CSE-201
Data Structure & Algorithm

Lecture - 1

Introduction to Data Structures

- **Data Structures**

The logical or mathematical model of a particular organization of data is called a data structure.

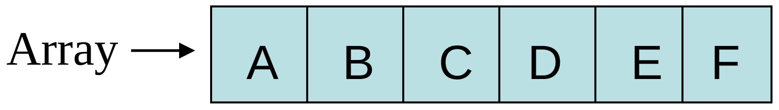
- **Types of Data Structure**

1. Linear Data Structure

   Example: Arrays, Linked Lists, Stacks, Queues

2. Nonlinear Data Structure

   Example: Trees, Graphs



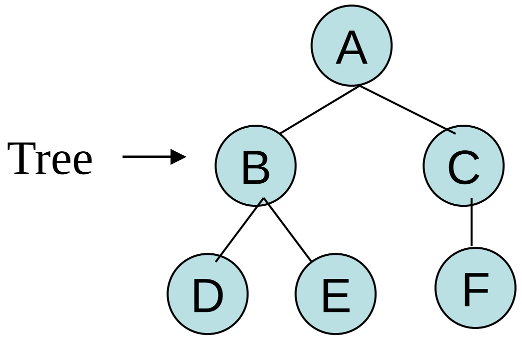


Figure: Linear and nonlinear structures

Choice of Data Structures

The choice of data structures depends on two considerations:

1. It must be rich enough in structure to mirror the actual relationships of data in the real world.
2. The structure should be simple enough that one can effectively process data when necessary.

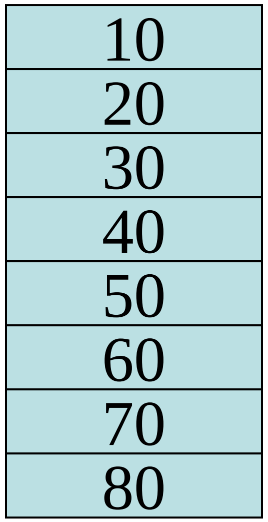


Figure 2: Array with 8 items

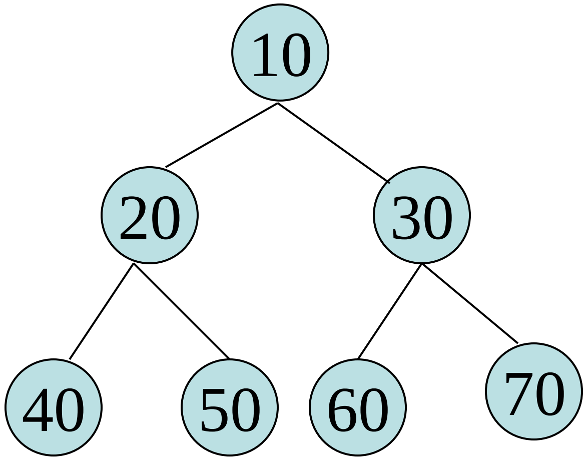


Figure 3: Tree with 8 nodes

Data Structure Operations

1. **Traversing:** Accessing each record exactly once so that certain items in the record may be processed.
2. **Searching:** Finding the location of the record with a given key value.
3. **Inserting:** Adding a new record to the structure.
4. **Deleting:** Removing a record from the structure.
5. **Sorting:** Arranging the records in some logical order.
6. **Merging:** Combing the records in two different sorted files into a single sorted file.

Algorithms

It is a well-defined set of instructions used to solve a particular problem.

Example:

Write an algorithm for finding the location of the largest element of an array Data.

```
Largest-Item (Data, N, Loc)
1.  set k:=1, Loc:=1 and Max:=Data[1]
2.  while k<=N repeat steps 3, 4
3.      If Max < Data[k] then Set Loc:=k and Max:=Data[k]
4.      Set k:=k+1
5.  write: Max and Loc
6.  exit
```

Complexity of Algorithms

- The complexity of an algorithm M is the function f(n) which gives the running time and/or storage space requirement of the algorithm in terms of the size n of the input data.

- Two types of complexity

  1. Time Complexity
  2. Space Complexity

- Sometimes the choice of data structures involves a time-space tradeoff. By increasing the amount of space for storing the data, it is possible to reduce the time needed for processing the data, or vice versa.

Analyzing Algorithms

- Predict the amount of resources required:
  - memory: how much space is needed?
  - computational time: how fast the algorithm runs?
- FACT: running time grows with the size of the input
- Input size (number of elements in the input)
  - Size of an array, polynomial degree, # of elements in a matrix, # of bits in the binary representation of the input, vertices and edges in a graph

Def: Running time = the number of primitive operations (steps) executed before termination

- Arithmetic operations (+, -, *), data movement, control, decision making (*if, while*), comparison

Algorithm Analysis: Example

- *Alg.:* MIN (a[1], …, a[n])

```
m ← a[1];
for i ← 2 to n
    if a[i] < m
      then m ← a[i];
```

- **Running time**:
  - the number of primitive operations (steps) executed before termination

$T(n) = 1$ [first step] + (n)[for loop] + $(n-1)$ [if condition] + $(n-1)$ [the assignment in then] = $3n - 1$

- Order (rate) of growth:
  - The leading term of the formula
  - Expresses the asymptotic behavior of the algorithm

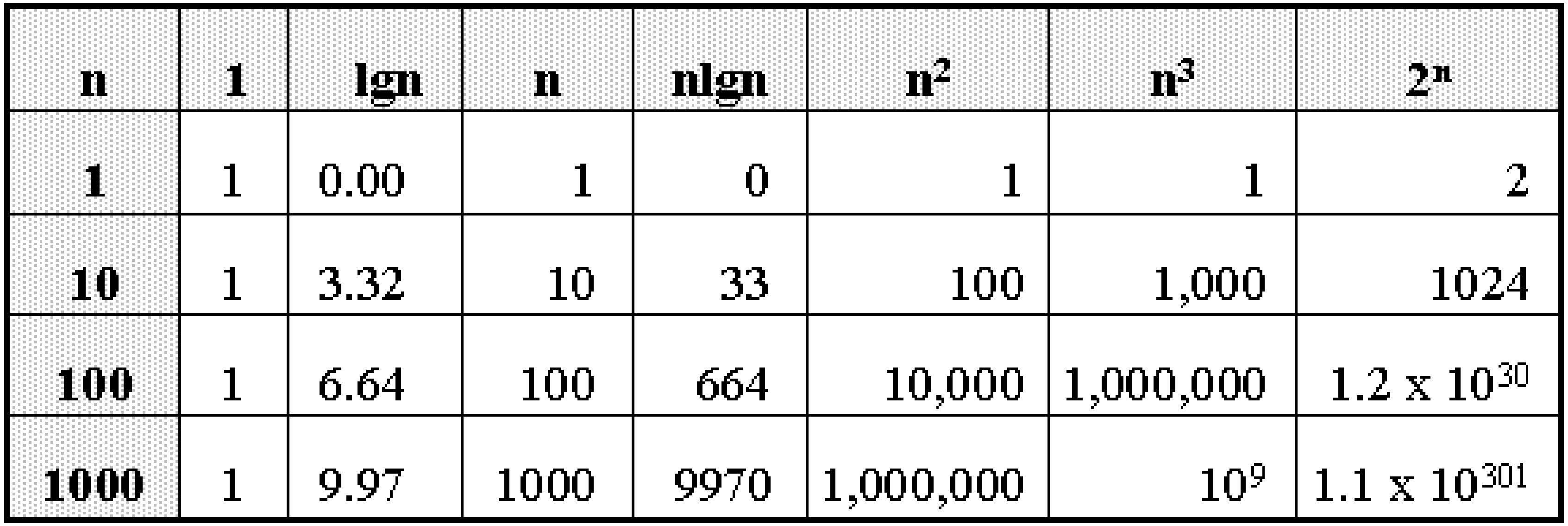
Typical Running Time Functions

- 1 (constant running time):
  - Instructions are executed once or a few times
- logN (logarithmic)
  - A big problem is solved by cutting the original problem in smaller sizes, by a constant fraction at each step
- N (linear)
  - A small amount of processing is done on each input element
- N logN
  - A problem is solved by dividing it into smaller problems, solving them independently and combining the solution

