CSE-221
Algorithms

Introduction to Algorithms

Algorithm Definition

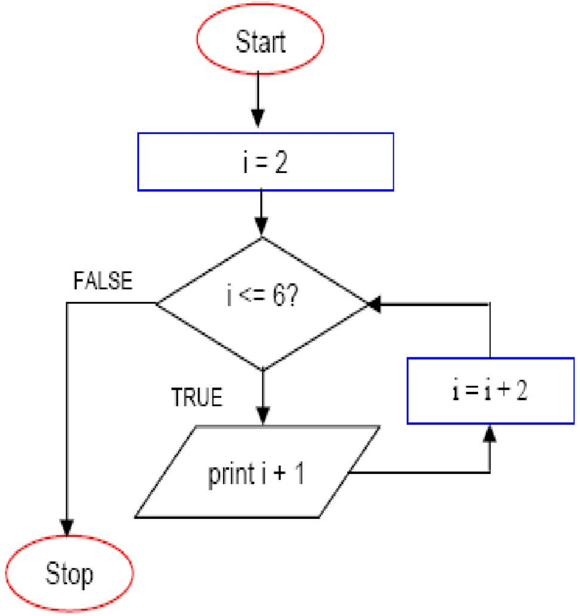
- A finite set of statements that guarantees an optimal solution in finite interval of time

- Algorithmic thinking and problem solving skill are vital in making efficient solutions.

- The English word "ALGORITHM" derives from the Latin word AL-AL-KHWARIZMI’S name. He developed the concept of an algorithm in Mathematics, and thus sometimes being called the “Grandfather of Computer Science".

Glance of Algorithm

- An algorithm is a finite set of instructions or logic, written in order, to accomplish a certain predefined task.
- Algorithm is not the complete code or program
- Can be expressed either as an informal high level description as pseudocode or using a flowchart.



WHILE loop

- Do the loop body if the condition is true.
- Example: Get the sum of 1, 2, 3, …, 100.
  - Algorithm:
    - Set the number = 1
    - Set the total = 0
    - While (number <= 100)
      - total = total + number
      - number = number + 1
    - End While
    - Display total

Algorithm Specifications

- *Input* - Every Algorithm must take zero or more number of input values from external.
- *Output* - Every Algorithm must produce an output as result.
- *Definiteness* - Every statement/instruction in an algorithm must be clear and unambiguous (only one interpretation)
- *Finiteness* - For all different cases, the algorithm must produce result within a finite number of steps.
- *Effectiveness* - Every Instruction must be basic enough to be carried out and it also must be feasible.

Good Algorithms?

