CSE 431/531: Analysis of Algorithms

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
Course webpage:
http://www.cse.buffalo.edu/~shil/courses/CSE531/

Please sign up the course on Piazza:
http://piazza.com/buffalo/fall2016/cse431531
CSE 431/531: Analysis of Algorithms

- **Time and location:**
  - MoWeFr, 9:00-9:50am
  - Cooke 121

- **Lecturer:**
  - Shi Li, shil@buffalo.edu
  - Office hours: TBD

- **TAs**
  - Di Wang, dwang45@buffalo.edu
  - Minwei Ye, minweiye@buffalo.edu
  - Alexander Stachnik, ajstachn@buffalo.edu
You should know:

- Mathematical tools
- Mathematical inductions
- Probabilities and random variables
- Data structures
  - Stacks, queues, linked lists
- Some programming experience
  - E.g., C, C++ or Java

You may know:

- Asymptotic analysis
- Simple algorithm design techniques such as greedy, divide-and-conquer, dynamic programming
You should know:

- **Mathematical Tools**
  - Mathematical inductions
  - Probabilities and random variables

- **Data Structures**
  - Stacks, queues, linked lists

- **Some Programming Experience**
  - E.g., C, C++ or Java

- **Asymptotic analysis**
- Simple algorithm design techniques such as greedy, divide-and-conquer, dynamic programming
You should know:

- **Mathematical Tools**
  - Mathematical inductions
  - Probabilities and random variables

- **Data Structures**
  - Stacks, queues, linked lists
You should know:

- **Mathematical Tools**
  - Mathematical inductions
  - Probabilities and random variables

- **Data Structures**
  - Stacks, queues, linked lists

- **Some Programming Experience**
  - E.g., C, C++ or Java
You should know:
- Mathematical Tools
  - Mathematical inductions
  - Probabilities and random variables
- Data Structures
  - Stacks, queues, linked lists
- Some Programming Experience
  - E.g., C, C++ or Java

You may know:
You should know:
- Mathematical Tools
  - Mathematical inductions
  - Probabilities and random variables
- Data Structures
  - Stacks, queues, linked lists
- Some Programming Experience
  - E.g., C, C++ or Java

You may know:
- Asymptotic analysis
You should know:
- Mathematical Tools
  - Mathematical inductions
  - Probabilities and random variables
- Data Structures
  - Stacks, queues, linked lists
- Some Programming Experience
  - E.g., C, C++ or Java

You may know:
- Asymptotic analysis
- Simple algorithm design techniques such as greedy, divide-and-conquer, dynamic programming
You Will Learn

- Classic algorithms for classic problems
  - Sorting
  - Shortest paths
  - Minimum spanning tree
  - Network flow

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

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

- NP-completeness
You Will Learn

- Classic algorithms for classic problems
  - Sorting
  - Shortest paths
  - Minimum spanning tree
  - Network flow

- How to analyze algorithms
  - Correctness
  - Running time (efficiency)
  - Space requirement
You Will Learn

- Classic algorithms for classic problems
  - Sorting
  - Shortest paths
  - Minimum spanning tree
  - Network flow

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

- Meta techniques to design algorithms
  - Greedy algorithms
  - Divide and conquer
  - Dynamic programming
  - Reductions
You Will Learn

- Classic algorithms for classic problems
  - Sorting
  - Shortest paths
  - Minimum spanning tree
  - Network flow

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

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

- NP-completeness
Textbook

Required Textbook:

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

Other Reference Books

Grading

- **20% for homeworks**
  - 5 homeworks, each worth 4%
- **20% for projects**
  - 2 projects, each worth 10%
- **30% for in-class exams**
  - 2 in-class exams, each worth 15%
- **30% for final exam**
  - If to your advantage: each in-class exam is worth 5% and final is worth 50%
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 discussing
  - 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 courses
- Copy solutions from other students
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 courses
- Copy solutions from other students

If you are not following the rules, you will get an “F” for the course.
Projects

- Need to implement an algorithm for each of the two projects
- Cannot copy codes from others or the Internet
Projects

- Need to implement an algorithm for each of the two projects
- Can not copy codes from others or the Internet

If you are not following the rules, you will get an “F” for the course.
Late policy

- You have one late credit
- Turn in a homework or a project late for three days using the late credit
- No other late submissions will be accepted
Exams

- Closed-book
- Can bring one A4 handwritten sheet

If you are caught cheating in exams, you will get an "F" for the course.
Exams

- Closed-book
- Can bring one A4 handwritten sheet

If you are caught cheating in exams, you will get an “F” for the course.
Exams

- Closed-book
- Can bring one A4 handwritten sheet

If you are caught cheating in exams, you will get an “F” for the course.

