n )</td>
<td>( \log_{10} n )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \log^{10} n )</td>
<td>( n^{0.1} )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( 2^n )</td>
<td>( 2^{n/2} )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \sqrt{n} )</td>
<td>( n^{\sin n} )</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Exercise

For each pair of functions $f, g$ in the following table, indicate whether $f$ is $O$, $\Omega$ or $\Theta$ of $g$.

<table>
<thead>
<tr>
<th>$f$</th>
<th>$g$</th>
<th>$O$</th>
<th>$\Omega$</th>
<th>$\Theta$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n^3 - 100n$</td>
<td>$5n^2 + 3n$</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>$n^5$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$3n - 50$</td>
<td>$n^2 - 7n$</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>$n^2 - 100n$</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>$5n^2 + 30n$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\log_2 n$</td>
<td>$\log_{10} n$</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>$\log^{10} n$</td>
<td>$n^{0.1}$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$2^n$</td>
<td>$2^{n/2}$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\sqrt{n}$</td>
<td>$n^{\sin n}$</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

We often use $\log n$ for $\log_2 n$. But for $O(\log n)$, the base is not important.
## Exercise

For each pair of functions \( f, g \) in the following table, indicate whether \( f \) is \( O, \Omega \) or \( \Theta \) of \( g \).

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>( O )</th>
<th>( \Omega )</th>
<th>( \Theta )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( n^3 - 100n )</td>
<td>( 5n^2 + 3n )</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>( 3n - 50 )</td>
<td>( n^2 - 7n )</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>( n^2 - 100n )</td>
<td>( 5n^2 + 30n )</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>( \log_2 n )</td>
<td>( \log_{10} n )</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>( \log^{10} n )</td>
<td>( n^{0.1} )</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>( 2^n )</td>
<td>( 2^{n/2} )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \sqrt{n} )</td>
<td>( n^{\sin n} )</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

We often use \( \log n \) for \( \log_2 n \). But for \( O(\log n) \), the base is not important.
Exercise

For each pair of functions $f, g$ in the following table, indicate whether $f$ is $O$, $\Omega$ or $\Theta$ of $g$.

<table>
<thead>
<tr>
<th>$f$</th>
<th>$g$</th>
<th>$O$</th>
<th>$\Omega$</th>
<th>$\Theta$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n^3 - 100n$</td>
<td>$5n^2 + 3n$</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>$3n - 50$</td>
<td>$n^2 - 7n$</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>$n^2 - 100n$</td>
<td>$5n^2 + 30n$</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>$\log_2 n$</td>
<td>$\log_{10} n$</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>$\log_{10} n$</td>
<td>$n^{0.1}$</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>$2^n$</td>
<td>$2^{n/2}$</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>$\sqrt{n}$</td>
<td>$n^{\sin n}$</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

We often use $\log n$ for $\log_2 n$. But for $O(\log n)$, the base is not important.
Exercise

For each pair of functions \( f, g \) in the following table, indicate whether \( f \) is \( O, \Omega \) or \( \Theta \) of \( g \).

<table>
<thead>
<tr>
<th>( f )</th>
<th>( g )</th>
<th>( O )</th>
<th>( \Omega )</th>
<th>( \Theta )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( n^3 - 100n )</td>
<td>( 5n^2 + 3n )</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>( 3n - 50 )</td>
<td>( n^2 - 7n )</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>( n^2 - 100n )</td>
<td>( 5n^2 + 30n )</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>( \log_2 n )</td>
<td>( \log_{10} n )</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>( \log_{10} n )</td>
<td>( n^{0.1} )</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>( 2^n )</td>
<td>( 2^{n/2} )</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>( \sqrt{n} )</td>
<td>( n^{\sin n} )</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