Typical Running Time Functions

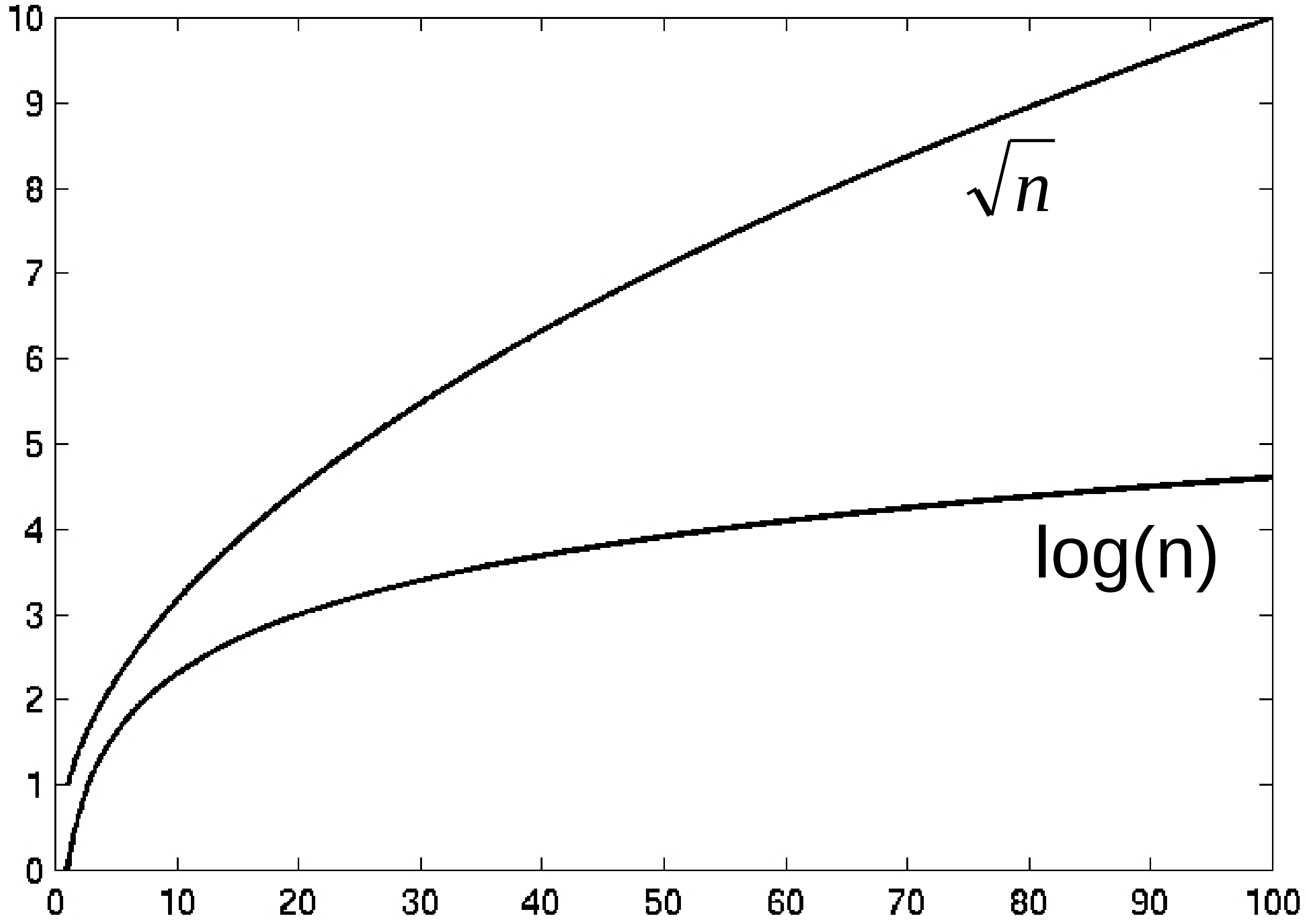
- N^2 (quadratic)
  - Typical for algorithms that process all pairs of data items (double nested loops)
- N^3 (cubic)
  - Processing of triples of data (triple nested loops)
- N^K (polynomial)
- 2^N (exponential)
  - Few exponential algorithms are appropriate for practical use