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

<table>
<thead>
<tr>
<th>Greatest Common Divisor</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong> two integers $a, b &gt; 0$</td>
</tr>
<tr>
<td><strong>Output:</strong> the greatest common divisor of $a$ and $b$</td>
</tr>
</tbody>
</table>
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

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

$$\rightarrow (210, 60) \rightarrow (60, 30) \rightarrow (30, 0)$$
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
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

\[
gcd(270, 210) = gcd(210, 270 \mod 210) = gcd(210, 60)
\]

\[
(270, 210) \rightarrow (210, 60) \rightarrow (60, 30) \rightarrow (30, 0)
\]
### Sort

**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, \ldots
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
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, ...
Examples

Shortest Path

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

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

Shortest Path

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

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

Algorithm: Dijkstra's algorithm
Examples

Shortest Path

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

**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
Algorithm = Computer Program?

- Algorithm: “abstract”, can be specified using computer program, English, pseudo-codes or flow charts.
- Computer program: “concrete”, implementation of algorithm, associated with a particular programming language
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)
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
  - readability
  - extensibility
  - user-friendliness
  . . .
Theoretical Analysis of Algorithms

- Main focus: correctness, running time (efficiency)
- Sometimes: memory usage
- Not covered in the course: engineering side
  - readability
  - extensibility
  - user-friendliness
  - ...

- 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
- readability
- extensibility
- user-friendliness
- ...

Why is it important to study the running time (efficiency) of an algorithm?
1 feasible vs. infeasible
Main focus: correctness, running time (efficiency)
Sometimes: memory usage
Not covered in the course: engineering side
  - readability
  - extensibility
  - user-friendliness
  - ...

Why is it important to study the running time (efficiency) of an algorithm?
1. feasible vs. infeasible
2. use efficiency to pay for user-friendliness, extensibility, etc.
Main focus: correctness, running time (efficiency)

Sometimes: memory usage

Not covered in the course: engineering side
  - readability
  - extensibility
  - user-friendliness
  - . . .

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

1. feasible vs. infeasible
2. use efficiency to pay for user-friendliness, extensibility, etc.
3. fundamental
Theoretical Analysis of Algorithms

- Main focus: correctness, running time (efficiency)
- Sometimes: memory usage
- Not covered in the course: engineering side
  - readability
  - extensibility
  - user-friendliness
  - ...

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

1. feasible vs. infeasible
2. use efficiency to pay for user-friendliness, extensibility, etc.
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, make the first $j$ numbers 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.   \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} $A[i + 1] \leftarrow key$
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. \( A[i+1] \leftarrow key \)

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

\[ \begin{array}{ccccccccc}
12 & 21 & 35 & 53 & 59 & 15 & \uparrow \\
\end{array} \]
Example:
- **Input:** 53, 12, 35, 21, 59, 15
- **Output:** 12, 15, 21, 35, 53, 59

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

1. for \( j \leftarrow 2 \) to \( n \)
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. \( A[i + 1] \leftarrow key \)

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

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

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

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

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

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

12 21 35 53 59 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. \( A[i + 1] \leftarrow key \)

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

12 21 35 53 53 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 \(n\)
2. \(\quad key \leftarrow A[j]\)
3. \(\quad i \leftarrow j - 1\)
4. while \(i > 0\) and \(A[i] > key\)
5. \(\quad A[i + 1] \leftarrow A[i]\)
6. \(\quad i \leftarrow i - 1\)
7. \(\quad A[i + 1] \leftarrow key\)

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

\(12 \quad 21 \quad 35 \quad 53 \quad 53 \quad 59\)

\(↑\)

\(i\)
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. \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 35 35 53 59

\[ i \]

\[ \uparrow \]
Example:

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

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

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

- $j = 6$
- key = 15

12 21 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* ← 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*

- *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 \( j \leftarrow 2 \) to \( n \)
2. \( key \leftarrow A[j] \)
3. \( i \leftarrow j - 1 \)
4. while \( i > 0 \) and \( A[i] > key \)
   - \( A[i + 1] \leftarrow A[i] \)
   - \( i \leftarrow i - 1 \)
5. \( A[i + 1] \leftarrow key \)