We often use \( \log n \) for \( \log_2 n \). But for \( O(\log n) \), the base is not important.
<table>
<thead>
<tr>
<th>Asymptotic Notations</th>
<th>( O )</th>
<th>( \Omega )</th>
<th>( \Theta )</th>
<th>( o )</th>
<th>( \omega )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comparison Relations</td>
<td>( \leq )</td>
<td>( \geq )</td>
<td>( = )</td>
<td>( &lt; )</td>
<td>( &gt; )</td>
</tr>
<tr>
<td>Asymptotic Notations</td>
<td>$O$</td>
<td>$\Omega$</td>
<td>$\Theta$</td>
<td>$o$</td>
<td>$\omega$</td>
</tr>
<tr>
<td>----------------------</td>
<td>-----</td>
<td>---------</td>
<td>---------</td>
<td>-----</td>
<td>-------</td>
</tr>
<tr>
<td>Comparison Relations</td>
<td>$\leq$</td>
<td>$\geq$</td>
<td>$=$</td>
<td>$&lt;$</td>
<td>$&gt;$</td>
</tr>
</tbody>
</table>

Questions?
Outline

1. Syllabus

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

3. Asymptotic Notations

4. Common Running times
$O(n)$ (Linear) Running Time

Computing the sum of $n$ numbers

**sum($A, n$)**

1. $S \leftarrow 0$
2. for $i \leftarrow 1$ to $n$
3. $S \leftarrow S + A[i]$
4. return $S$
$O(n)$ (Linear) Running Time

Merge two sorted arrays

\[
\begin{array}{cccccc}
3 & 8 & 12 & 20 & 32 & 48 \\
5 & 7 & 9 & 25 & 29 \\
\end{array}
\]
Merge two sorted arrays
$O(n)$ (Linear) Running Time

- Merge two sorted arrays

\[
\begin{array}{cccccc}
3 & 8 & 12 & 20 & 32 & 48 \\
\end{array}
\]

\[
\begin{array}{cccccc}
5 & 7 & 9 & 25 & 29 \\
\end{array}
\]

\[
\begin{array}{c}
3 \\
\end{array}
\]
$O(n)$ (Linear) Running Time

- Merge two sorted arrays

3 8 12 20 32 48

5 7 9 25 29

3
Merge two sorted arrays

\[ 3 \quad 8 \quad 12 \quad 20 \quad 32 \quad 48 \]
\[ 5 \quad 7 \quad 9 \quad 25 \quad 29 \]
\[ 3 \quad 5 \]
$O(n)$ (Linear) Running Time

- Merge two sorted arrays

```
  3 8 12 20 32 48
  5 7 9 25 29
  3 5
```
Merge two sorted arrays

3 8 12 20 32 48
5 7 9 25 29
3 5 7
$O(n)$ (Linear) Running Time

- Merge two sorted arrays

3 8 12 20 32 48
5 7 9 25 29
3 5 7
$O(n)$ (Linear) Running Time

- Merge two sorted arrays

\[
\begin{array}{cccccc}
3 & 8 & 12 & 20 & 32 & 48 \\
5 & 7 & 9 & 25 & 29 \\
3 & 5 & 7 & 8 \\
\end{array}
\]
Merge two sorted arrays
$O(n)$ (Linear) Running Time

- Merge two sorted arrays

<table>
<thead>
<tr>
<th>3</th>
<th>8</th>
<th>12</th>
<th>20</th>
<th>32</th>
<th>48</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>7</td>
<td>9</td>
<td>25</td>
<td>29</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>12</td>
</tr>
</tbody>
</table>
**$O(n)$ (Linear) Running Time**

- Merge two sorted arrays

```
3  8 12 20 32 48
5  7  9 25 29
3  5  7  8  9 12 20 25 29 32 48
```
**$O(n)$ (Linear) Running Time**

```
merge(B, C, n_1, n_2) \ \ \ \ \ \ B and C are sorted, with length $n_1$ and $n_2$

1. $A \leftarrow []$; $i \leftarrow 1$; $j \leftarrow 1$
2. while $i \leq n_1$ and $j \leq n_2$
   3. if $(B[i] \leq C[j])$ then
      4. append $B[i]$ to $A$; $i \leftarrow i + 1$
   5. else
      6. append $C[j]$ to $A$; $j \leftarrow j + 1$
7. if $i \leq n_1$ then append $B[i..n_1]$ to $A$
8. if $j \leq n_2$ then append $C[j..n_2]$ to $A$
9. return $A$
```
**O(n) (Linear) Running Time**

\[
\text{merge}(B, C, n_1, n_2) \quad \text{\textbackslash\textbackslash B and C are sorted, with length } n_1 \text{ and } n_2
\]