Growth of Functions

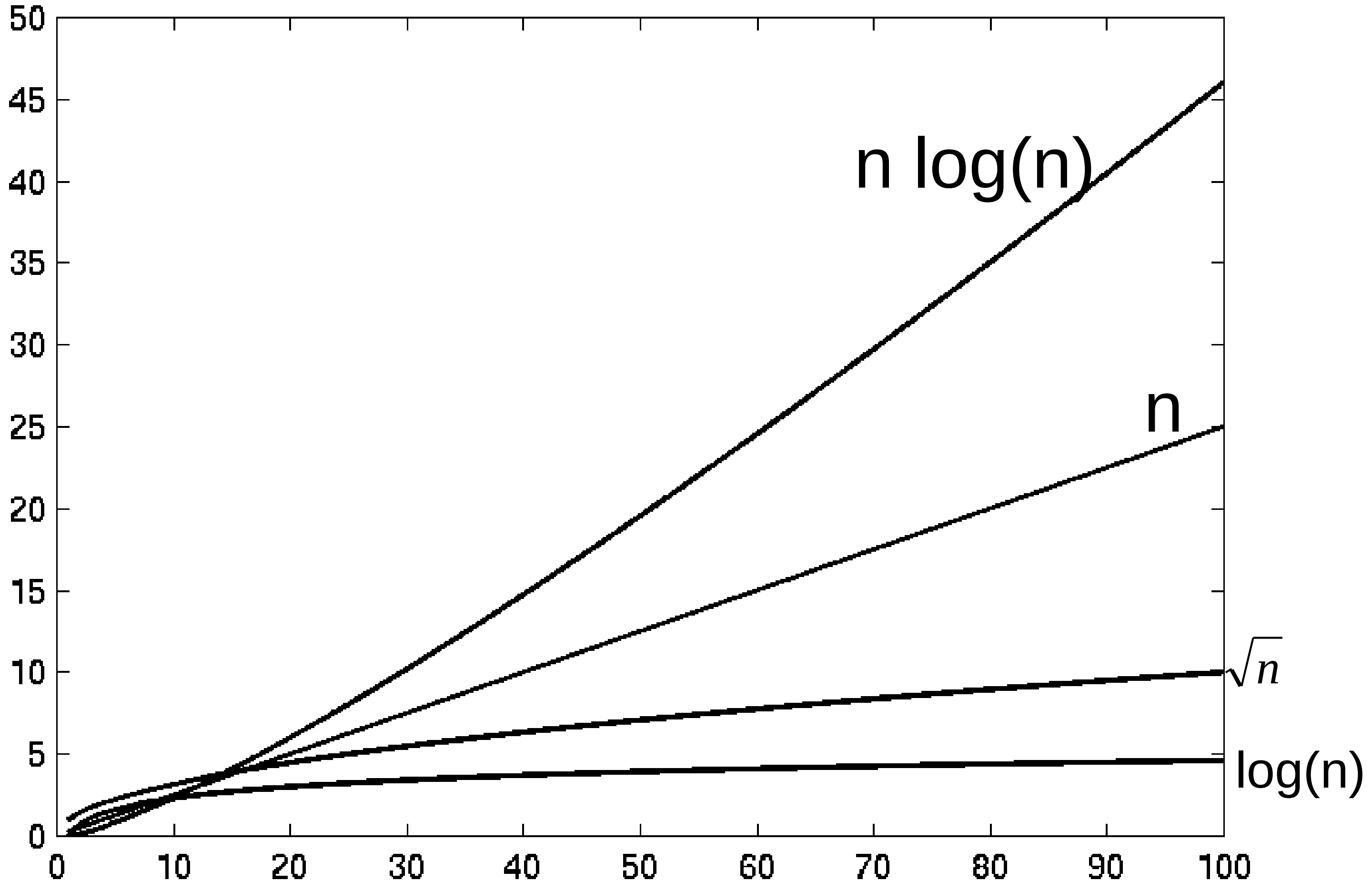


n	1	lgn	n	nlgn	n^2	n^3	2^n
1	1	0.00	1	0	1	1	2
10	1	3.32	10	33	100	1,000	1024
100	1	6.64	100	664	10,000	1,000,000	1.2×10^{30}
1000	1	9.97	1000	9970	1,000,000	10^9	1.1×10^{301}