- Run in less time

- Consume less memory

But computational resources (time complexity) usually important

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

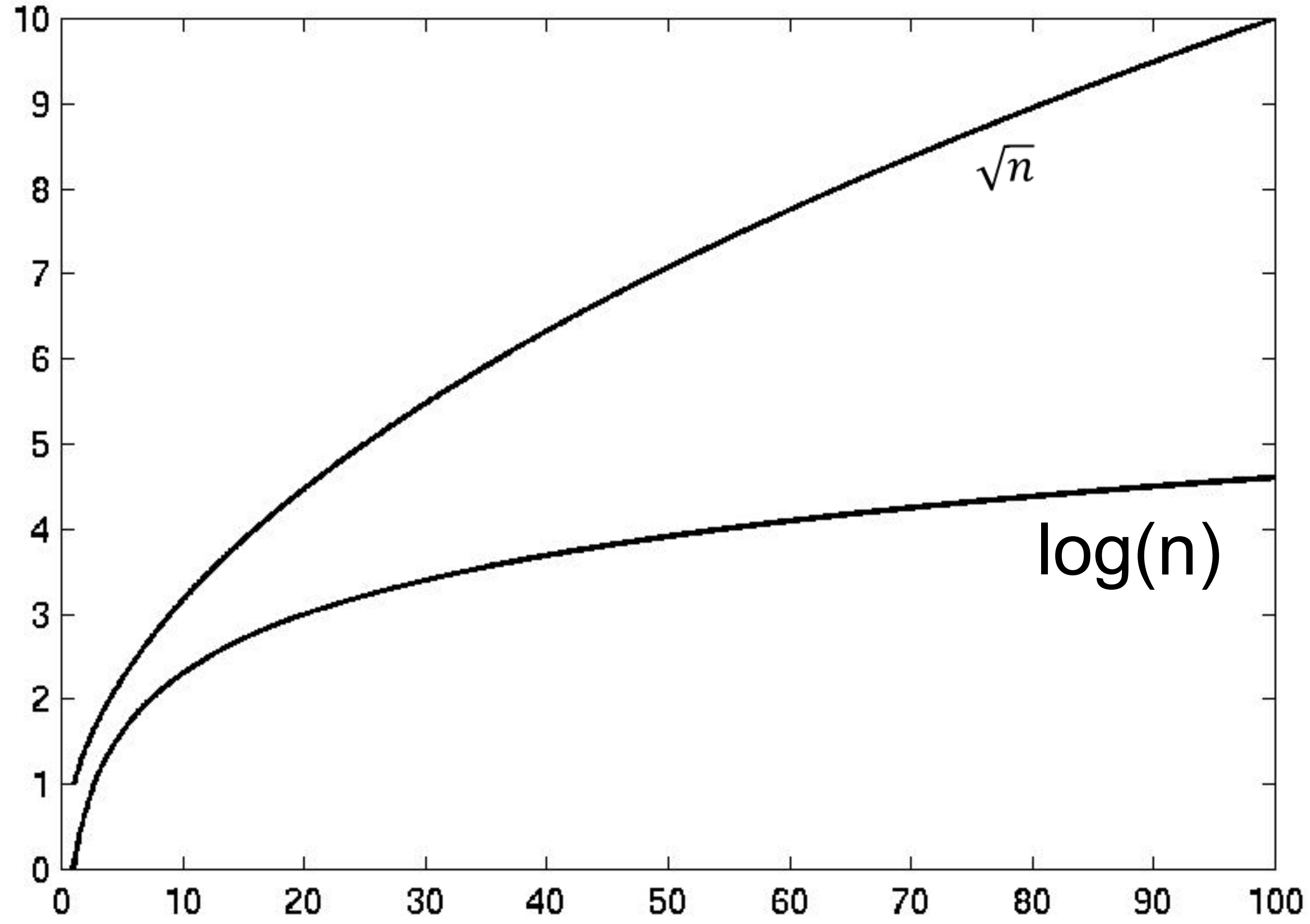
Typical Running Time Functions

