UMC 201: Data Structures & Algorithms

Lecture 1: Course info; Introduction; Insertion sort; Order notations

Indian Institute of Science

Course Info

- **Course no.:** UMC 201 **Credits:** 3:1
- **Instructors:** Chandan Saha (chandan@iisc)
  Sathish Govindarajan (gsat@iisc)
- **Lecture time:** T,Th, 10-11:20 am **Venue:** G-21
- **Teams group:** Will be set up soon.
- **TAs:** Agrim Dewan (agrimdewan@iisc); Sreesankar T M (sreesankar1@iisc), and 4 others.
- **Course homepage:**
  https://www.csa.iisc.ac.in/~chandan/courses/dsa26/home.html

Course Info

- **Prerequisites**: UENG 101; Mathematical maturity.

- **Primary reference:** Introduction to Algorithms by *Cormen, Leiserson, Rivest and Stein*. (We'll follow this textbook ***closely***.)

- **Other references:** Data Structures and Algorithm Analysis in C by *Weiss*; Algorithm Design by *Kleinberg & Tardos*.

Course Info

- **Number of lectures:** ~25.

- Chandan Saha will teach the *first half* of the course (until the mid-term exam). Sathish Govindarajan will teach the *second half*.
- **Lectures:** Slides of the *first half* of the course will be posted on the course homepage. The *second half* will have board lectures; materials will be posted on Teams as and when required.

Course Info

- **Grading policy**: Two *class tests* - 30% (15% each)
  Mid-term exam - 35%
  End-term exam - 35%

- **Mid-term exam:** Between Sep 19 – Sep 26
- **Final-term exam:** Between Nov 20 – Dec 2

Class Tests

- *Practice problems* will be provided in the slides (in the first half) and over Teams (in the second half).
- In a class test, a student will be asked to solve problems (which will be chosen from among the practice problems or slight variants of them).
- There will be multiple sets of question papers for each of the two class tests.
- **Class test duration:** 80 mins, in the class.
- **First test:** Sep 3 (Th). **Second test:** Nov 3 (T)

Class Tests

- *Practice problems* will be provided in the slides (in the first half) and over Teams (in the second half).
- In a class test, a student will be asked to solve problems (which will be chosen from among the practice problems or slight variants of them).
- There will be multiple sets of question papers for each of the two class tests.
- The syllabus for the tests will be announced on the course homepage, over Teams, and in the class.

Mid-term and End-term exams

- Would be 3 hr long written tests.
- **When?** Exact dates will be announced on the course homepage, over Teams, and in the class.
- The problems given in these exams **need not** be from among the practice problems.

Mid-term and End-term exams

- **Mid-term syllabus:** Everything covered before the mid-term exam week.

- **End-term syllabus:** Everything covered *after the mid-term* and before the end-term exam.

Tutorial sessions?

- If required, we will have doubt-clearing sessions or tutorial sessions. The dates will be announced on the course homepage, over Teams, and in the class.

Introduction

What is an algorithm?

- An algorithm is a well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output in a *finite amount of time*.
- An algorithm is a tool for solving a well-specified (computational) problem.

What is an algorithm?

- An algorithm is a well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output in a *finite amount of time*.
- An algorithm is a tool for solving a well-specified (computational) problem.
- Example. *(Sorting problem)* Sort a sequence of numbers into monotonically increasing order.

$$\underbrace{(31, 41, 59, 26, 41, 58)}_{\text{Input (instance)}} \mapsto \underbrace{(26, 31, 41, 41, 58, 59)}_{\text{Output}}$$

What is an algorithm?

- An algorithm is a well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output in a *finite amount of time*.
- An algorithm is a tool for solving a well-specified (computational) problem.
- An algorithm solves a problem if for *every* input instance, it halts, and outputs the correct solution.

What is an algorithm?