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

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

after $j = 1$ : 53, 12, 35, 21, 59, 15
after $j = 2$ : 12, 53, 35, 21, 59, 15
after $j = 3$ : 12, 35, 53, 21, 59, 15
after $j = 4$ : 12, 21, 35, 53, 59, 15
after $j = 5$ : 12, 21, 35, 53, 59, 15
after $j = 6$ : 12, 15, 21, 35, 53, 59
Q: Size of input?
Analyze Running Time of Insertion Sort

- Q: Size of input?
- A: Running time as function of size
Q: Size of input?
A: Running time as function of size

possible definition of size: # integers, total length of integers, # vertices in graph, # edges in graph
Q: Size of input?
A: Running time as function of size
possible definition of size: # integers, total length of integers, # vertices in graph, # edges in graph
Q: Which input?
Q: Size of input?
A: Running time as function of size
possible definition of size: # integers, total length of integers, # vertices in graph, # edges in graph

Q: Which input?
A: Worst-case analysis:
  - Worst running time over all input instances of a given size
Q: Size of input?
A: Running time as function of size
possible definition of size: # integers, total length of integers, # vertices in graph, # edges in graph

Q: Which input?
A: Worst-case analysis:
  Worst running time over all input instances of a given size

Q: How fast is the computer?
Analyze Running Time of Insertion Sort

- Q: Size of input?
- A: Running time as function of size
- possible definition of size: # integers, total length of integers, # vertices in graph, # edges in graph

- Q: Which input?
- A: Worst-case analysis:
  - Worst running time over all input instances of a given size

- Q: How fast is the computer?
- Q: Programming language?
Q: Size of input?
A: Running time as function of size
possible definition of size: # integers, total length of integers, # vertices in graph, # edges in graph

Q: Which input?
A: Worst-case analysis:
- Worst running time over all input instances of a given size

Q: How fast is the computer?
Q: Programming language?
A: Important idea: asymptotic analysis
- Focus on growth of running-time as a function, not any particular value.
Asymptotic Analysis: $O$-notation

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

- Ignoring lower order terms
- Ignoring leading constant

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

- 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

- Ignoring lower order terms
- Ignoring leading constant

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

- Ignoring lower order terms
- Ignoring leading constant

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

- Ignoring lower order terms
- Ignoring leading constant

$O$-notation allows us to

- ignore architecture of computer
- ignore programming language
Asymptotic Analysis of Insertion Sort

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
6.       i ← i − 1
7.  A[i + 1] ← key

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

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

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

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

\[
\text{insertion-sort}(A, n)
\]

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

Worst-case running time for iteration \( j \) in the outer loop?
Answer: \( O(j) \)
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 \( j \) in the outer loop?
  Answer: \( O(j) \)
- Total running time = \( \sum_{j=2}^{n} O(j) = O(n^2) \) (informal)

Basic operations take $O(1)$ time: addition, subtraction, multiplication, etc.

Each integer (word) has $c \log n$ bits, $c \geq 1$ large enough.

Basic operations take $O(1)$ time: addition, subtraction, multiplication, etc.

Each integer (word) has $c \log n$ bits, $c \geq 1$ large enough.

Precision of real numbers?

Basic operations take $O(1)$ time: addition, subtraction, multiplication, etc.

Each integer (word) has $c \log n$ bits, $c \geq 1$ large enough

Precision of real numbers?
In most scenarios in the course, assuming real numbers are represented exactly

Basic operations take $O(1)$ time: addition, subtraction, multiplication, etc.

Each integer (word) has $c \log n$ bits, $c \geq 1$ large enough.

Precision of real numbers?
In most scenarios in the course, assuming real numbers are represented exactly.

Can we do better than insertion sort asymptotically?

Basic operations take $O(1)$ time: addition, subtraction, multiplication, etc.

Each integer (word) has $c \log n$ bits, $c \geq 1$ large enough

Precision of real numbers?
In most scenarios in the course, assuming real numbers are represented exactly

Can we do better than insertion sort asymptotically?
Yes: merge sort, quicksort, heap sort, ...
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
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 $f(n) > 0$
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 $n$.

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

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

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

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

- $n^2 - n - 30$  Yes
- $2^n - n^{20}$  Yes
Def. \( f : \mathbb{N} \to \mathbb{R} \) is an asymptotically positive function if:

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

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

- \( n^2 - n - 30 \) Yes
- \( 2^n - n^{20} \) Yes
- \( 100n - n^2/10 + 50 \) ? We only consider 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

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 \( n \).