- 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

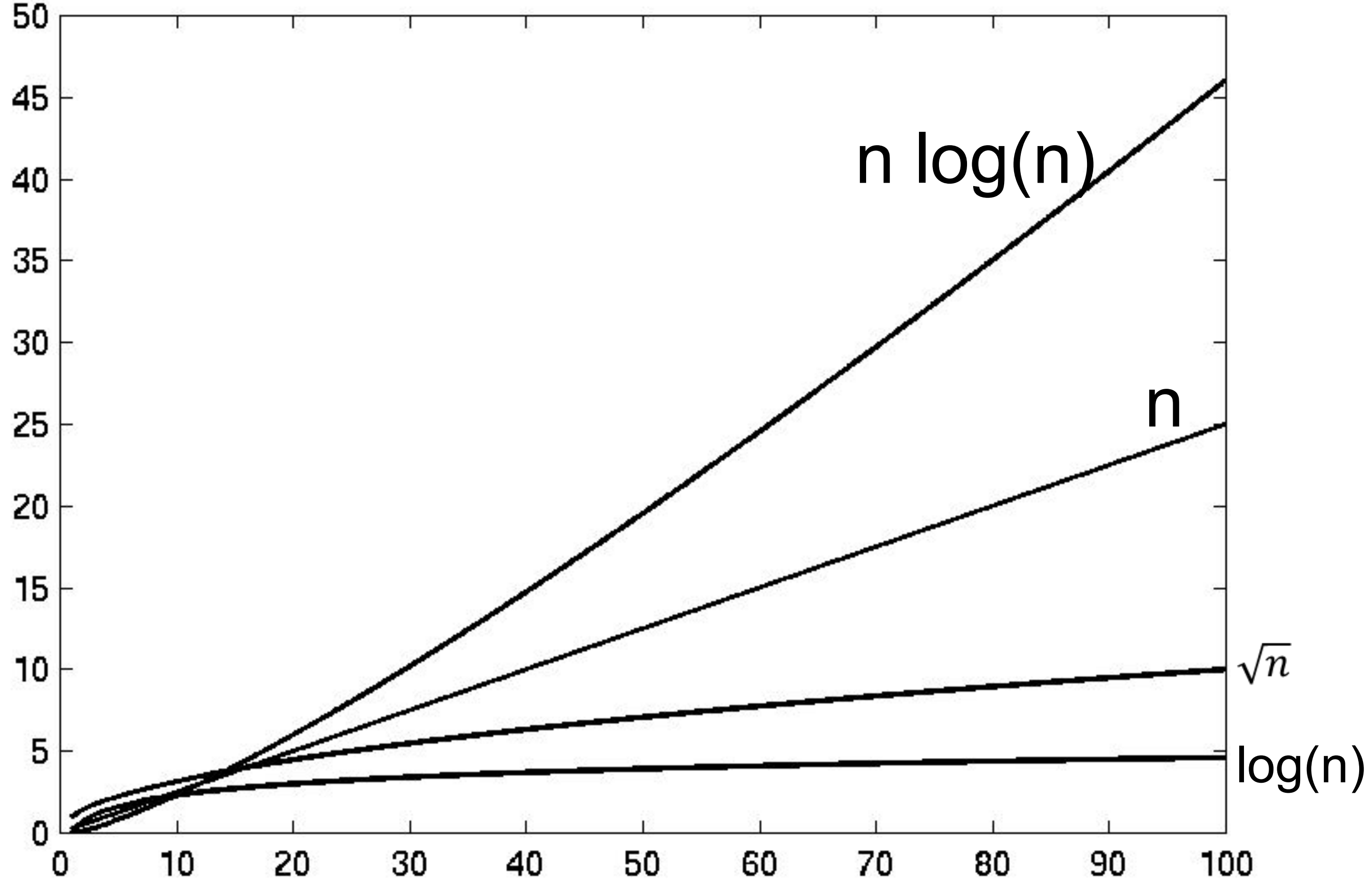
Growth of Functions

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}

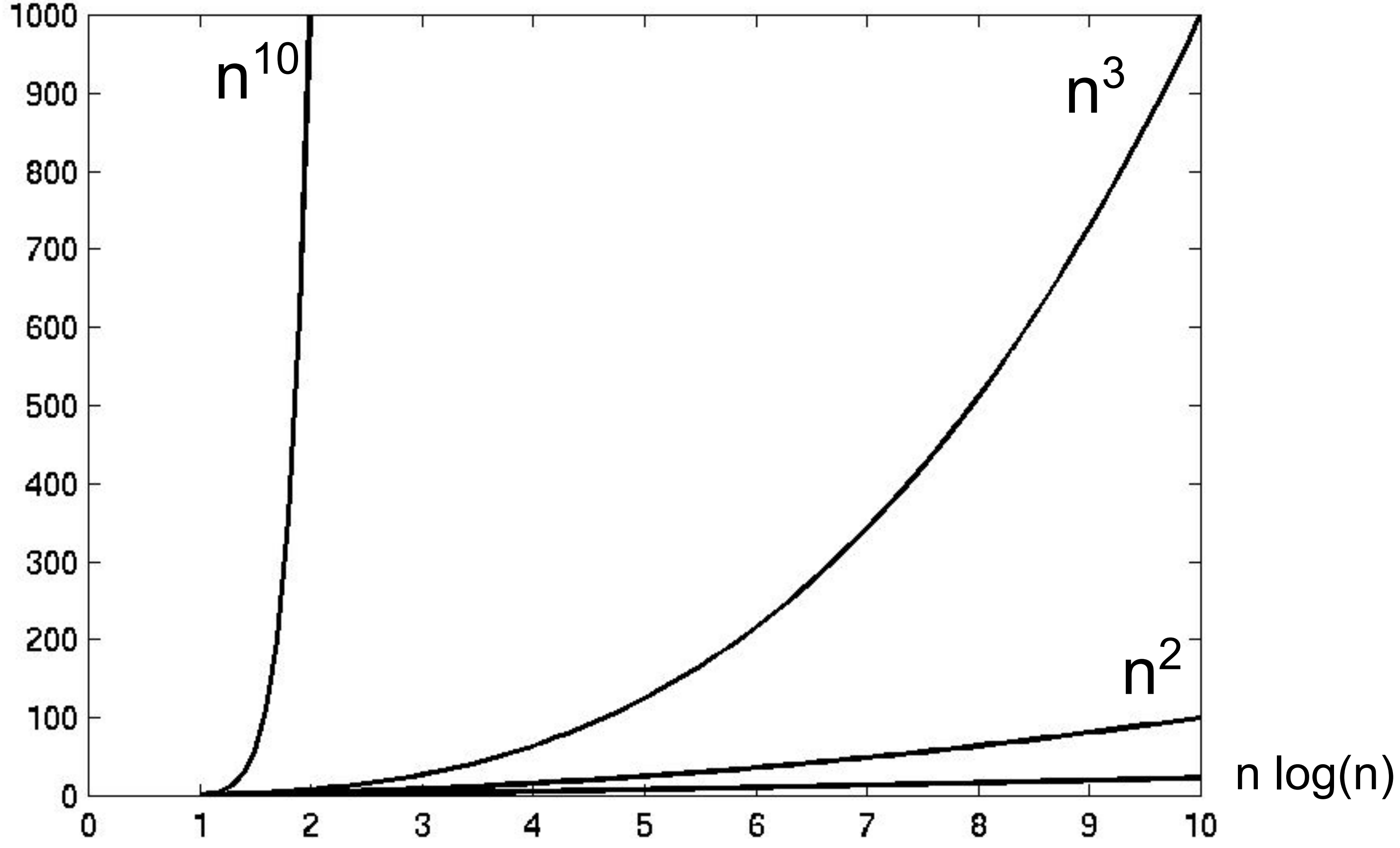
Complexity Graphs



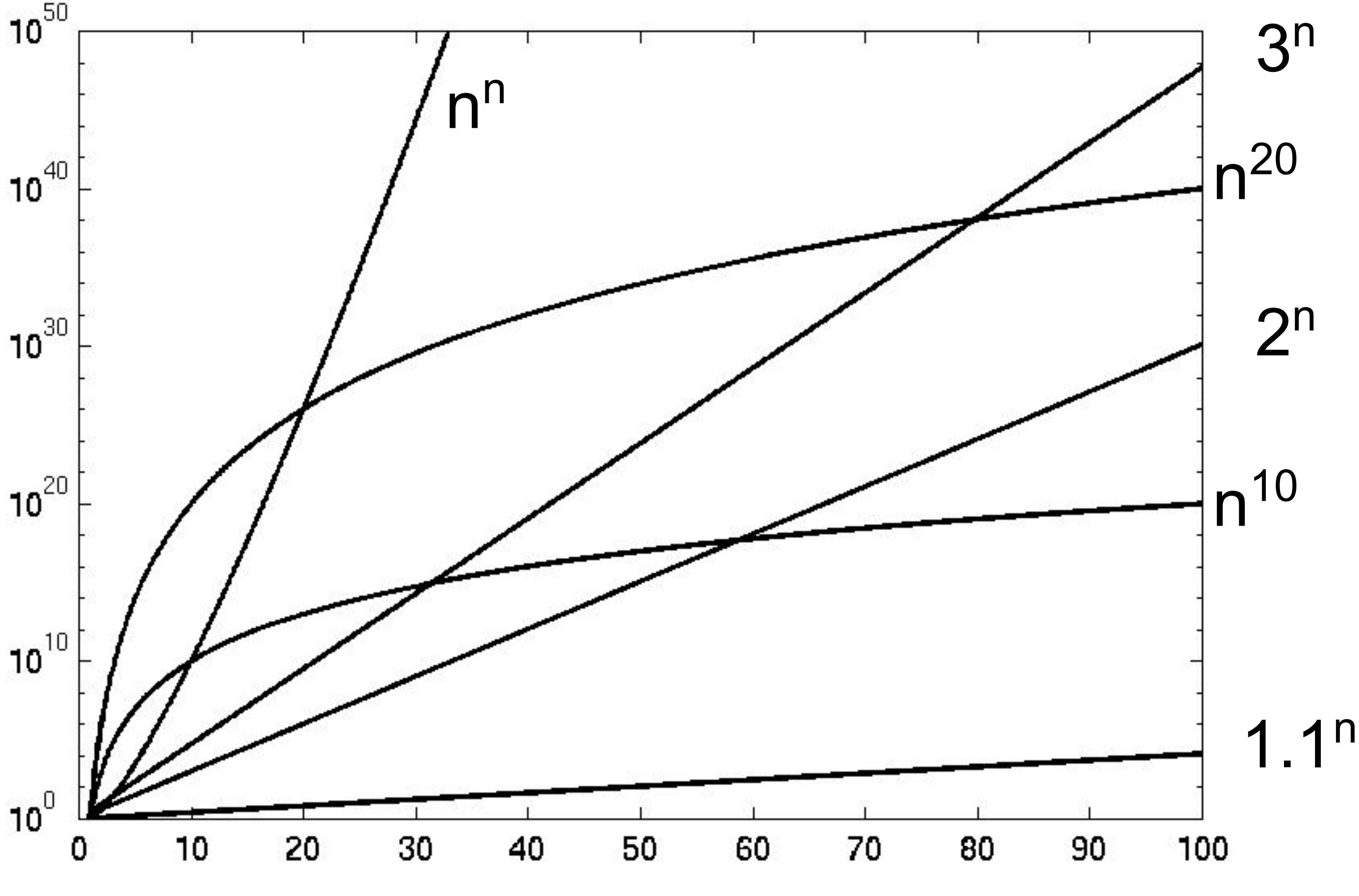