1. \( A \leftarrow []; \ i \leftarrow 1; \ j \leftarrow 1 \)
2. \( \text{while } i \leq n_1 \text{ and } j \leq n_2 \)
3. \( \text{if } (B[i] \leq C[j]) \text{ then} \)
4. \( \text{append } B[i] \text{ to } A; \ i \leftarrow i + 1 \)
5. \( \text{else} \)
6. \( \text{append } C[j] \text{ to } A; \ j \leftarrow j + 1 \)
7. \( \text{if } i \leq n_1 \text{ then append } B[i..n_1] \text{ to } A \)
8. \( \text{if } j \leq n_2 \text{ then append } C[j..n_2] \text{ to } A \)
9. \( \text{return } A \)

Running time = \( O(n) \) where \( n = n_1 + n_2 \).
$O(n \log n)$ Running Time

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

1. if $n = 1$ then
2. return $A$
3. else
4. $B \leftarrow$ merge-sort($A[1..\lfloor n/2 \rfloor], \lfloor n/2 \rfloor$)
5. $C \leftarrow$ merge-sort($A[\lceil n/2 \rceil + 1..n], n - \lceil n/2 \rceil$)
6. return merge($B, C, \lfloor n/2 \rfloor, n - \lfloor n/2 \rfloor$)
$O(n \log n)$ Running Time

- **Merge-Sort**

![Diagram showing the divide-and-conquer structure of Merge-Sort](image-url)
**$O(n \log n)$ Running Time**

- **Merge-Sort**

  ![Merge-Sort Tree Diagram]

  - Each level takes running time $O(n)$
$O(n \log n)$ Running Time

- Merge-Sort

Each level takes running time $O(n)$
- There are $O(\log n)$ levels
$O(n \log n)$ Running Time

- **Merge-Sort**

![Diagram of Merge-Sort](image)

- Each level takes running time $O(n)$
- There are $O(\log n)$ levels
- Running time = $O(n \log n)$
Closest Pair

**Input:** $n$ points in plane: $(x_1, y_1), (x_2, y_2), \cdots, (x_n, y_n)$

**Output:** the pair of points that are closest
Closest Pair

**Input:** \( n \) points in plane: \((x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\)

**Output:** the pair of points that are closest
Closest Pair

**Input:** $n$ points in plane: $(x_1, y_1), (x_2, y_2), \cdots, (x_n, y_n)$

**Output:** the pair of points that are closest

\[
\text{closest-pair}(x, y, n)
\]

1. $bestd \leftarrow \infty$
2. for $i \leftarrow 1$ to $n - 1$
3. \hspace{.5cm} for $j \leftarrow i + 1$ to $n$
4. \hspace{1.5cm} $d \leftarrow \sqrt{(x[i] - x[j])^2 + (y[i] - y[j])^2}$
5. \hspace{1.5cm} if $d < bestd$ then
6. \hspace{2cm} $besti \leftarrow i, bestj \leftarrow j, bestd \leftarrow d$
7. return $(besti, bestj)$

$O(n^2)$ (Quadratic) Running Time

Closest pair can be solved in $O(n \log n)$ time!
**$O(n^2)$ (Quadratic) Running Time**

**Closest Pair**

**Input:** $n$ points in plane: $(x_1, y_1), (x_2, y_2), \cdots, (x_n, y_n)$

**Output:** the pair of points that are closest

```
closest-pair(x, y, n)

1. bestd ← ∞
2. for i ← 1 to n - 1
3.     for j ← i + 1 to n
4.         d ← $\sqrt{(x[i] - x[j])^2 + (y[i] - y[j])^2}$
5.         if d < bestd then
6.             besti ← i, bestj ← j, bestd ← d
7. end if
8. end for
9. end for
10. return (besti, bestj)
```

Closest pair can be solved in $O(n \log n)$ time!
$O(n^3)$ (Cubic) Running Time

Multiply two matrices of size $n \times n$

```
matrix-multiplication(A, B, n)

1. $C \leftarrow$ matrix of size $n \times n$, with all entries being 0
2. for $i \leftarrow 1$ to $n$
3.     for $j \leftarrow 1$ to $n$
4.         for $k \leftarrow 1$ to $n$
5.             $C[i, k] \leftarrow C[i, k] + A[i, j] \times B[j, k]$
6. return $C$
```
Def. An independent set of a graph $G = (V, E)$ is a subset $S \subseteq V$ of vertices such that for every $u, v \in S$, we have $(u, v) \notin E$. 
Def. An independent set of a graph $G = (V, E)$ is a subset $S \subseteq V$ of vertices such that for every $u, v \in S$, we have $(u, v) \notin E$.
Def. An independent set of a graph $G = (V, E)$ is a subset $S \subseteq V$ of vertices such that for every $u, v \in S$, we have $(u, v) \notin E$. 