- \( n^2 - n - 30 \) \hspace{1cm} Yes
- \( 2^n - n^{20} \) \hspace{1cm} Yes
- \( 100n - n^2/10 + 50? \) \hspace{1cm} 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**  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 $n$. 
**$O$-Notation: Asymptotic Upper Bound**

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

$$\forall c > 0, n_0 > 0 \text{ such that } f(n) \leq cg(n), \forall n \geq n_0 \text{.}$$

- In other words, $f(n) \in O(g(n))$ if $f(n) \leq cg(n)$ for some $c$ and large enough $n$.
- Informally, think of it as "$f \leq g$".
\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 \(n\).
- Informally, think of it as "\(f \leq g\)".
- \(3n^2 + 2n \in O(n^3)\)
\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 \( n \).

- Informally, think of it as \( f \leq g \).

- \( 3n^2 + 2n \in O(n^3) \)
- \( 3n^2 + 2n \in O(n^2) \)
**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 \( n \).
- Informally, think of it as “\( f \leq g \)”. 

\[
\begin{align*}
 3n^2 + 2n & \in O(n^3) \\
 3n^2 + 2n & \in O(n^2) \\
n^{100} & \in O(2^n)
\end{align*}
\]
**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 $n$.
- Informally, think of it as “$f \leq g$”.

- $3n^2 + 2n \in O(n^3)$
- $3n^2 + 2n \in O(n^2)$
- $n^{100} \in O(2^n)$
- $n^3 \notin O(n^2)$
We use \( f(n) = O(g(n)) \) to denote \( f(n) \in O(g(n)) \)
We use “$f(n) = O(g(n))$” to denote “$f(n) \in O(g(n))$”

$3n^2 + 2n = O(n^3)$
Conventions

- We use \( f(n) = O(g(n)) \) to denote \( f(n) \in O(g(n)) \)
- \( 3n^2 + 2n = O(n^3) \)
- \( 4n^3 + 3n^2 + 2n = 4n^3 + O(n^3) \)
We use “\( f(n) = O(g(n)) \)” to denote “\( f(n) \in O(g(n)) \)”

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

\[ 4n^3 + 3n^2 + 2n = 4n^3 + O(n^3) \]

There exists a function \( f(n) \in O(n^3) \), such that

\[ 4n^3 + 3n^2 + 2n = 4n^3 + f(n). \]
Conventions

- We use “\( f(n) = O(g(n)) \)” to denote “\( f(n) \in O(g(n)) \)”
- \( 3n^2 + 2n = O(n^3) \)
- \( 4n^3 + 3n^2 + 2n = 4n^3 + O(n^3) \)
  - There exists a function \( f(n) \in O(n^3) \), such that \( 4n^3 + 3n^2 + 2n = 4n^3 + f(n) \).
- \( n^2 + O(n) = O(n^2) \)
We use \( f(n) = O(g(n)) \) to denote \( f(n) \in O(g(n)) \)

- \( 3n^2 + 2n = O(n^3) \)
- \( 4n^3 + 3n^2 + 2n = 4n^3 + O(n^3) \)
  - There exists a function \( f(n) \in O(n^3) \), such that \( 4n^3 + 3n^2 + 2n = 4n^3 + f(n) \).

- \( n^2 + O(n) = O(n^2) \)
  - For every function \( f(n) \in O(n) \), there exists a function \( g(n) \in O(n^2) \), such that \( n^2 + f(n) = g(n) \).
We use “\(f(n) = O(g(n))\)” to denote “\(f(n) \in O(g(n))\)”

3\(n^2 + 2n = O(n^3)\)

4\(n^3 + 3n^2 + 2n = 4n^3 + O(n^3)\)

There exists a function \(f(n) \in O(n^3)\), such that
\(4n^3 + 3n^2 + 2n = 4n^3 + f(n)\).

\(n^2 + O(n) = O(n^2)\)

For every function \(f(n) \in O(n)\), there exists a function
\(g(n) \in O(n^2)\), such that \(n^2 + f(n) = g(n)\).

Rule: left side \(\rightarrow \forall\), right side \(\rightarrow \exists\)
Conventions

- $3n^2 + 2n = O(n^3)$
- $4n^3 + 3n^2 + 2n = 4n^3 + O(n^3)$
- $n^2 + O(n) = O(n^2)$

“=” is asymmetric! Following statements are wrong:
- $O(n^3) = 3n^2 + 2n$
- $4n^3 + O(n^3) = 4n^3 + 3n^2 + 2n$
- $O(n^2) = n^2 + O(n)$
Conventions

- $3n^2 + 2n = O(n^3)$
- $4n^3 + 3n^2 + 2n = 4n^3 + O(n^3)$
- $n^2 + O(n) = O(n^2)$