Complexity Graphs



Complexity Graphs



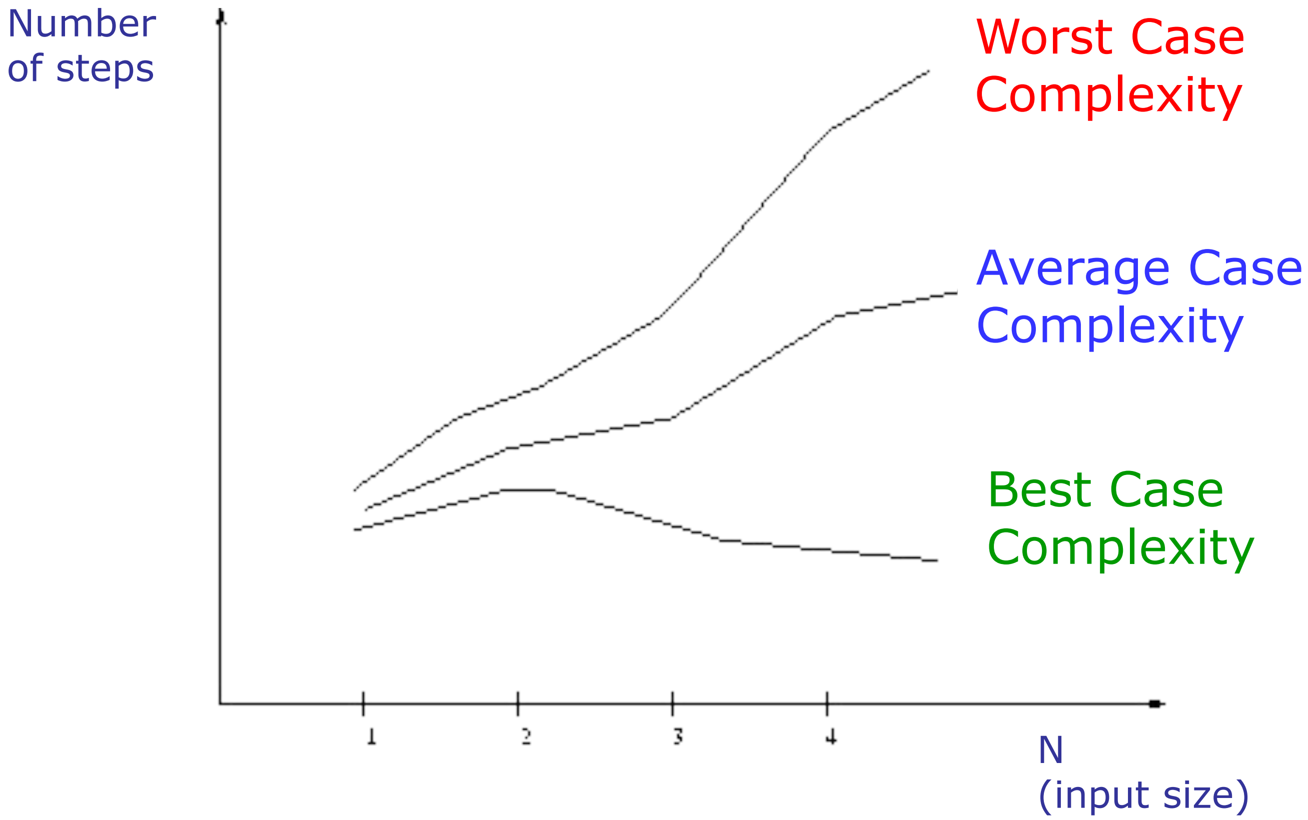
Complexity Graphs (log scale)



Algorithm Complexity

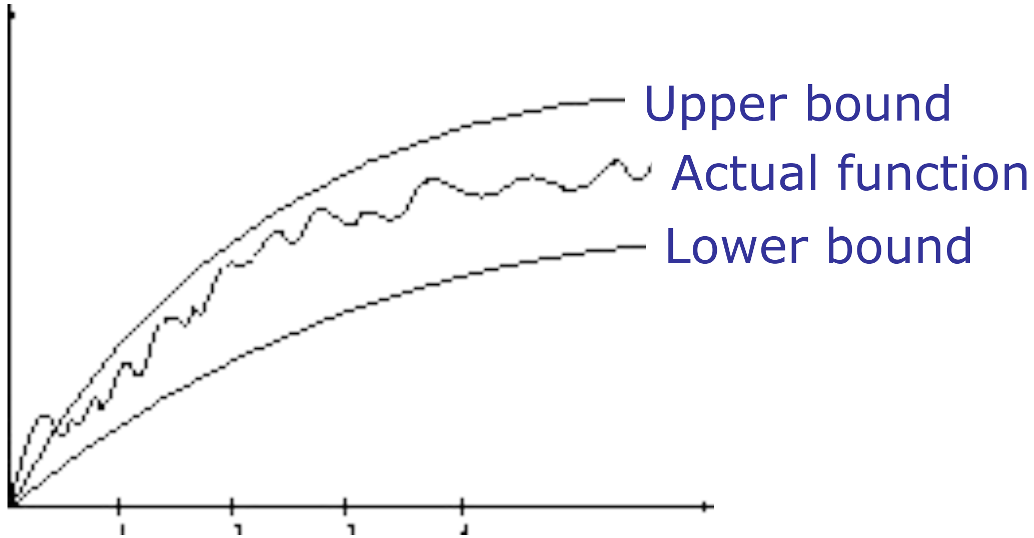
- Worst Case Complexity:
  - the function defined by the *maximum* number of steps taken on any instance of size *n*
- Best Case Complexity:
  - the function defined by the *minimum* number of steps taken on any instance of size *n*
- Average Case Complexity:
  - the function defined by the *average*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