- An algorithm is a well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output in a *finite amount of time*.
- An algorithm is a tool for solving a well-specified (computational) problem.
- An algorithm solves a problem if for *every* input instance, it halts, and outputs the correct solution.
- An algorithm can be specified as a pseudocode or a computer program, or as a hardware design.

What is a pseudocode?

```
SUM-ARRAY(A, n)
1  sum = 0
2  for i = 1 to n
3      sum = sum + A[i]
4  return sum
```

- The above pseudocode (algorithm) takes an array A and a number n as input and returns the sum of the first n values of the array.

Problems and Applications

Problems	Applications
Sorting	To order records in databases
Topological Sorting	To indicate precedence among events

Problems and Applications

Problems	Applications
Sorting	To order records in databases
Topological Sorting	To indicate precedence among events
Matrix Multiplication	AI & ML, Graphics & Image processing

Problems and Applications

Problems	Applications
Sorting	To order records in databases
Topological Sorting	To indicate precedence among events
Matrix Multiplication	AI & ML, Graphics & Image processing
Longest Common Subsequence	Determining similarity between DNA sequences

Problems and Applications

Problems	Applications
Sorting	To order records in databases
Topological Sorting	To indicate precedence among events
Matrix Multiplication	AI & ML, Graphics & Image processing
Longest Common Subsequence	Determining similarity between DNA sequences
Coding	File compression & error correction

Problems and Applications

Problems	Applications
Sorting	To order records in databases
Topological Sorting	To indicate precedence among events
Matrix Multiplication	AI & ML, Graphics & Image processing
Longest Common Subsequence	Determining similarity between DNA sequences
Coding	File compression & error correction
Minimum Spanning Tree	Electronic Circuit Design

Problems and Applications

Problems	Applications
Sorting	To order records in databases
Topological Sorting	To indicate precedence among events
Matrix Multiplication	AI & ML, Graphics & Image processing
Longest Common Subsequence	Determining similarity between DNA sequences
Coding	File compression & error correction
Minimum Spanning Tree	Electronic Circuit Design
Shortest Paths in Graphs	Modelling road maps, routes for data travel

Problems and Applications

Problems	Applications
Sorting	To order records in databases
Topological Sorting	To indicate precedence among events
Matrix Multiplication	AI & ML, Graphics & Image processing
Longest Common Subsequence	Determining similarity between DNA sequences
Coding	File compression & error correction
Minimum Spanning Tree	Electronic Circuit Design
Shortest Paths in Graphs	Modelling road maps, routes for data travel

We'll discuss all these problems in later lectures.

Problems and Applications

Problems	Applications
Linear Programming	Optimizing an objective, given limited resources and competing constraints
Encryption/Decryption	E-commerce

We'll discuss these problems if time permits.

Problems and Applications

Problems	Applications
Linear Programming	Optimizing an objective, given limited resources and competing constraints
Encryption/Decryption	E-commerce

We'll discuss these problems if time permits.

- Algorithms to solve these and many other problems are at the *core of most technologies* used in contemporary computers.

What is a data structure?

- A data structure is a way to store and organize data in order to facilitate access and modifications.

- Using appropriate data structures is an important part of algorithm design.

- Examples of data structures. Arrays, lists, binary trees, hash tables, etc. We will study these in later lectures.

Techniques for Algorithm Design

- We'll also discuss the following broad techniques for designing and analysing algorithms:
  - Divide-and-Conquer
  - Dynamic Programming
  - Greedy Strategies
  - Amortized Analysis

Techniques for Algorithm Design

- We'll also discuss the following broad techniques for designing and analysing algorithms:
  - Divide-and-Conquer
  - Dynamic Programming
  - Greedy Strategies
  - Amortized Analysis
- These techniques are used to design time efficient algorithms. Question: Other than time, what other measure of efficiency can you think of?

Techniques for Algorithm Design

- We'll also discuss the following broad techniques for designing and analysing algorithms:
  - Divide-and-Conquer
  - Dynamic Programming
  - Greedy Strategies
  - Amortized Analysis
- These techniques are used to design time efficient algorithms. Question: Other than time, what other measure of efficiency can you think of? Memory space.