“=” is asymmetric! Following statements are wrong:

- $O(n^3) = 3n^2 + 2n$
- $4n^3 + O(n^3) = 4n^3 + 3n^2 + 2n$
- $O(n^2) = n^2 + O(n)$

Chaining is allowed:

$4n^3 + 3n^2 + 2n = 4n^3 + O(n^3) = O(n^3) = O(n^4)$
**Ω-Notation: Asymptotic Lower Bound**

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

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

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

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

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

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

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

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

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


**Ω-Notation: Asymptotic Lower Bound**

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

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

- In other words, \( f(n) \in Ω(g(n)) \) if \( f(n) \geq cg(n) \) for some \( c \) and large enough \( n \).
- Informally, think of it as “\( f \geq g \)”. 
Again, we use “=” instead of $\in$.

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

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

Theorem \[ f(n) = O(g(n)) \iff g(n) = \Omega(f(n)). \]
**Θ-Notation**  For a function $g(n)$,

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

Informally, think of it as $f(n) = g(n)$.
\(\Theta\)-Notation: Asymptotic Tight Bound

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

- \(f(n) = \Theta(g(n))\), then for large enough \(n\), we have “\(f(n) \approx g(n)\)”. 

Informally, think of it as “\(f(n) = g(n)\)”.

\(2n^2 + 2n = \Theta(n^2)\)

\(2n/3 + 100 = \Theta(2n/3)\)

Theorem \(f(n) = \Theta(g(n))\) if and only if \(f(n) = O(g(n))\) and \(f(n) = \Omega(g(n))\).
\(\Theta\)-Notation: Asymptotic Tight Bound

\(\Theta\)-Notation  For a function \(g(n)\),
\[
\Theta(g(n)) = \{ \text{function } f : \exists c_2 \geq c_1 > 0, n_0 > 0 \text{ such that } c_1g(n) \leq f(n) \leq c_2g(n), \forall n \geq n_0 \}.
\]

- \(f(n) = \Theta(g(n))\), then for large enough \(n\), we have “\(f(n) \approx g(n)\)”.
- Informally, think of it as “\(f = g\)”. 
\textbf{Θ-Notation} For a function $g(n)$,
\[ Θ(g(n)) = \{ \text{function } f : \exists c_2 \geq c_1 > 0, n_0 > 0 \text{ such that } c_1 g(n) \leq f(n) \leq c_2 g(n), \forall n \geq n_0 \}. \]

- $f(n) = Θ(g(n))$, then for large enough $n$, we have “$f(n) \approx g(n)$”.
- Informally, think of it as “$f = g$”.
- $3n^2 + 2n = Θ(n^2)$
**Θ-Notation: Asymptotic Tight Bound**

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

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

- \( f(n) = Θ(g(n)) \), then for large enough \( n \), we have “\( f(n) \approx g(n) \)”.
- Informally, think of it as “\( f = g \)”.

- \( 3n^2 + 2n = Θ(n^2) \)
- \( 2^{n/3} + 100 = Θ(2^{n/3}) \)
**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 “$f(n) \approx g(n)$”.
- Informally, think of it as “$f = g$”.

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

**Theorem**  $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$. 
### 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>$\log_{10} n$</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>
<tr>
<td>$n^2 - 100n$</td>
<td>$5n^2 + 30n$</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>$\lg^{10} n$</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>
<tr>
<td>$n^2 - 100n$</td>
<td>$5n^2 + 30n$</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>$\lg^{10} n$</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>
<tr>
<td>$n^2 - 100n$</td>
<td>$5n^2 + 30n$</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>( \lg^{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>
<tr>
<td>( n^2 - 100n )</td>
<td>( 5n^2 + 30n )</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</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>$\lg^{10} n$</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>
<tr>
<td>$n^2 - 100n$</td>
<td>$5n^2 + 30n$</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Asymptotic Notations</td>
<td>$O$</td>
<td>$\Omega$</td>
<td>$\Theta$</td>
<td></td>
</tr>
<tr>
<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></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 $f(n) = \Omega(g(n))$

**Incorrect Analogy**

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

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

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

**Correct Analogies**

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

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

### 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 \( g(n) = O(f(n)) \)

\[
\begin{align*}
  f(n) &= n^2 \\
  g(n) &= \begin{cases} 
    1 & \text{if } n \text{ is odd} \\
    2^n & \text{if } n \text{ is even}
  \end{cases}
\end{align*}
\]
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: $3n^2 \rightarrow n^2$
- Thus $3n^2 - 10n - 5 = O(n^2)$
Recall: informal way to define $O$-notation