$O(n^k)$ Running Time for Integer $k \geq 4$
$O(n^k)$ Running Time for Integer $k \geq 4$

**Def.** An independent set of a graph $G = (V, E)$ is a subset $S \subseteq V$ of vertices such that for every $u, v \in S$, we have $(u, v) \notin E$.

**Input:** graph $G = (V, E)$

**Output:** whether there is an independent set of size $k$
$O(n^k)$ Running Time for Integer $k \geq 4$

### Independent Set of Size $k$

**Input:** graph $G = (V, E)$

**Output:** whether there is an independent set of size $k$

#### independent-set($G = (V, E)$)

1. for every set $S \subseteq V$ of size $k$
2. $b \leftarrow true$
3. for every $u, v \in S$
4. if $(u, v) \in E$ then $b \leftarrow false$
5. if $b$ return true
6. return false

Running time = $O\left(\frac{n^k}{k!} \times k^2\right) = O(n^k)$ (assume $k$ is a constant)
Beyond Polynomial Time: $2^n$

**Maximum Independent Set Problem**

**Input:** graph $G = (V, E)$

**Output:** the maximum independent set of $G$

---

**max-independent-set**($G = (V, E)$)

1. $R \leftarrow \emptyset$
2. for every set $S \subseteq V$
3. \hspace{1em} $b \leftarrow \text{true}$
4. \hspace{1em} for every $u, v \in S$
5. \hspace{2em} if $(u, v) \in E$ then $b \leftarrow \text{false}$
6. \hspace{1em} if $b$ and $|S| > |R|$ then $R \leftarrow S$
7. return $R$

Running time $= O(2^n n^2)$. 
Beyond Polynomial Time: $n!$

Hamiltonian Cycle Problem

**Input:** a graph with $n$ vertices

**Output:** a cycle that visits each node exactly once, or say no such cycle exists
Beyond Polynomial Time: \( n! \)

Hamiltonian Cycle Problem

**Input:** a graph with \( n \) vertices

**Output:** a cycle that visits each node exactly once, or say no such cycle exists.
Beyond Polynomial Time: $n!$

**Hamiltonian**($G = (V, E)$)

1. for every permutation $(p_1, p_2, \cdots, p_n)$ of $V$
2. $b \leftarrow \text{true}$
3. for $i \leftarrow 1$ to $n - 1$
   4. if $(p_i, p_{i+1}) \notin E$ then $b \leftarrow \text{false}$
5. if $(p_n, p_1) \notin E$ then $b \leftarrow \text{false}$
6. if $b$ then return $(p_1, p_2, \cdots, p_n)$
7. return “No Hamiltonian Cycle”

Running time = $O(n! \times n)$
$O(\log n)$ (Logarithmic) Running Time

Binary search

Input: sorted array $A$ of size $n$, an integer $t$;
Output: whether $t$ appears in $A$.

E.g., search 35 in the following array:
**$O(\log n)$ (Logarithmic) Running Time**

- Binary search
  - Input: sorted array $A$ of size $n$, an integer $t$;
  - Output: whether $t$ appears in $A$.  

E.g., search 35 in the following array:
O(\log n) (Logarithmic) Running Time

- Binary search
  - Input: sorted array \( A \) of size \( n \), an integer \( t \);
  - Output: whether \( t \) appears in \( A \).
- E.g, search 35 in the following array:
$O(\log n)$ (Logarithmic) Running Time

- Binary search
  - Input: sorted array $A$ of size $n$, an integer $t$;
  - Output: whether $t$ appears in $A$.
- E.g, search 35 in the following array:

```
3 8 10 25 29 37 38 42 46 52 59 61 63 75 79
```
$O(\log n)$ (Logarithmic) Running Time