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

Classifying functions by their Asymptotic Growth Rates

- 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 $\Theta(g)$, big omega $\Omega(g)$

Classifying functions by their Asymptotic Growth Rates

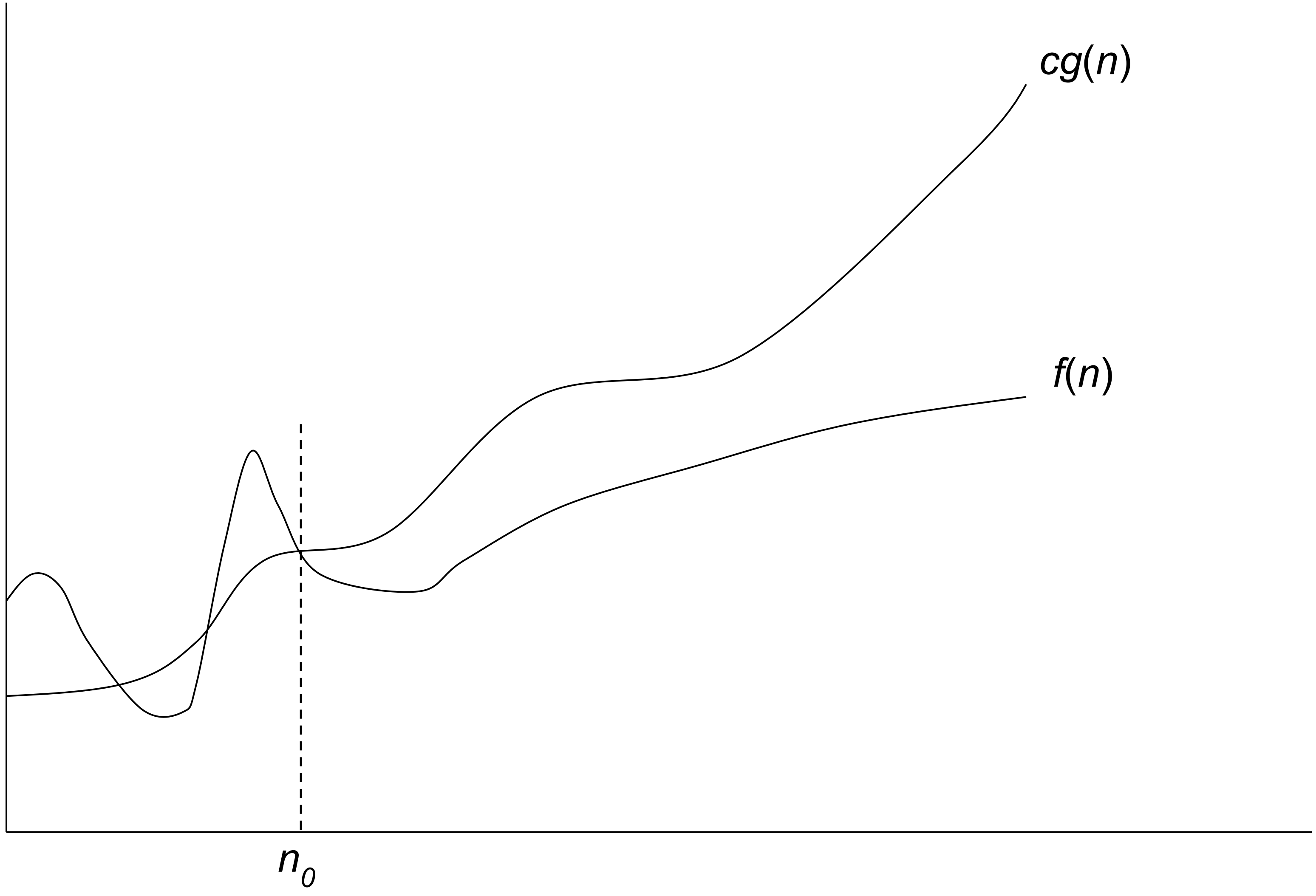
1. $O(g(n))$, Big-Oh of g of n, the Asymptotic Upper Bound
2. $\Theta(g(n))$, Theta of g of n, the Asymptotic Tight Bound
3. $\Omega(g(n))$, Omega of g of n, the Asymptotic Lower Bound