Insertion Sort:
Our first algorithm

Insertion Sort

- Sorting problem. Input is an array of n values $(a_1, \ldots, a_n)$. Output a reordering $(a'_1, \ldots, a'_n)$ of the input array such that $a'_1 \leq \ldots \leq a'_n$.

Insertion Sort

- Sorting problem. Input is an array of n values $(a_1, \ldots, a_n)$. Output a reordering $(a'_1, \ldots, a'_n)$ of the input array such that $a'_1 \leq \ldots \leq a'_n$.

- Insertion sort works the way you might sort a hand of playing cards.
- Idea. Remove one card from the pile and insert it into the correct position.

Insertion Sort: Pseudocode

```
INSERTION-SORT(A, n)
1  for i = 2 to n
2      key = A[i]
3      // Insert A[i] into the sorted subarray A[1 : i − 1].
4      j = i − 1
5      while j > 0 and A[j] > key
6          A[j + 1] = A[j]
7          j = j − 1
8      A[j + 1] = key
```

- Input is an array A of values and the number n of values to sort.

Insertion Sort: Pseudocode

```
INSERTION-SORT(A, n)
1  for i = 2 to n
2      key = A[i]
3      // Insert A[i] into the sorted subarray A[1 : i − 1].
4      j = i − 1
5      while j > 0 and A[j] > key
6          A[j + 1] = A[j]
7          j = j − 1
8      A[j + 1] = key
```

- Analyzing a pseudocode (algorithm) means proving its **correctness** and bounding the resources (which in our case would be **computational time**) that the algorithm requires.

Insertion Sort: Pseudocode

```
INSERTION-SORT(A, n)
1  for i = 2 to n
2      key = A[i]
3      // Insert A[i] into the sorted subarray A[1 : i − 1].
4      j = i − 1
5      while j > 0 and A[j] > key
6          A[j + 1] = A[j]
7          j = j − 1
8      A[j + 1] = key
```

- Correctness. Inside the **for**-loop (*Steps 2-8*) we assume that the subarray $A[1 : i-1]$ is already sorted.

Insertion Sort: Pseudocode

```
INSERTION-SORT(A, n)
1  for i = 2 to n
2      key = A[i]
3      // Insert A[i] into the sorted subarray A[1 : i − 1].
4      j = i − 1
5      while j > 0 and A[j] > key
6          A[j + 1] = A[j]
7          j = j − 1
8      A[j + 1] = key
```

- Correctness. Inside the **for**-loop (*Steps 2-8*) we assume that the subarray $A[1 : i-1]$ is already sorted. *Step 2* stores the value of $A[i]$ in key. This frees up the i^{th} position of A.