- Binary search
  - Input: sorted array $A$ of size $n$, an integer $t$;
  - Output: whether $t$ appears in $A$.
- E.g, search 35 in the following array:
$O(\log n)$ (Logarithmic) Running Time

- Binary search
  - Input: sorted array $A$ of size $n$, an integer $t$;
  - Output: whether $t$ appears in $A$.
- E.g, search 35 in the following array:

```
3 8 10 25 29 37 38 42 46 52 59 61 63 75 79
```

$42 > 35$
$O(\log n)$ (Logarithmic) Running Time

- Binary search
  - Input: sorted array $A$ of size $n$, an integer $t$;
  - Output: whether $t$ appears in $A$.
- E.g, search 35 in the following array:

```
3 8 10 25 29 37 38 42 46 52 59 61 63 75 79
```
**$O(\log n)$ (Logarithmic) Running Time**

- **Binary search**
  - **Input:** sorted array $A$ of size $n$, an integer $t$;
  - **Output:** whether $t$ appears in $A$.

- **E.g., search 35 in the following array:**

```
3  8  10  25  29  37  38  42  46  52  59  61  63  75  79
```
$O(\log n)$ (Logarithmic) Running Time

- **Binary search**
  - Input: sorted array $A$ of size $n$, an integer $t$;
  - Output: whether $t$ appears in $A$.

- E.g., search 35 in the following array:

```
3 8 10 25 29 37 38 42 46 52 59 61 63 75 79
```

25 < 35
$O(\log n)$ (Logarithmic) Running Time

- Binary search
  - Input: sorted array $A$ of size $n$, an integer $t$;
  - Output: whether $t$ appears in $A$.
- E.g, search 35 in the following array:
**$O(\log n)$ (Logarithmic) Running Time**

- Binary search
  - Input: sorted array $A$ of size $n$, an integer $t$;
  - Output: whether $t$ appears in $A$.
- E.g, search 35 in the following array:
**$O(\log n)$ (Logarithmic) Running Time**

- **Binary search**
  - Input: sorted array $A$ of size $n$, an integer $t$;
  - Output: whether $t$ appears in $A$.
- E.g., search 35 in the following array:

```
3  8  10  25  29  37  38  42  46  52  59  61  63  75  79
```

37 > 35

37 > 35
**O**(log \( n \)) (Logarithmic) Running Time

- Binary search
  - Input: sorted array \( A \) of size \( n \), an integer \( t \);
  - Output: whether \( t \) appears in \( A \).
- E.g, search 35 in the following array:

```
3  8  10  25  29  37  38  42  46  52  59  61  63  75  79
```
**O(\log n) (Logarithmic) Running Time**

Binary search

- Input: sorted array \( A \) of size \( n \), an integer \( t \);
- Output: whether \( t \) appears in \( A \).

```plaintext
binary-search(A, n, t)
1 i ← 1, j ← n
2 while \( i \leq j \) do
3 \( k ← \lfloor (i + j)/2 \rfloor \)
4 if \( A[k] = t \) return true
5 if \( A[k] < t \) then \( j ← k - 1 \) else \( i ← k + 1 \)
6 return false
```
$O(\log n)$ (Logarithmic) Running Time

Binary search
- Input: sorted array $A$ of size $n$, an integer $t$;
- Output: whether $t$ appears in $A$.

**binary-search($A$, $n$, $t$)**

1. $i \leftarrow 1$, $j \leftarrow n$
2. while $i \leq j$ do
3.     $k \leftarrow \lfloor (i + j)/2 \rfloor$
4.     if $A[k] = t$ return true
5.     if $A[k] < t$ then $j \leftarrow k - 1$ else $i \leftarrow k + 1$
6. return false

Running time = $O(\log n)$
Comparing the Orders

- Sort the functions from smallest to largest asymptotically
  \( \log n, \ n, \ n^2, \ n \log n, \ n!, \ 2^n, \ e^n, \ n^n \)
- \( \log n = O(n) \)
Comparing the Orders

- Sort the functions from smallest to largest asymptotically
  \( \log n, \ n, \ n^2, \ n \log n, \ n!, \ 2^n, \ e^n, \ n^n \)
- \( \log n = O(n) \)
- \( n = O(n^2) \)
Comparing the Orders