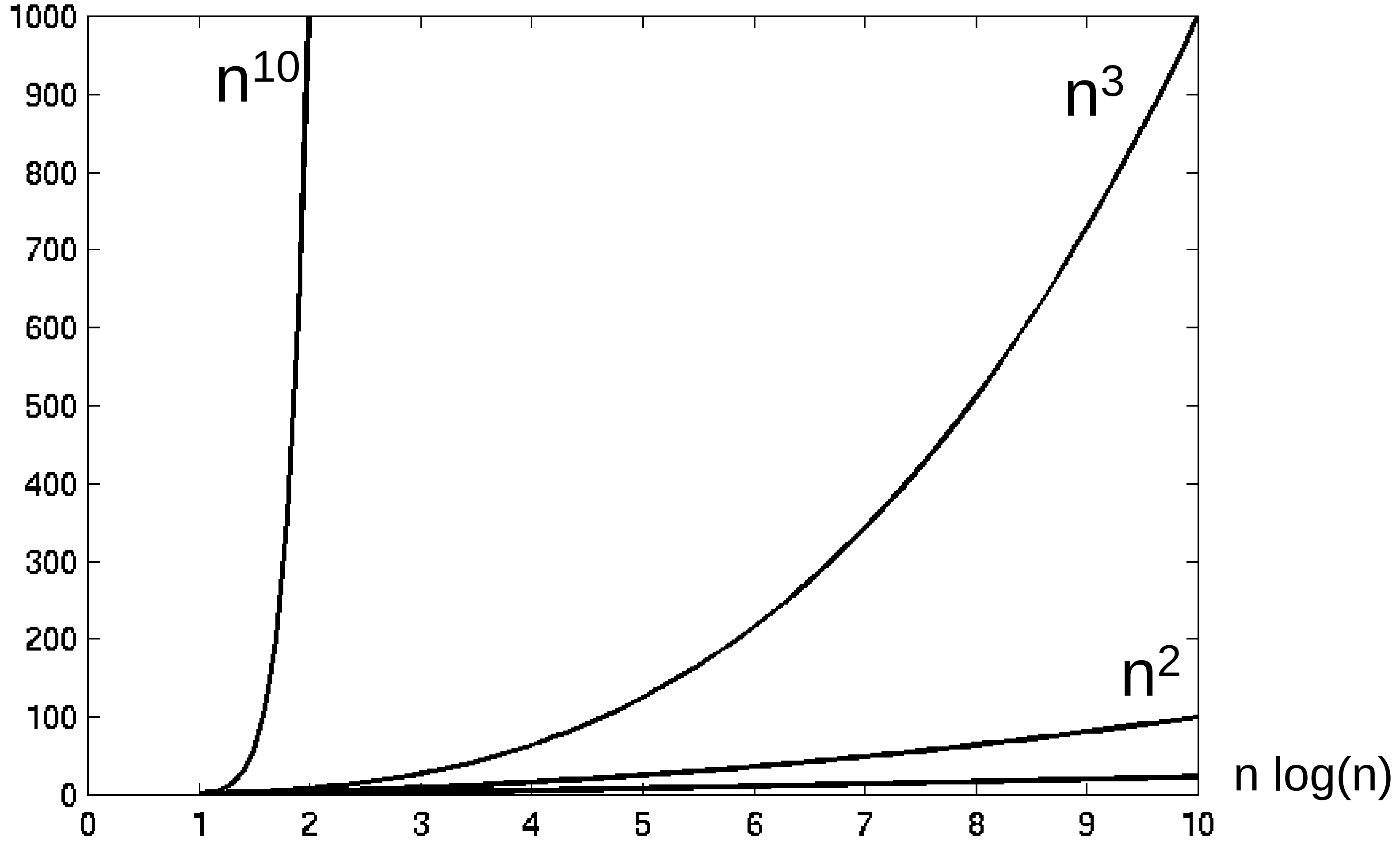
Complexity Graphs



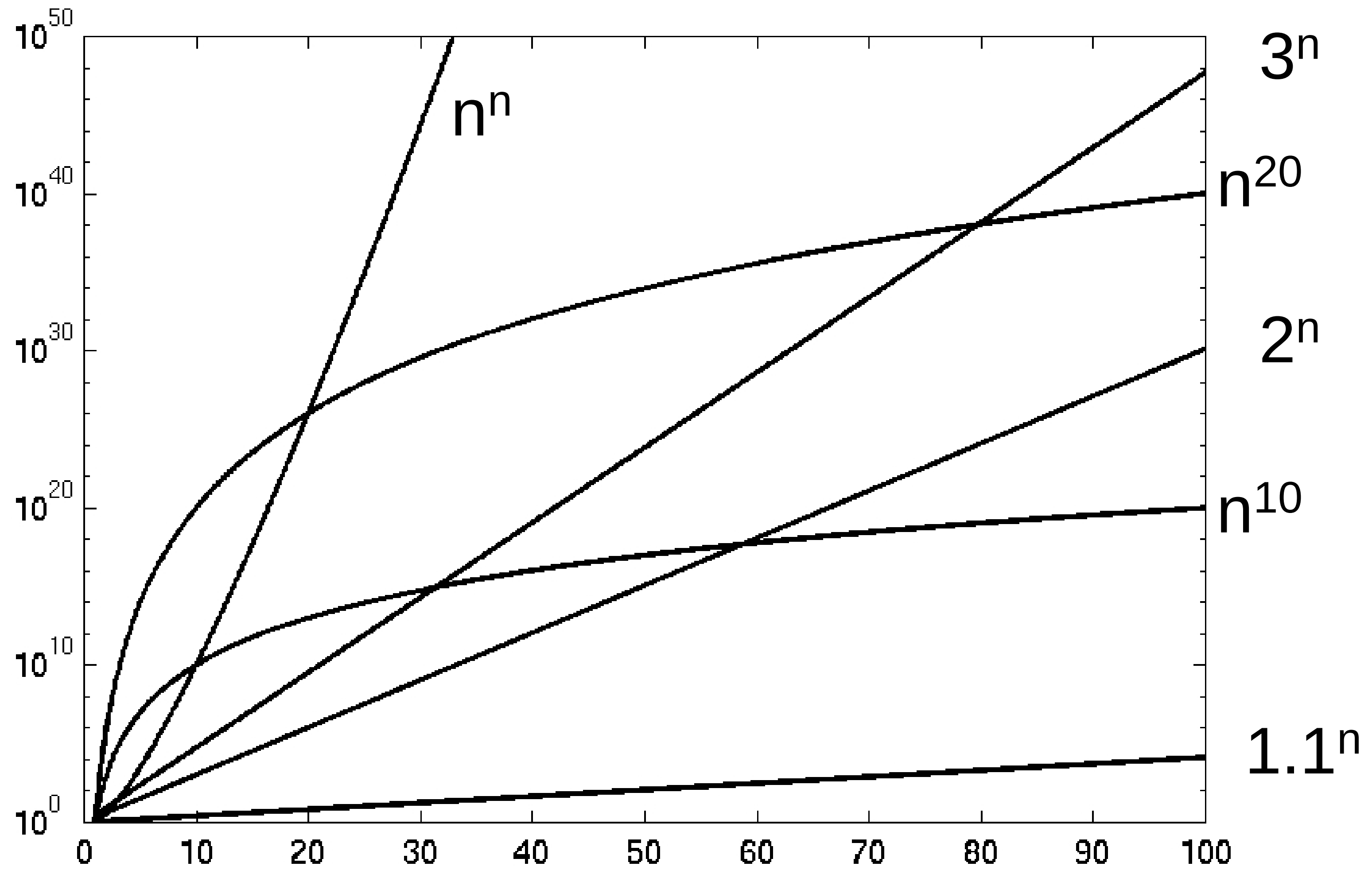
Complexity Graphs



Complexity Graphs



Complexity Graphs (log scale)



Algorithm Complexity

- Worst Case Complexity:
  - the function defined by the *maximum* number of steps taken on any instance of size *n*
- Best Case Complexity:
  - the function defined by the *minimum* number of steps taken on any instance of size *n*
- Average Case Complexity:
  - the function defined by the *average* number of steps taken on any instance of size *n*

Is input size enough to determine time complexity unambiguously?

```
int find_a(char *str)
{
    int i;
    for (i = 0; str[i]; i++)
    {
        if (str[i] == 'a')
            return i;
    }
    return -1;
}
```

- **Time complexity:** $O(n)$

What is the time complexity of the above algorithm?

Depends on the input (str):

- $\Theta(1)$ for best possible input which starts with ‘a’ e.g. when str = “alibaba”
- $\Theta(n)$ for worst possible input which doesn’t contain any ‘a’ e.g. when str = “nitin”

Types of Time Complexity Analysis

- So how does the running time vary with respect to various input?

- Three scenarios
  - Best case
    - $best_{find\_a} = \min_{x \in X_n} runtime_{find\_a}$
  - Worst case
    - $worst_{find\_a} = \max_{x \in X_n} runtime_{find\_a}$
  - Average case
    - $avg_{find\_a} = \frac{1}{|X_n|} \sum_{x \in X_n} runtime_{find\_a}$

Types of Time Complexity Analysis

- **Worst-case:** (usually done)
  - Running time on worst possible input

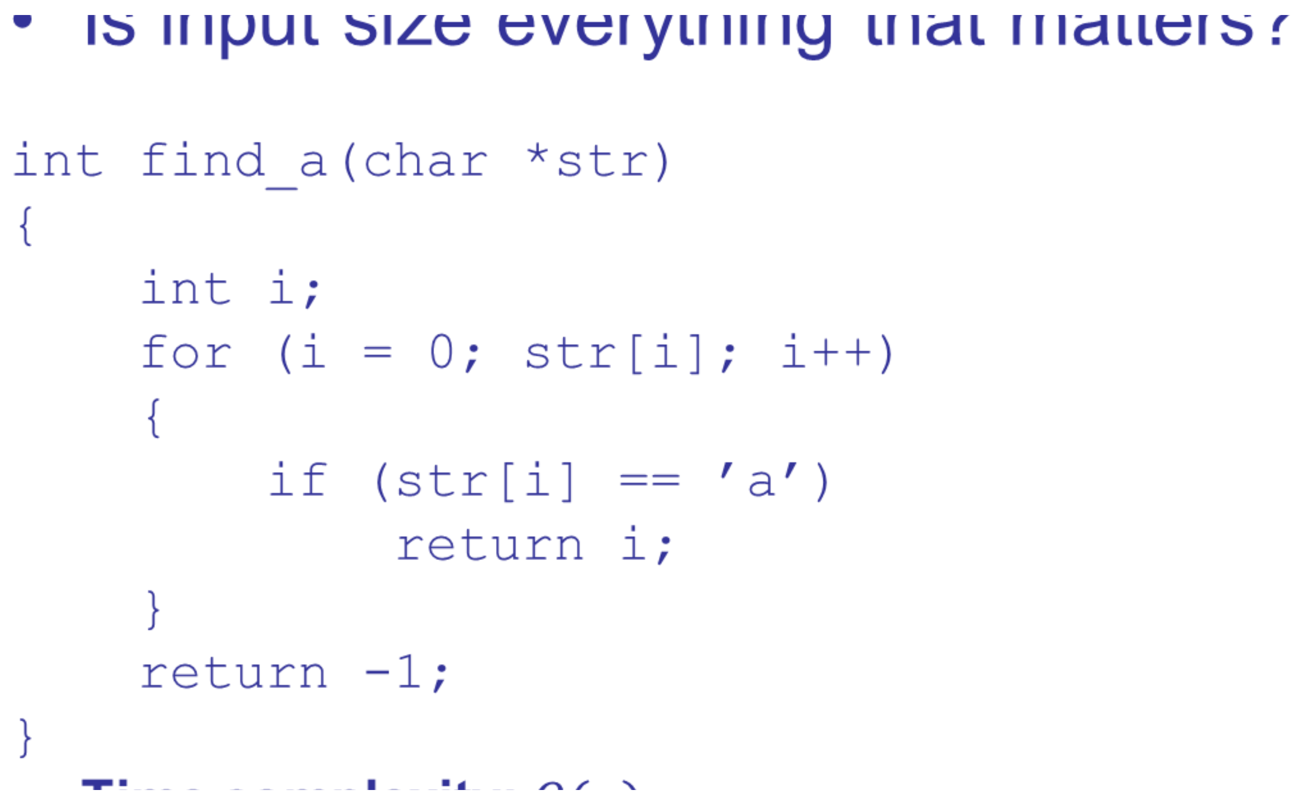
- **Best-case:** (bogus)
  - Running time on best possible input

- **Average-case:** (sometimes done)
  - ➢ We take all possible inputs and calculate computing time for all of the inputs sum all the calculated values and divide the sum by total number of inputs
  - ➢ We must know (or predict) distribution of inputs to calculate this
  - ➢ Often we compute it on an assumed distribution of inputs using *expectation*, in which case it is called **Expected running time**
  - ➢ This is typically calculated to show that although our algorithm's worst case running time is not better, its expected time is better than its competitors

We are typically interested in the runtime of an algorithm in the <u>worst case</u> scenario. Because it provides us a guarantee that the algorithm won't take any longer than this, no matter which input is given.