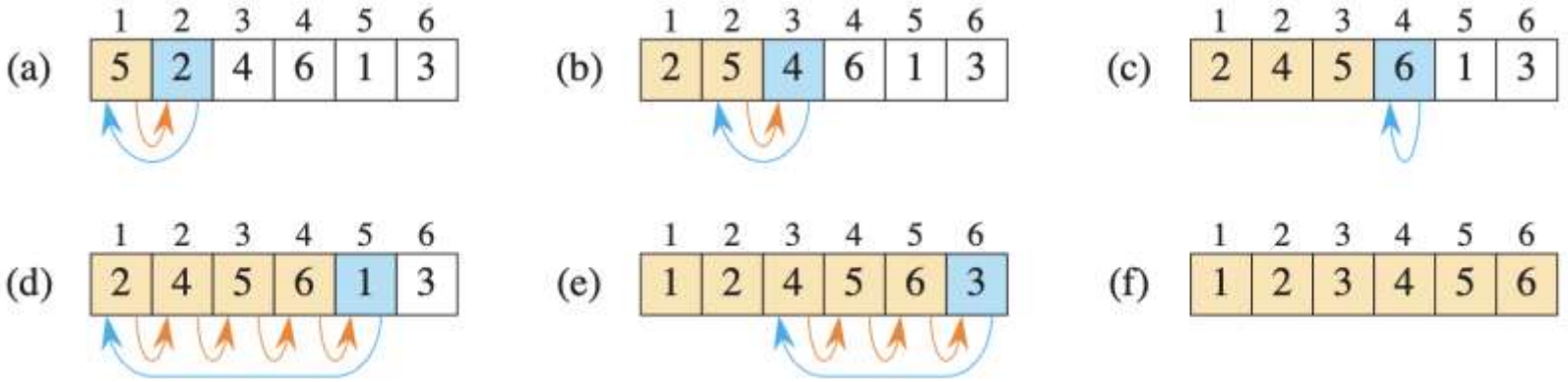
Insertion Sort: Pseudocode

```
INSERTION-SORT(A, n)
1  for i = 2 to n
2      key = A[i]
3      // Insert A[i] into the sorted subarray A[1 : i − 1].
4      j = i − 1
5      while j > 0 and A[j] > key
6          A[j + 1] = A[j]
7          j = j − 1          // At this point A[j+1] is free
8      A[j + 1] = key
```

- Correctness. The **while**-loop (*Steps 5-7*) finds the correct position of key in $A[1 : i]$ by shifting values to the right. *Step 8* then inserts the key in the correct position–the $(j+1)^{th}$ position–once the loop exits.

Insertion Sort: Pseudocode

```
INSERTION-SORT(A, n)
1  for i = 2 to n
2      key = A[i]
3      // Insert A[i] into the sorted subarray A[1 : i − 1].
4      j = i − 1
5      while j > 0 and A[j] > key
6          A[j + 1] = A[j]
7          j = j − 1
8      A[j + 1] = key
```



Insertion Sort: Pseudocode

```
INSERTION-SORT(A, n)
1  for i = 2 to n
2      key = A[i]
3      // Insert A[i] into the sorted subarray A[1 : i − 1].
4      j = i − 1
5      while j > 0 and A[j] > key
6          A[j + 1] = A[j]
7          j = j − 1
8      A[j + 1] = key
```

- Before we discuss the computational/running time (a.k.a **time complexity**) of the algorithm, we need to specify the **computational model**.
- Our model should be reasonably realistic.

The RAM computational model

- We assume a one-processor random access memory (RAM) model of computation.
- In the RAM model, instructions execute *sequentially* with no concurrent operations.

The RAM computational model

- We assume a one-processor random access memory (RAM) model of computation.
- In the RAM model, instructions execute *sequentially* with no concurrent operations.
- Each instruction takes the *same amount of time* as any other instruction.
- Each data access (such as using the value of a variable or storing in a variable) takes the *same amount of time* as any other data access.

The RAM computational model

- We assume a one-processor random access memory (RAM) model of computation.
- The RAM model contains instructions *commonly found in real computers*: arithmetic (addition, subtraction, multiplication, division, etc.), data movement (load, store, copy), and control (conditional and unconditional branching, subroutine call and return).
- RAM model analyses are usually *excellent predictors of performance* on actual machines.

Running time or Time complexity

- We'll describe the running time of an algorithm *as a function of the size of its input*.
- Input size depends on the problem being studied:
  - For an array, size is the *number of items* in it.
  - If the input consists of two integers (as in the integer multiplication problem), then size means the *number of bits* needed to represent the input.

Running time or Time complexity

- We'll describe the running time of an algorithm *as a function of the size of its input*.
- Input size depends on the problem being studied:
  - For an array, size is the *number of items* in it.
  - If the input consists of two integers (as in the integer multiplication problem), then size means the *number of bits* needed to represent the input.
- The running time on a particular input is the number of instructions or data accesses executed.

Worst-case running time

- We'll usually (but not always) focus on finding the worst-case running time, i.e., the longest running time for *any* input of size n. Why?