- Sort the functions from smallest to largest asymptotically
  \( \log n, \ n, \ n^2, \ n \log n, \ n!, \ 2^n, \ e^n, \ n^n \)
- \( \log n = O(n) \)
- \( n = O(n \log n) \)
- \( n \log n = O(n^2) \)
Comparing the Orders

- Sort the functions from smallest to largest asymptotically
  \(\log n, \ n, \ n^2, \ n \log n, \ n!, \ 2^n, \ e^n, \ n^n\)
- \(\log n = O(n)\)
- \(n = O(n \log n)\)
- \(n \log n = O(n^2)\)
- \(n^2 = O(n!)\)
Comparing the Orders

- Sort the functions from smallest to largest asymptotically:
  \( \log n, \ n, \ n^2, \ n \log n, \ n!, \ 2^n, \ e^n, \ n^n \)

- \( \log n = O(n) \)
- \( n = O(n \log n) \)
- \( n \log n = O(n^2) \)
- \( n^2 = O(2^n) \)
- \( 2^n = O(n!) \)
Comparing the Orders

- Sort the functions from smallest to largest asymptotically:
  \( \log n, \ n, \ n^2, \ n \log n, \ n!, \ 2^n, \ e^n, \ n^n \)

- \( \log n = O(n) \)
- \( n = O(n \log n) \)
- \( n \log n = O(n^2) \)
- \( n^2 = O(2^n) \)
- \( 2^n = O(e^n) \)
- \( e^n = O(n!) \)
Comparing the Orders

- Sort the functions from smallest to largest asymptotically
  \text{log}\, n,\  n,\  n^2,\  n\, \text{log}\, n,\  n!,\  2^n,\  e^n,\  n^n

- \log n = O(n)
- n = O(n \log n)
- n \log n = O(n^2)
- n^2 = O(2^n)
- 2^n = O(e^n)
- e^n = O(n!)
- n! = O(n^n)
When we talk about upper bound on running time:

- Logarithmic time: $O(\log n)$
- Linear time: $O(n)$
- Quadratic time $O(n^2)$
- Cubic time $O(n^3)$
- Polynomial time: $O(n^k)$ for some constant $k$
- Exponential time: $O(c^n)$ for some $c > 1$
- Sub-linear time: $o(n)$
- Sub-quadratic time: $o(n^2)$
Goal of Algorithm Design

- Design algorithms to minimize the order of the running time.
Goal of Algorithm Design

- Design algorithms to minimize the order of the running time.

- Using asymptotic analysis allows us to ignore the leading constants and lower order terms.
Goal of Algorithm Design

- Design algorithms to minimize the order of the running time.

- Using asymptotic analysis allows us to ignore the leading constants and lower order terms.

- Makes our life much easier! (E.g., the leading constant depends on the implementation, compiler and computer architecture of computer.)
Q: Does ignoring the leading constant cause any issues?

- e.g. how can we compare an algorithm with running time $0.1n^2$ with an algorithm with running time $1000n$?
Q: Does ignoring the leading constant cause any issues?

- e.g., how can we compare an algorithm with running time $0.1n^2$
  with an algorithm with running time $1000n$?

A:
Q: Does ignoring the leading constant cause any issues?

- e.g., how can we compare an algorithm with running time $0.1n^2$ with an algorithm with running time $1000n$?

A:

- Sometimes yes
Q: Does ignoring the leading constant cause any issues?

- e.g., how can we compare an algorithm with running time $0.1n^2$ with an algorithm with running time $1000n$?

A:

- Sometimes yes
- However, when $n$ is big enough, $1000n < 0.1n^2$
Q: Does ignoring the leading constant cause any issues?

- e.g., how can we compare an algorithm with running time \(0.1n^2\) with an algorithm with running time \(1000n\)?

A:

- Sometimes yes
- However, when \(n\) is big enough, \(1000n < 0.1n^2\)
- For “natural” algorithms, constants are not so big!
Q: Does ignoring the leading constant cause any issues?

- e.g., how can we compare an algorithm with running time $0.1n^2$ with an algorithm with running time $1000n$?

A:

- Sometimes yes
- However, when $n$ is big enough, $1000n < 0.1n^2$
- For “natural” algorithms, constants are not so big!
- So, for reasonably large $n$, algorithm with lower order running time beats algorithm with higher order running time.