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

Formally: if $n > 10$, then $n^2 < 3n^2 - 10n - 5 < 3n^2$. So, $3n^2 - 10n - 5 \in \Theta(n^2)$. 
\( o \) and \( \omega \)-Notations

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

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

Example:
- \( 3n^2 + 5n + 10 = o(n^2 \log n) \).
- \( 3n^2 + 5n + 10 = \omega(n^2 / \log n) \).
<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
Computing the sum of $n$ numbers

```plaintext
sum(A, n)

1. $S \leftarrow 0$
2. for $i \leftarrow 1$ to $n$
3. $S \leftarrow S + A[i]$
4. return $S$
```
Merge two sorted arrays

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>8</td>
<td>12</td>
<td>20</td>
<td>32</td>
<td>48</td>
</tr>
</tbody>
</table>

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>7</td>
<td>9</td>
<td>25</td>
<td>29</td>
<td></td>
</tr>
</tbody>
</table>
Merge two sorted arrays
Merge two sorted arrays

\[
\begin{array}{cccccc}
3 & 8 & 12 & 20 & 32 & 48 \\
5 & 7 & 9 & 25 & 29 \\
3 & \\
\end{array}
\]
Merge two sorted arrays

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

\[
\begin{array}{cccccc}
3 & 8 & 12 & 20 & 32 & 48 \\
5 & 7 & 9 & 25 & 29 \\
3 & 5 \\
\end{array}
\]
Merge two sorted arrays

\[
\begin{array}{cccccc}
3 & 8 & 12 & 20 & 32 & 48 \\
5 & 7 & 9 & 25 & 29 \\
3 & 5 \\
\end{array}
\]
Merge two sorted arrays

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

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

\[ 3 \quad 8 \quad 12 \quad 20 \quad 32 \quad 48 \]

\[ 5 \quad 7 \quad 9 \quad 25 \quad 29 \]

\[ 3 \quad 5 \quad 7 \quad 8 \]
\(O(n)\) (Linear) Running Time

- Merge two sorted arrays

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

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

\[
\begin{array}{cccc}
3 & 5 & 7 & 8 \\
\end{array}
\]
Merge two sorted arrays

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

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

\[
\begin{array}{cccccccc}
3 & 5 & 7 & 8 & 9 & 12 & 20 & 25 & 29 \\
\end{array}
\]
Merge two sorted arrays

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

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

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 $A$

Running time = $O(n)$ where $n = n_1 + n_2$. 
**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 \(A\)

Running time = \(O(n)\) where \(n = n_1 + n_2\).
merge-sort\((A, n)\)

1. if \( n = 1 \) then
2. return \( A \)
3. else
4. \( B \leftarrow \text{merge-sort}\left( A[1..\lfloor n/2 \rfloor], \lfloor n/2 \rfloor \right) \)
5. \( C \leftarrow \text{merge-sort}\left( A[\lfloor n/2 \rfloor + 1..n], n - \lfloor n/2 \rfloor \right) \)
6. return merge\((B, C, \lfloor n/2 \rfloor, n - \lfloor n/2 \rfloor)\)
Merge-Sort

Each level takes running time $O(n)$.

There are $O(\log n)$ levels.

Running time = $O(n \log n)$.
\(O(n \lg n)\) Running Time

- **Merge-Sort**

Each level takes running time \(O(n)\)
**$O(n \lg n)$ Running Time**

- **Merge-Sort**

![Diagram](image)

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

- Merge-Sort

Each level takes running time $O(n)$
There are $O(\lg n)$ levels
Running time $= O(n \lg n)$
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: \((x_1, y_1), (x_2, y_2), \cdots, (x_n, y_n)\)