– **Amortized running time** (discussed later)

Example of Best & Worst Case Analysis



• Is input size everything that matters?

```
int find_a(char *str)
{
    int i;
    for (i = 0; str[i]; i++)
    {
        if (str[i] == 'a')
            return i;
    }
    return -1;
}
```

Best case input: strings that contain an ‘a’ in its first index

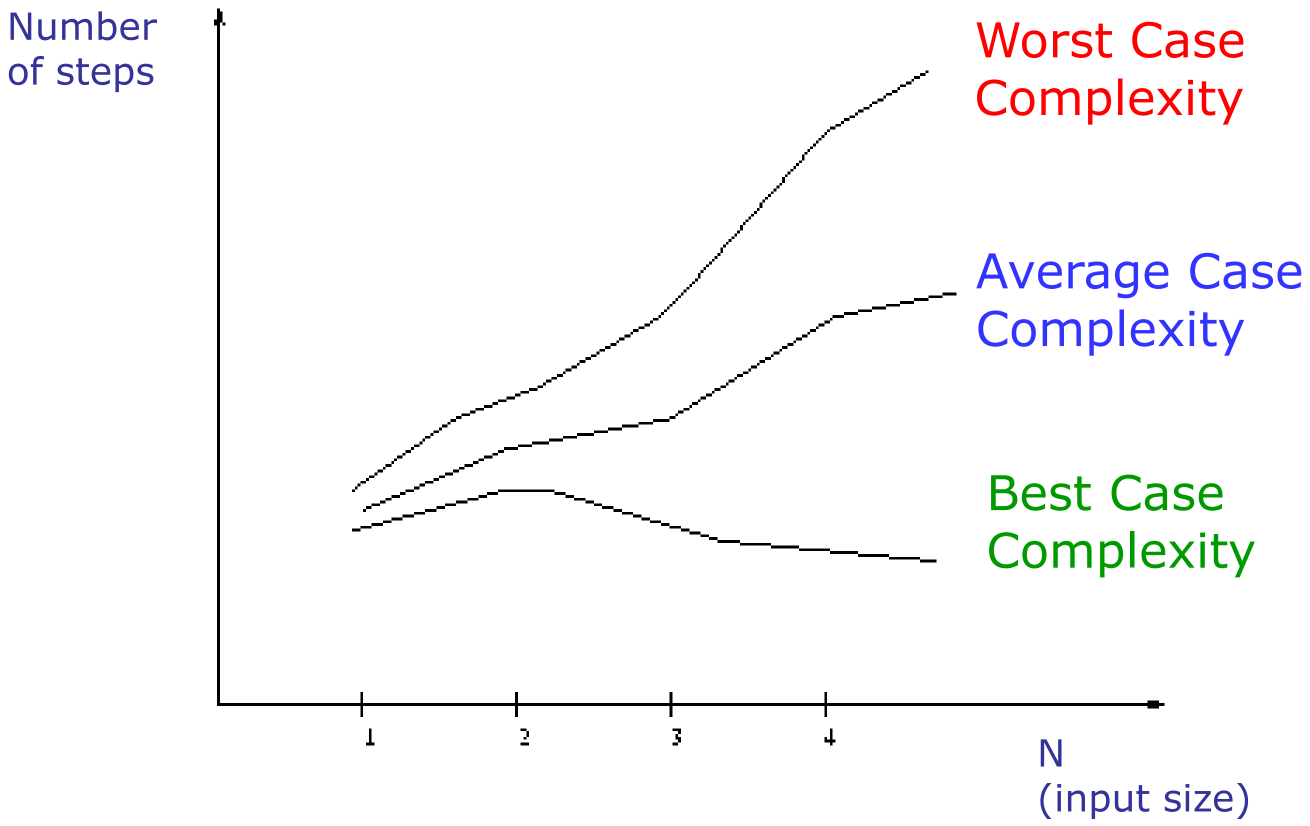
Best case time complexity: ***O(1)***

Worst case input: strings that do not contain any ‘a’

Worst case time complexity: ***O(n)***

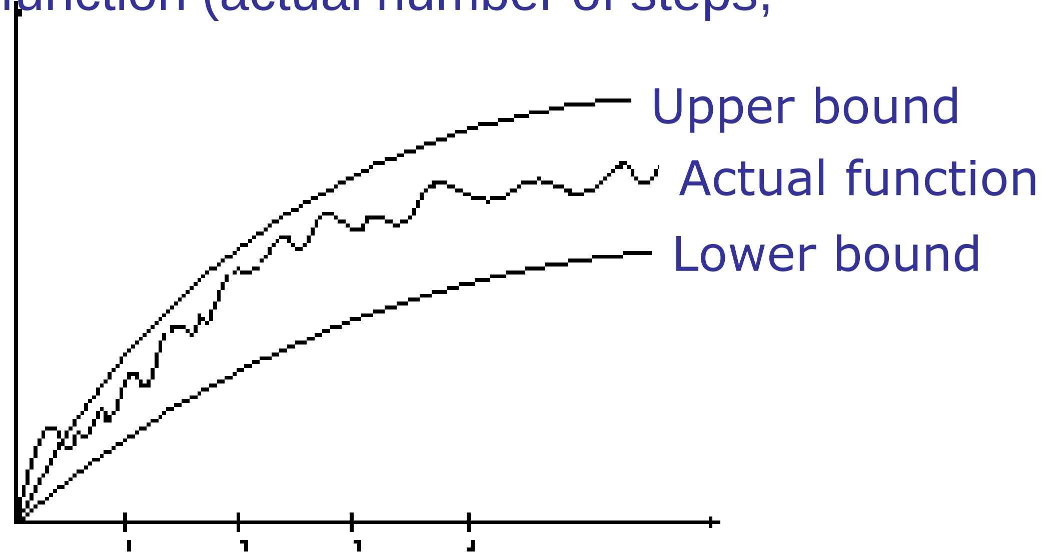
Average time complexity: ***O(n)***, ***how?***

Best, Worst, and Average Case Complexity



Doing the Analysis

- It's hard to estimate the running time exactly
  - Best case depends on the input
  - Average case is difficult to compute
  - So we usually focus on worst case analysis
    - Easier to compute
    - Usually close to the actual running time
- Strategy: find a function (an equation) that, for large n, is an upper bound to the actual function (actual number of steps, memory usage, etc.)