Big-O

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

- 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 $\boldsymbol{n_0}$: $f(n) < cn^2$
    - 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

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

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

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

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

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

More Big-O

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

- Prove that:
- 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 $O(n^2)$

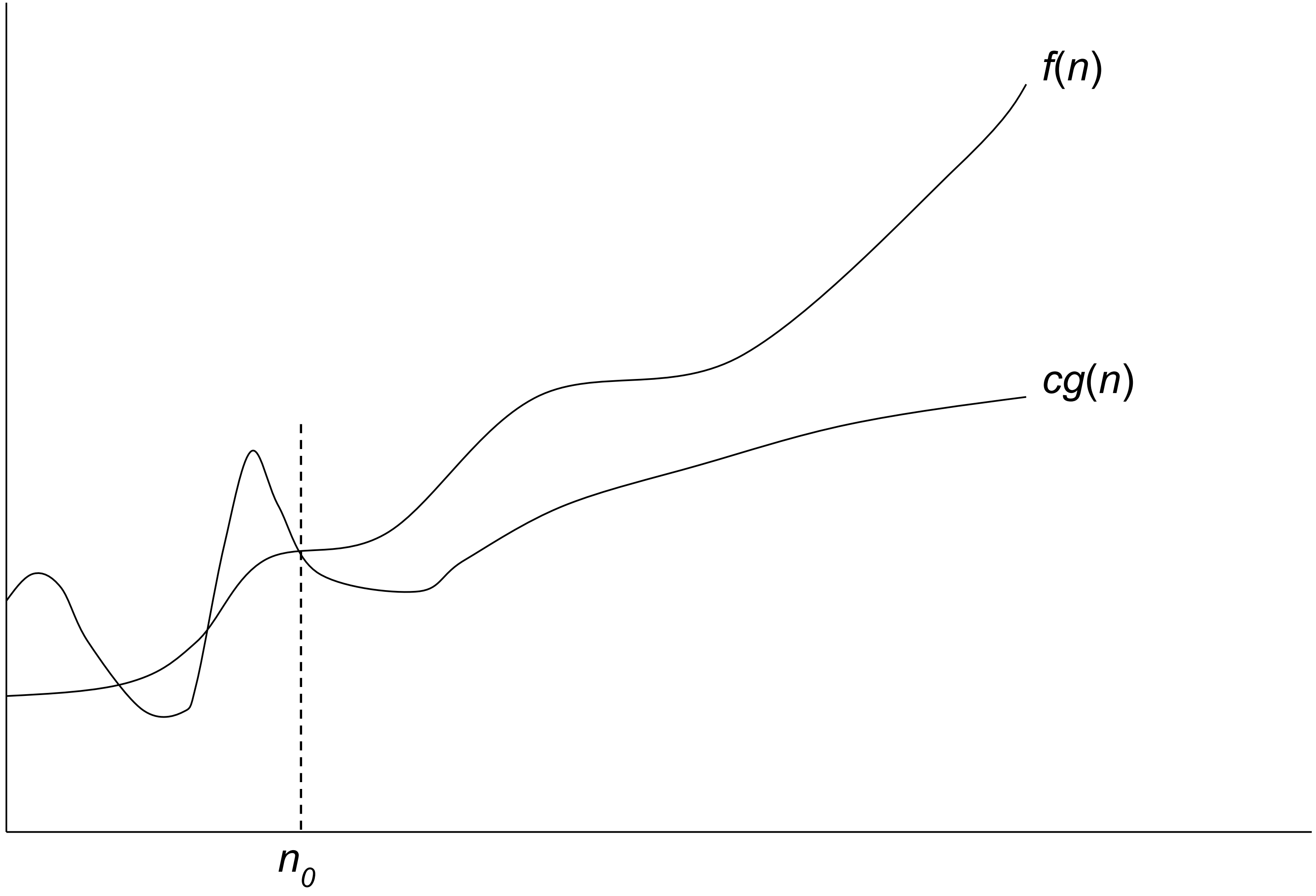
Big Omega – Notation

- $\Omega()$ – A **lower** bound

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