Worst-case running time

- We'll usually (but not always) focus on finding the worst-case running time, i.e., the longest running time for *any* input of size n. Why?
  - The worst-case running time *gives an upper bound* on the running time for any input.
  - For some algorithms, the *worst-case occurs fairly often*. For example, searching a database when the information is absent.

Worst-case running time

- We'll usually (but not always) focus on finding the worst-case running time, i.e., the longest running time for *any* input of size n. Why?
  - The worst-case running time *gives an upper bound* on the running time for any input.
  - For some algorithms, the *worst-case occurs fairly often*. For example, searching a database when the information is absent.
- We'll be interested in the rate of growth or the order of growth of the worst-case running time.

Insertion Sort: Running time

```
INSERTION-SORT(A, n)
1  for i = 2 to n
2      key = A[i]
3      // Insert A[i] into the sorted subarray A[1 : i − 1].
4      j = i − 1
5      while j > 0 and A[j] > key
6          A[j + 1] = A[j]
7          j = j − 1
8      A[j + 1] = key
```

- The worst-case running time of the **while**-loop (*Steps 5-7*) is at most $c \cdot i$, where c is a constant.
- So, the worst-case running time of the algorithm is at most $c' \cdot n^2$, where c' is a constant.

Insertion Sort: Running time

```
INSERTION-SORT(A, n)
1  for i = 2 to n
2      key = A[i]
3      // Insert A[i] into the sorted subarray A[1 : i − 1].
4      j = i − 1
5      while j > 0 and A[j] > key
6          A[j + 1] = A[j]
7          j = j − 1
8      A[j + 1] = key
```

- Hence, we say, the worst-case running time of Insertion Sort is $O(n^2)$.

Insertion Sort: Running time

```
INSERTION-SORT(A, n)
1  for i = 2 to n
2      key = A[i]
3      // Insert A[i] into the sorted subarray A[1 : i − 1].
4      j = i − 1
5      while j > 0 and A[j] > key
6          A[j + 1] = A[j]
7          j = j − 1
8      A[j + 1] = key
```

- Hence, we say, the worst-case running time of Insertion Sort is $O(n^2)$.

- It's time to introduce the order notations…

Order Notations

Asymptotic growth of functions

- We'll be interested in the asymptotic efficiency of algorithms.
- That is, we'll be concerned with how the running time of an algorithm *grows with the input size*.

Asymptotic growth of functions

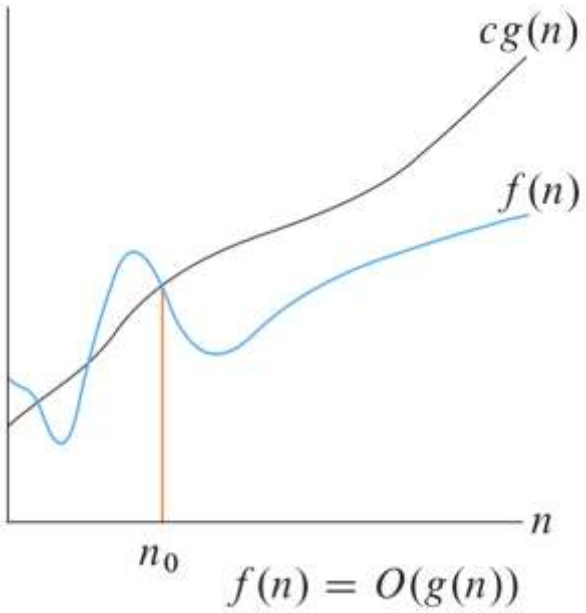
- We'll be interested in the asymptotic efficiency of algorithms.
- That is, we'll be concerned with how the running time of an algorithm *grows with the input size*.

- We usually consider one algorithm to be asymptotically more efficient than another if its *worst-case running time has a lower order of growth*.
- Usually, an algorithm that is asymptotically more efficient is the best choice for all but small inputs.