Motivation for Asymptotic Analysis

- An *exact computation* of worst-case running time can be difficult
  - Function may have many terms:
    - $4n^2 - 3n \log n + 17.5\, n - 43\, n^{2/3} + 75$
- An *exact computation* of worst-case running time is unnecessary
  - Remember that we are already approximating running time by using RAM model

Classifying functions by their Asymptotic Growth Rates (1/2)

- asymptotic growth rate, asymptotic order, or order of functions
  - Comparing and classifying functions that ignores
    - *constant factors* and
    - *small inputs*.
- The Sets big oh O(g), big theta Θ(g), big omega Ω(g)

Classifying functions by their Asymptotic Growth Rates (2/2)

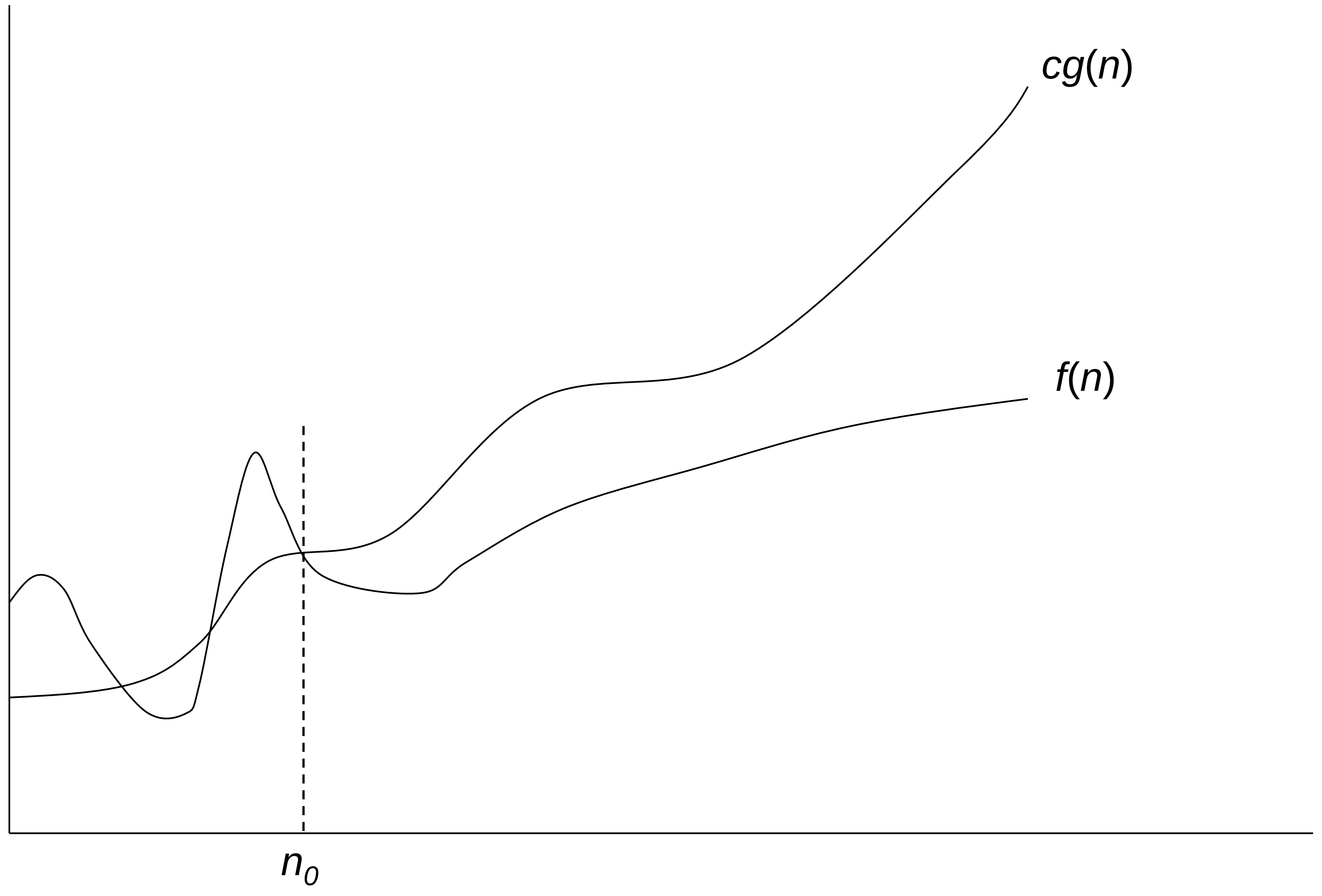
- O(g(n)), Big-Oh of g of n, the Asymptotic Upper Bound;
- Θ(g(n)), Theta of g of n, the Asymptotic Tight Bound; and
- Ω(g(n)), Omega of g of n, the Asymptotic Lower Bound.

Big-O

$$f(n) = O(g(n)): \text{there exist positive constants } c \text{ and } n_0 \text{ such that}$$
$$0 \le f(n) \le cg(n) \text{ for all } n \ge n_0$$

- What does it mean?
  - If $f(n) = O(n^2)$, then:
    - $f(n)$ can be larger than n^2 sometimes, **but…**
    - We can choose some constant c and some value n_0 such that for **every** value of n larger than n_0 : $f(n) < cn^2$
    - That is, for values larger than n_0, $f(n)$ is never more than a constant multiplier greater than n^2
    - Or, in other words, $f(n)$ does not grow more than a constant factor faster than n^2.

Visualization of $O(g(n))$



Examples

- $2n^2 = O(n^3)$: $2n^2 \leq cn^3 \Rightarrow 2 \leq cn \Rightarrow c = 1$ and $n_0 = 2$

- $n^2 = O(n^2)$: $n^2 \leq cn^2 \Rightarrow c \geq 1 \Rightarrow c = 1$ and $n_0 = 1$

- $1000n^2+1000n = O(n^2)$:

$1000n^2+1000n \leq cn^2 \leq cn^2+ 1000n \Rightarrow c=1001$ and $n_0 = 1$

- $n = O(n^2)$: $n \leq cn^2 \Rightarrow cn \geq 1 \Rightarrow c = 1$ and $n_0 = 1$

Big-O

$$2n^2 = O(n^2)$$

$$1{,}000{,}000n^2 + 150{,}000 = O(n^2)$$

$$5n^2 + 7n + 20 = O(n^2)$$

$$2n^3 + 2 \neq O(n^2)$$

$$n^{2.1} \neq O(n^2)$$

29

More Big-O

- Prove that: $20n^2 + 2n + 5 = O(n^2)$
- Let $c = 21$ and $n_0 = 4$
- $21n^2 > 20n^2 + 2n + 5$ for all $n > 4$

  $n^2 > 2n + 5$ for all $n > 4$

  TRUE

Tight bounds

- We generally want the tightest bound we can find.
- While it is true that $n^2 + 7n$ is in $O(n^3)$, it is more interesting to say that it is in $O(n^2)$