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

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

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

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

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

Closest pair can be solved in $O(n^2)$ time!
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(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. return $(besti, bestj)$
```

Closest pair can be solved in $O(n \lg n)$ 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 $(u, v) \notin E$. 
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 $(u, v) \notin E$.

Independent set of size $k$

Input: graph $G = (V, E)$, an integer $k$
Output: whether there is an independent set of size $k$
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)
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{2em} for every $u, v \in S$
5. \hspace{3em} if $(u, v) \in E$ then $b \leftarrow \text{false}$
6. \hspace{2em} if $b$ and $|S| > |R|$ then $R \leftarrow S$
7. return $R$

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

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

Running time = $O(n! \times n)$
$O(\lg n)$ (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:
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:
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
```
**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
```
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:
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:
$O(\lg n)$ (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:
**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:

$$
\begin{array}{cccccccccccccccc}
3 & 8 & 10 & 25 & 29 & 37 & 38 & 42 & 46 & 52 & 59 & 61 & 63 & 75 & 79 \\
\end{array}
$$

25 < 35
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:

<table>
<thead>
<tr>
<th>3</th>
<th>8</th>
<th>10</th>
<th>25</th>
<th>29</th>
<th>37</th>
<th>38</th>
<th>42</th>
<th>46</th>
<th>52</th>
<th>59</th>
<th>61</th>
<th>63</th>
<th>75</th>
<th>79</th>
</tr>
</thead>
</table>

$O(\lg n)$ (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:**

<table>
<thead>
<tr>
<th>3</th>
<th>8</th>
<th>10</th>
<th>25</th>
<th>29</th>
<th>37</th>
<th>38</th>
<th>42</th>
<th>46</th>
<th>52</th>
<th>59</th>
<th>61</th>
<th>63</th>
<th>75</th>
<th>79</th>
</tr>
</thead>
</table>
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:
**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:

<table>
<thead>
<tr>
<th>3</th>
<th>8</th>
<th>10</th>
<th>25</th>
<th>29</th>
<th>37</th>
<th>38</th>
<th>42</th>
<th>46</th>
<th>52</th>
<th>59</th>
<th>61</th>
<th>63</th>
<th>75</th>
<th>79</th>
</tr>
</thead>
</table>

$O(\lg n)$ (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 $i \leftarrow k + 1$
3. return false
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 $= O(lg n)$
Compare the Orders

- Sort the functions from asymptotically smallest to asymptotically largest (informally, using “<” and “=”)

\[ n^{\sqrt{n}}, \ lg\ n, \ n, \ n^2, \ n\ lg\ n, \ n!, \ 2^n, \ e^n, \ lg(n!), \ n^n \]
Compare the Orders

- Sort the functions from asymptotically smallest to asymptotically largest (informally, using “<” and “=” )
  
  \( n^{\sqrt{n}}, \ lg\ n, \ n, \ n^2, \ n \ lg\ n, \ n!, \ 2^n, \ e^n, \ lg(n!), \ n^n \)

- \( lg\ n < n^{\sqrt{n}} \)
Sort the functions from asymptotically smallest to asymptotically largest (informally, using “<” and “=”)

\[ n^{\sqrt{n}}, \ lg\ n, \ n, \ n^2, \ n\ lg\ n, \ n!, \ 2^n, \ e^n, \ lg(n!), \ n^n \]

\[ lg\ n < n^{\sqrt{n}} \]

\[ lg\ n < n < n^{\sqrt{n}} \]
Compare the Orders

- Sort the functions from asymptotically smallest to asymptotically largest (informally, using “<” and “=”):
  \( n^{\sqrt{n}}, \ lg\ n, \ n, \ n^2, \ n \ lg\ n, \ n!, \ 2^n, \ e^n, \ lg(n!), \ n^n \)