Big-oh (O) notation

- O-notation describes an asymptotic upper bound.
- Definition. For a function $g(n)$,

$O(g(n)) = \{f(n)$: $\exists$ positive constants c and n_0 s.t. $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0\}$.



Big-oh (O) notation

- O-notation describes an asymptotic upper bound.
- Definition. For a function $g(n)$,

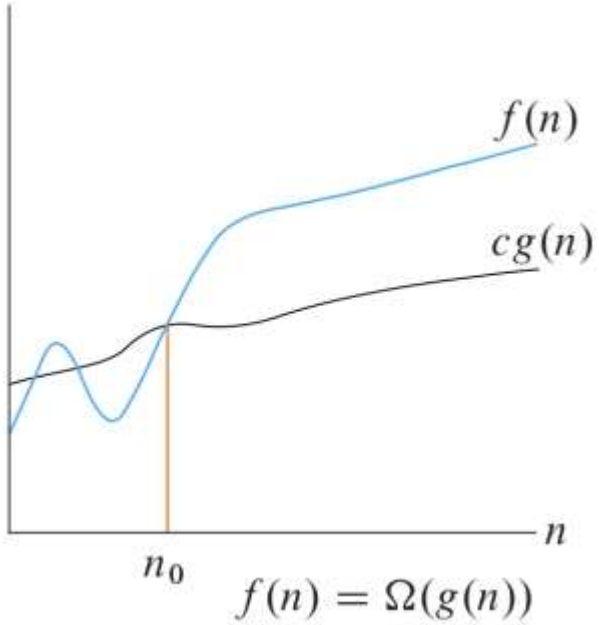
$O(g(n)) = \{f(n)$: $\exists$ positive constants c and n_0 s.t. $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0\}$.

- We usually write "$f(n) = O(g(n))$" and say "$f(n)$ is big-oh of $g(n)$" to mean "$f(n) \in O(g(n))$".
- Example. $4n^2 + 100n + 500 = O(n^2)$. We use asymptotic notations to provide the *simplest bound*.

Big-omega (Ω) notation

- Ω-notation describes an asymptotic lower bound.
- Definition. For a function $g(n)$,

$\Omega(g(n)) = \{f(n)$: $\exists$ positive constants c and n_0 s.t. $0 \leq cg(n) \leq f(n)$ for all $n \geq n_0\}$.



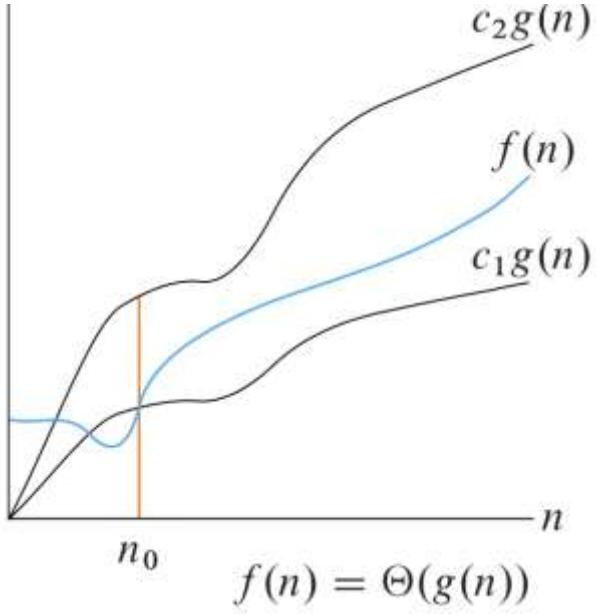
- Example. $2n^3 + 4n^2 + 100n + 500 = \Omega(n^3)$.

Theta (Θ) notation

- Θ-notation describes an asymptotically tight bound.
- Definition. For a function $g(n)$,

$\Theta(g(n)) = \{f(n)$: $\exists$ positive constants c_1, c_2 & n_0 s.t.