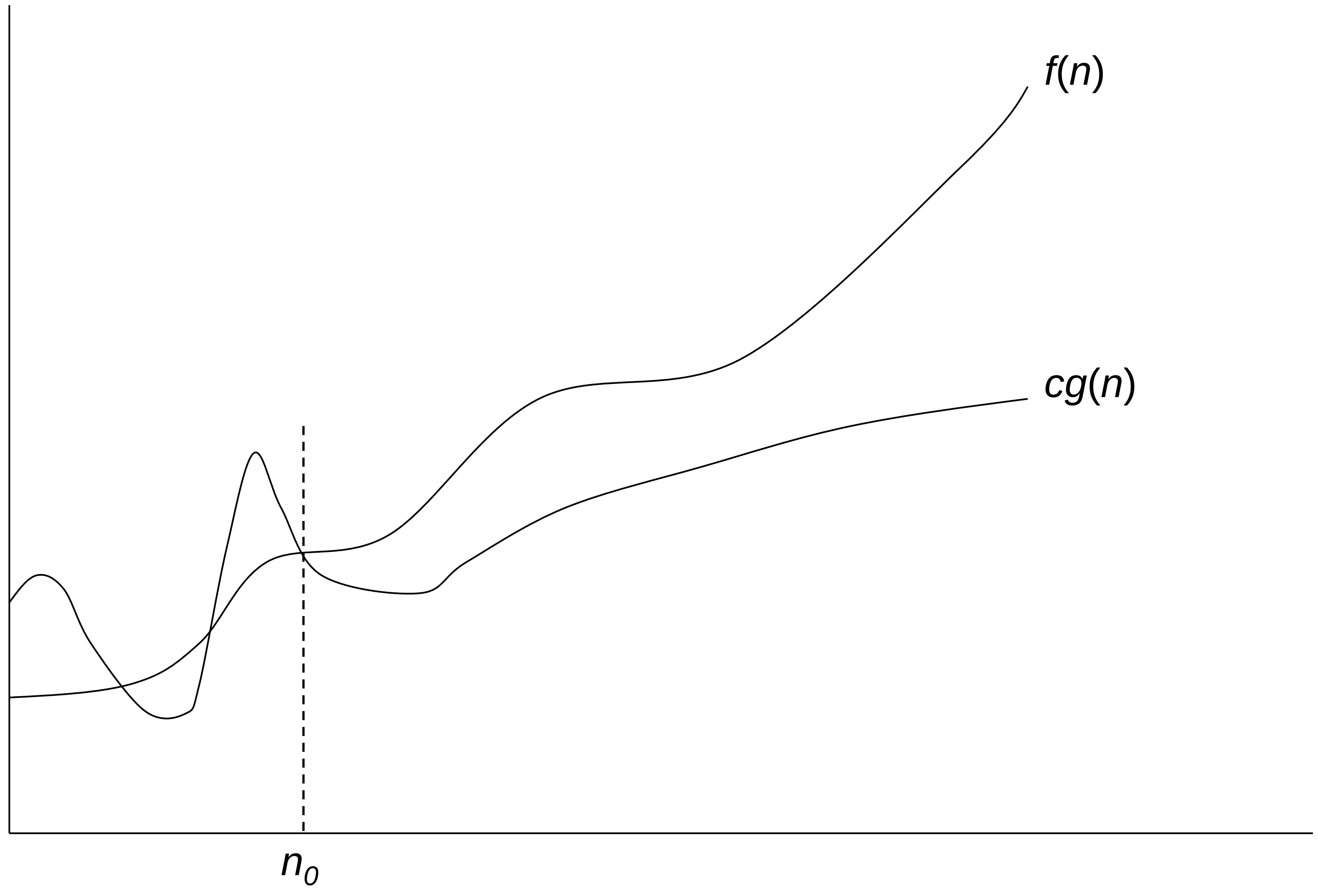
Big Omega – Notation

- $\Omega()$ – A **lower** bound

$$f(n)=\Omega(g(n)): \text{there exist positive constants } c \text{ and } n_0 \text{ such that}$$
$$0 \leq f(n) \geq cg(n) \text{ for all } n \geq n_0$$

  - $n^2 = \Omega(n)$
  - Let $c = 1$, $n_0 = 2$
  - For all $n \geq 2$, $n^2 > 1 \times n$

Visualization of $\Omega(g(n))$



33

Θ-notation

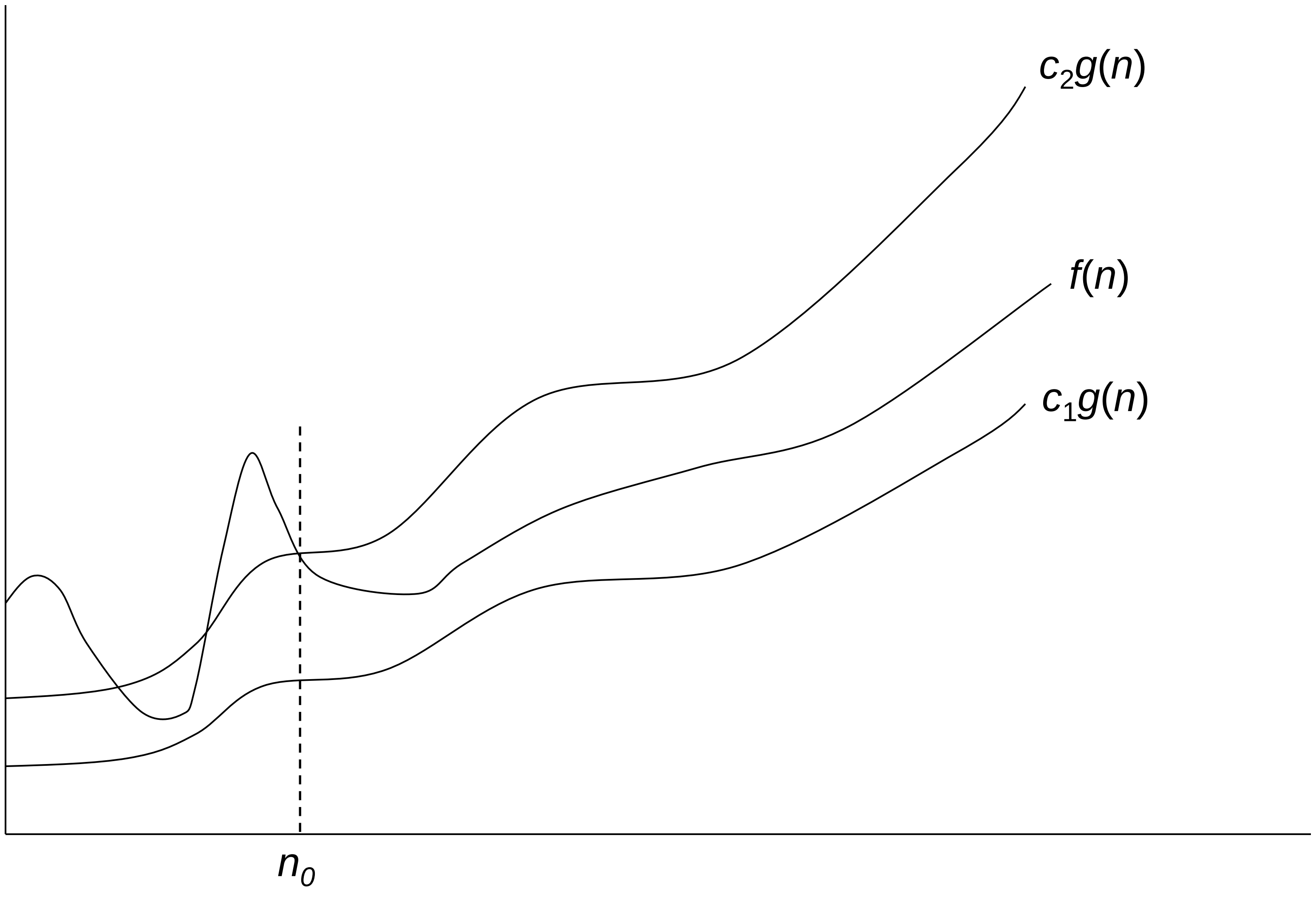
- Big-*O* is not a tight upper bound. In other words $n = O(n^2)$
- Θ provides a tight bound

$$f(n)=\Theta(g(n)): \text{there exist positive constants } c_1, c_2, \text{ and } n_0 \text{ such that}$$
$$0 \le c_1 g(n) \le f(n) \le c_2 g(n) \text{ for all } n \ge n_0$$