- \( \lg n < n^{\sqrt{n}} \)
- \( \lg n < n < n^{\sqrt{n}} \)
- \( \lg n < n < n^2 < n^{\sqrt{n}} \)
Compare the Orders

- Sort the functions from asymptotically smallest to asymptotically largest (informally, using “<” and “=”)

  \[ n^{\sqrt{n}}, \ lg\ n, \ n, \ n^2, \ n\ lg\ n, \ n!, \ 2^n, \ e^n, \ lg(n!), \ n^n \]

- \[ \lg\ n < n^{\sqrt{n}} \]
- \[ \lg\ n < n < n^{\sqrt{n}} \]
- \[ \lg\ n < n < n^2 < n^{\sqrt{n}} \]
- \[ \lg\ n < n < n \ lg\ n < n^2 < n^{\sqrt{n}} \]
Compare the Orders

Sort the functions from asymptotically smallest to asymptotically largest (informally, using “<” and “=”):

\( n^{\sqrt{n}}, \ \lg n, \ n, \ n^2, \ n \lg n, \ n!, \ 2^n, \ e^n, \ \lg(n!), \ n^n \)

- \( \lg n < n^{\sqrt{n}} \)
- \( \lg n < n < n^{\sqrt{n}} \)
- \( \lg n < n < n^2 < n^{\sqrt{n}} \)
- \( \lg n < n < n \lg n < n^2 < n^{\sqrt{n}} \)
- \( \lg n < n < n \lg n < n^2 < n^{\sqrt{n}} < n! \)
Compare the Orders

- Sort the functions from asymptotically smallest to asymptotically largest (informally, using “<” and “=”)
  \[ n^{\sqrt{n}}, \ lg\ n, \ n, \ n^2, \ n\ lg\ n, \ n!, \ 2^n, \ e^n, \ lg(n!), \ n^n \]
- \[ lg\ n < n^{\sqrt{n}} \]
- \[ lg\ n < n < n^{\sqrt{n}} \]
- \[ lg\ n < n < n^2 < n^{\sqrt{n}} \]
- \[ lg\ n < n < n \ lg\ n < n^2 < n^{\sqrt{n}} \]
- \[ lg\ n < n < n \ lg\ n < n^2 < n^{\sqrt{n}} < n! \]
- \[ lg\ n < n < n \ lg\ n < n^2 < n^{\sqrt{n}} < 2^n < n! \]
Compare the Orders

- Sort the functions from asymptotically smallest to asymptotically largest (informally, using “<” and “=”)
  \[ n^{\sqrt{n}}, \lg n, n, n^2, n \lg n, n!, 2^n, e^n, \lg(n!), n^n \]
- \[ \lg n < n^{\sqrt{n}} \]
- \[ \lg n < n < n^{\sqrt{n}} \]
- \[ \lg n < n < n^2 < n^{\sqrt{n}} \]
- \[ \lg n < n < n \lg n < n^2 < n^{\sqrt{n}} \]
- \[ \lg n < n < n \lg n < n^2 < n^{\sqrt{n}} < n! \]
- \[ \lg n < n < n \lg n < n^2 < n^{\sqrt{n}} < 2^n < n! \]
- \[ \lg n < n < n \lg n < n^2 < n^{\sqrt{n}} < 2^n < e^n < n! \]
Compare the Orders

- Sort the functions from asymptotically smallest to asymptotically largest (informally, using "<" and "\=")

\( n\sqrt{n}, \ lg\ n, \ n, \ n^2, \ n\ lg\ n, \ n!, \ 2^n, \ e^n, \ lg(n!), \ n^n \)

- \( \lg\ n < n\sqrt{n} \)
- \( \lg\ n < n < n\sqrt{n} \)
- \( \lg\ n < n < n^2 < n\sqrt{n} \)
- \( \lg\ n < n < n\ lg\ n < n^2 < n\sqrt{n} \)
- \( \lg\ n < n < n\ lg\ n < n^2 < n\sqrt{n} < n! \)
- \( \lg\ n < n < n\ lg\ n < n^2 < n\sqrt{n} < 2^n < n! \)
- \( \lg\ n < n < n\ lg\ n < n^2 < n\sqrt{n} < 2^n < e^n < n! \)
- \( \lg\ n < n < n\ lg\ n = lg(n!) < n^2 < n\sqrt{n} < 2^n < e^n < n! \)
Compare the Orders

- Sort the functions from asymptotically smallest to asymptotically largest (informally, using “<” and “=”)
  
  \( n^{\sqrt{n}}, \ lg\ n, \ n, \ n^2, \ n\ lg\ n, \ n!, \ 2^n, \ e^n, \ lg(n!), \ n^n \)

- \( lg\ n < n^{\sqrt{n}} \)
- \( lg\ n < n < n^{\sqrt{n}} \)
- \( lg\ n < n < n^2 < n^{\sqrt{n}} \)
- \( lg\ n < n < n \ lg\ n < n^2 < n^{\sqrt{n}} \)
- \( lg\ n < n < n \ lg\ n < n^2 < n^{\sqrt{n}} < n! \)
- \( lg\ n < n < n \ lg\ n < n^2 < n^{\sqrt{n}} < 2^n < n! \)
- \( lg\ n < n < n \ lg\ n < n^2 < n^{\sqrt{n}} < 2^n < e^n < n! \)
- \( lg\ n < n < n \ lg\ n = \ lg(n!) < n^2 < n^{\sqrt{n}} < 2^n < e^n < n! \)
- \( lg\ n < n < n \ lg\ n = \ lg(n!) < n^2 < n^{\sqrt{n}} < 2^n < e^n < n! < n^n \)
When we talk about upper bounds:

- Logarithmic time: $O(\lg n)$
- 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: $o(n^2)$
Terminologies

When we talk about upper bounds:
- 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: $o(n^2)$

When we talk about lower bounds:
- Super-linear time: $\omega(n)$
- Super-quadratic time: $\omega(n^2)$
- Super-polynomial time: $\bigcap_{k>0} \omega(n^k)$
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!
- For reasonable $n$, algorithm with lower order running time beats algorithm with higher order running time.