$0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0\}$.



- Example. $2n^3 + 4n^2 + 100n + 500 = \Theta(n^3)$.

Theta (Θ) notation

- Θ-notation describes an asymptotically tight bound.
- Definition. For a function $g(n)$,

$\Theta(g(n)) = \{f(n)$: $\exists$ positive constants c_1, c_2 & n_0 s.t.
$0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0\}$.

- Obs. For any two functions $f(n)$ and $g(n)$, we have $f(n) = \Theta\big(g(n)\big)$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

Little-oh (o) notation

- o-notation denotes an upper bound that is not tight.
- Definition. For a function $g(n)$, $o(g(n))$ is the set

$\{f(n) :$ $\forall$ constant $c > 0$, $\exists$ a constant $n_0 > 0$ s.t.
$0 \leq f(n) < cg(n)$ for all $n \geq n_0\}$.

- Example. $20n = o(n^2)$, but $20n^2 \neq o(n^2)$.

Little-oh (o) notation

- o-notation denotes an upper bound that is not tight.
- Definition. For a function $g(n)$, $o(g(n))$ is the set

$\{f(n) :$ $\forall$ constant $c > 0$, $\exists$ a constant $n_0 > 0$ s.t.
$0 \leq f(n) < cg(n)$ for all $n \geq n_0\}$.

- The main difference between O and o is: In $f(n) = O(g(n))$, the condition $0 \leq f(n) \leq cg(n)$ holds *for some* constant $c > 0$, but in $f(n) = o(g(n))$, the condition $0 \leq f(n) < cg(n)$ holds *for all* $c > 0$.

Little-omega (ω) notation

- ω-notation denotes a lower bound that is not tight.
- Definition. For a function $g(n)$, $\omega(g(n))$ is the set

$\{f(n) :$ $\forall$ constant $c > 0$, $\exists$ a constant $n_0 > 0$ s.t.
$0 \leq cg(n) < f(n)$ for all $n \geq n_0\}$.

- Example. $\frac{n^2}{20} = \omega(n)$, but $\frac{n^2}{20} \neq \omega(n^2)$.

Little-omega (ω) notation

- ω-notation denotes a lower bound that is not tight.
- Definition. For a function $g(n)$, $\omega(g(n))$ is the set

$\{f(n) :$ $\forall$ constant $c > 0$, $\exists$ a constant $n_0 > 0$ s.t.
$0 \leq cg(n) < f(n)$ for all $n \geq n_0\}$.

- Example. $\frac{n^2}{20} = \omega(n)$, but $\frac{n^2}{20} \neq \omega(n^2)$.
- Obs. $f(n) = \omega(g(n))$ if and only if $g(n) = o(f(n))$.

Comparing functions

Transitivity:

$$
\begin{aligned}
f(n) &= \Theta(g(n)) \text{ and } g(n) = \Theta(h(n)) \text{ imply } f(n) = \Theta(h(n)) , \\
f(n) &= O(g(n)) \text{ and } g(n) = O(h(n)) \text{ imply } f(n) = O(h(n)) , \\
f(n) &= \Omega(g(n)) \text{ and } g(n) = \Omega(h(n)) \text{ imply } f(n) = \Omega(h(n)) , \\
f(n) &= o(g(n)) \text{ and } g(n) = o(h(n)) \text{ imply } f(n) = o(h(n)) , \\
f(n) &= \omega(g(n)) \text{ and } g(n) = \omega(h(n)) \text{ imply } f(n) = \omega(h(n)) .
\end{aligned}
$$

Reflexivity:

$$
\begin{aligned}
f(n) &= \Theta(f(n)) , \\
f(n) &= O(f(n)) , \\
f(n) &= \Omega(f(n)) .
\end{aligned}
$$

Symmetry:

$$f(n) = \Theta(g(n)) \text{ if and only if } g(n) = \Theta(f(n)) .$$

Comparing functions

Transpose symmetry:

$$f(n) = O(g(n)) \quad \text{if and only if} \quad g(n) = \Omega(f(n)),$$
$$f(n) = o(g(n)) \quad \text{if and only if} \quad g(n) = \omega(f(n)) .$$

Practice problems

Practice Problem 1

Consider the ***searching problem***:

Input: A sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$ stored in array $A[1:n]$ and a value x.