- In other words,

$$f(n)=\Theta(g(n)) \Rightarrow f(n)=O(g(n)) \text{ AND } f(n)=\Omega(g(n))$$

Visualization of Θ($g(n)$)



A Few More Examples

- $n = O(n^2) \neq \Theta(n^2)$
- $200n^2 = O(n^2) = \Theta(n^2)$
- $n^{2.5} \neq O(n^2) \neq \Theta(n^2)$

Example 2

- Prove that: $20n^3 + 7n + 1000 = \Theta\left(n^3\right)$
- Let $c = 21$ and $n_0 = 10$
- $21n^3 > 20n^3 + 7n + 1000$ for all $n > 10$

  $n^3 > 7n + 5$ for all $n > 10$

  TRUE, but we also need…
- Let $c = 20$ and $n_0 = 1$
- $20n^3 < 20n^3 + 7n + 1000$ for all $n \geq 1$

  TRUE

Example 3

- Show that $2^n + n^2 = O(2^n)$
- Let $c = 2$ and $n_0 = 5$

$$2 \times 2^n > 2^n + n^2$$

$$2^{n+1} > 2^n + n^2$$

$$2^{n+1} - 2^n > n^2$$

$$2^n(2-1) > n^2$$

$$2^n > n^2 \quad \forall n \geq 5 \quad \checkmark$$

Asymptotic Notations - Examples

- Θ notation
  - $n^2/2 - n/2 \quad = \Theta(n^2)$
  - $(6n^3 + 1)\lg n/(n + 1) = \Theta(n^2 \lg n)$
  - n vs. $n^2 \quad n \neq \Theta(n^2)$
- Ω notation
  - n^3 vs. $n^2 \quad n^3 = \Omega(n^2)$
  - n vs. $\log n \quad n = \Omega(\log n)$
  - n vs. $n^2 \quad n \neq \Omega(n^2)$
- O notation
  - $2n^2$ vs. $n^3 \quad 2n^2 = O(n^3)$
  - n^2 vs. $n^2 \quad n^2 = O(n^2)$
  - n^3 vs. $n\log n \quad n^3 \neq O(n\lg n)$

Asymptotic Notations - Examples

- For each of the following pairs of functions, either f(n) is O(g(n)), f(n) is Ω(g(n)), or f(n) = Θ(g(n)). Determine which relationship is correct.

– $f(n) = \log n^2$; $g(n) = \log n + 5$	$f(n) = \Theta(g(n))$
– $f(n) = n$; $g(n) = \log n^2$	$f(n) = \Omega(g(n))$
– $f(n) = \log \log n$; $g(n) = \log n$	$f(n) = O(g(n))$
– $f(n) = n$; $g(n) = \log^2 n$	$f(n) = \Omega(g(n))$
– $f(n) = n \log n + n$; $g(n) = \log n$	$f(n) = \Omega(g(n))$
– $f(n) = 10$; $g(n) = \log 10$	$f(n) = \Theta(g(n))$
– $f(n) = 2^n$; $g(n) = 10n^2$	$f(n) = \Omega(g(n))$
– $f(n) = 2^n$; $g(n) = 3^n$	$f(n) = O(g(n))$

Simplifying Assumptions

- 1. If $f(n) = O(g(n))$ and $g(n) = O(h(n))$, then $f(n) = O(h(n))$
- 2. If $f(n) = O(kg(n))$ for any $k > 0$, then $f(n) = O(g(n))$
- 3. If $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$,
- then $f_1(n) + f_2(n) = O(\max(g_1(n), g_2(n)))$
- 4. If $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$,
- then $f_1(n) * f_2(n) = O(g_1(n) * g_2(n))$

Some Examples

- Determine the time complexity for the following algorithm.

```
count = 0; //c1
•  i = 0; //c1
•  while(i < n){     //(n+1)c2
•     count++; //nc3
•     i++; //nc3
}
```

- So total time taken by this code,
- $T(n) = (2c_1 + c_2) + n\,(c_2+2c_3)$ which is $\leq (2c_1+ 2c_2+2c_3)\, n$
- for very large values of n. Therefore we say that the time complexity of this code is O(n).

Usually, when the code is very simple (like above), we simply say that the runtime of this code is O(n), without doing the straightforward mathematical analysis. When the code is not so simple, we do the actual analysis to compute the time complexity.

Some Examples

- Determine the time complexity for the following algorithm.

```
count = 0;
for(i=0; i<10000; i++)
    count++;
```

Some Examples

- Determine the time complexity for the following algorithm.

$\Theta(1)$

```
count = 0;
for(i=0; i<10000; i++)
    count++;
```

Some Examples

- Determine the time complexity for the following algorithm.

```
count = 0;
for(i=0; i<n; i++)
    count++;
```

Some Examples

- Determine the time complexity for the following algorithm.

$\Theta(n)$

```
count = 0;
for(i=0; i<n; i++)
    count++;
```

Some Examples

- Determine the time complexity for the following algorithm.

```
sum = 0;
for(i=0; i<n; i++)
    for(j=0; j<n; j++)
        sum += arr[i][j];
```

$O(n^2)$

Some Examples

- Determine the time complexity for the following algorithm.

```
sum = 0;
for(i=0; i<n; i++)
    for(j=0; j<n; j++)
        sum += arr[i][j];
```

$\Theta(n^2)$

Some Examples

- Determine the time complexity for the following algorithm.

```
sum = 0;
for(i=1; i<=n; i=i*2)
    sum += i;
```

Some Examples

- Determine the time complexity for the following algorithm.

```
sum = 0;
for(i=1; i<=n; i=i*2)
    sum += i;
```

Θ(lg n)

WHY? Show mathematical analysis to prove that it is indeed Θ(lg n)

Some Examples

- Determine the time complexity for the following algorithm.

```
sum = 0;
```

-
  ```
  for(i=1; i<=n; i=i*2)
     for(j=0; j<n; j++)
           sum += i*j;
  ```

Some Examples

- Determine the time complexity for the following algorithm.

```
sum = 0;
```

Θ(n lg n)

```
• for(i=1; i<=n; i=i*2)
      for(j=0; j<n; j++)
            sum += i*j;
```

Why? Outer for loop runs Θ(lg n) times (prove it!) and for each iteration of outer loop, the inner loop runs Θ(n) times

Some Examples

- Determine the time complexity for the following algorithm.

```
sum = 0;
```

-
  ```
  for(i=1; i<=n; i=i*2)
     for(j=0; j<i; j++)
          sum += i*j;
  ```

Some Examples

- Determine the time complexity for the following algorithm.

```
sum = 0;
•  for(i=1; i<=n; i=i*2)
       for(j=0; j<i; j++)
           sum += i*j;
```

Loose Upper Bound: ***O(n lg n)***

Tight Upper Bound: ***O(n)***

Asymptotic Tight Bound: ***Θ(n)***

WHY?

Recurrences

Def.: Recurrence = an equation or inequality that describes a function in terms of its value on smaller inputs, and one or more base cases

- E.g.: $T(n) = T(n-1) + n$
- Useful for analyzing recurrent algorithms
- Methods for solving recurrences
  - Substitution method
  - Recursion tree method
  - Master method
  - Iteration method