Output: An index i such that x equals $A[i]$ or the special value NIL if x does not appear in A.

Write pseudocode for ***linear search***, which scans through the array from beginning to end, looking for x. ~~Using a loop invariant, prove that your algorithm is~~

Practice Problem 2

Consider the problem of adding two n-bit binary integers a and b, stored in two n-element arrays $A[0:n-1]$ and $B[0:n-1]$, where each element is either 0 or 1, $a = \sum_{i=0}^{n-1} A[i] \cdot 2^i$, and $b = \sum_{i=0}^{n-1} B[i] \cdot 2^i$. The sum $c = a + b$ of the two integers should be stored in binary form in an $(n+1)$-element array $C[0:n]$, where $c = \sum_{i=0}^{n} C[i] \cdot 2^i$. Write a procedure ADD-BINARY-INTEGERS that takes as input arrays A and B, along with the length n, and returns array C holding the sum.

Practice Problem 3

Consider sorting n numbers stored in array $A[1:n]$ by first finding the smallest element of $A[1:n]$ and exchanging it with the element in $A[1]$. Then find the smallest element of $A[2:n]$, and exchange it with $A[2]$. Then find the smallest element of $A[3:n]$, and exchange it with $A[3]$. Continue in this manner for the first $n-1$ elements of A. Write pseudocode for this algorithm, which is known as ***selection sort***. ~~What loop invariant does this algorithm maintain? Why does it~~

- Analyze the worst-case running time of selection sort.

Practice Problem 4

Bubblesort is a popular, but inefficient, sorting algorithm. It works by repeatedly swapping adjacent elements that are out of order. The procedure BUBBLESORT sorts array $A[1:n]$.

```
BUBBLESORT(A, n)
1  for i = 1 to n − 1
2      for j = n downto i + 1
3          if A[j] < A[j − 1]
4              exchange A[j] with A[j − 1]
```

- Question. What is the worst-case running time of BUBBLESORT?

Practice Problem 5

- Is $2^{n+1} = O(2^n)$? Is $2^{2n} = O(2^n)$?

- Prove that $o(g(n)) \cap \omega(g(n))$ is the empty set.

- Show that $k \lg k = \Theta(n)$ implies $k = \Theta(n / \lg n)$.

Practice Problem 6

Let

$$p(n) = \sum_{i=0}^{d} a_i n^i ,$$

where $a_d > 0$, be a degree-d polynomial in n, and let k be a constant. Use the definitions of the asymptotic notations to prove the following properties.

a. If $k \geq d$, then $p(n) = O(n^k)$.

b. If $k \leq d$, then $p(n) = \Omega(n^k)$.

c. If $k = d$, then $p(n) = \Theta(n^k)$.

d. If $k > d$, then $p(n) = o(n^k)$.

e. If $k < d$, then $p(n) = \omega(n^k)$.

Practice Problem 7

Indicate, for each pair of expressions (A, B) in the table below whether A is O, o, Ω, ω, or Θ of B. Assume that $k \geq 1$, $\epsilon > 0$, and $c > 1$ are constants. Write your answer in the form of the table with "yes" or "no" written in each box.

	A	B	O	o	Ω	ω	Θ
a.	$\lg^k n$	n^ϵ					
b.	n^k	c^n					
c.	$\sqrt{n}$	$n^{\sin n}$					
d.	2^n	$2^{n/2}$					
e.	$n^{\lg c}$	$c^{\lg n}$					
f.	$\lg(n!)$	$\lg(n^n)$					

$$\lg n = \log_2 n$$