Θ-notation

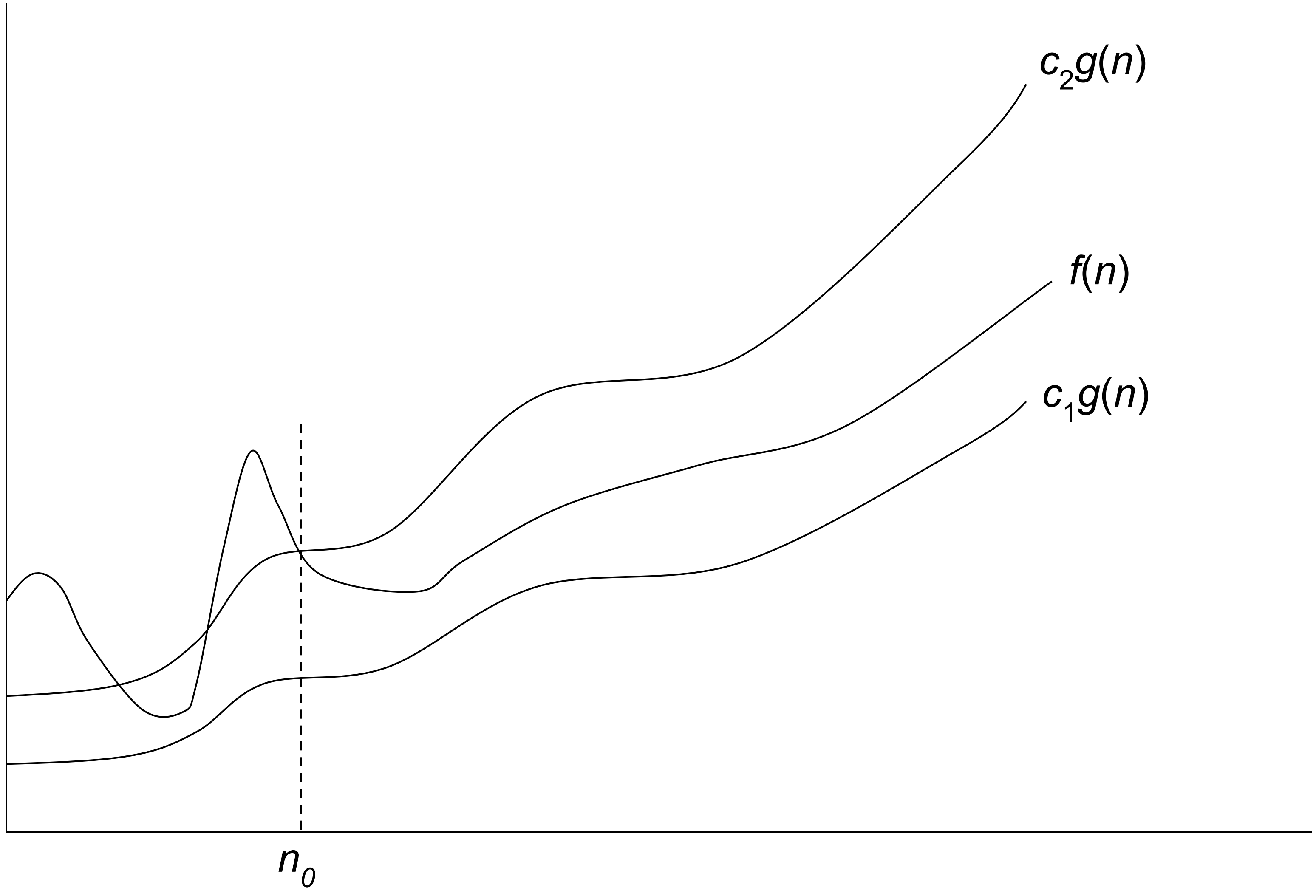
- Big-*O* is not a tight upper bound. In other words $n = O(n^2)$
- Θ provides a tight bound

$f(n) = \Theta(g(n))$: there exist positive constants c_1, c_2, and n_0 such that
$0 \le c_1 g(n) \le f(n) \le c_2 g(n)$ 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 $\Theta(g(n))$



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: $$20n^3 + 7n + 1000 = \Theta(n^3)$$

- 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 $n_0 = 5$

$$2 \times 2^n > 2^n + n^2$$
$$2^{n+1} > 2^n + n^2$$
$$2^{n+1} - 2^n > n^2$$
$$2^n(2-1) > n^2$$
$$2^n > n^2 \quad \forall n \geq 5$$

✔

Asymptotic Notations - Examples

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

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 Simplified Rules

- $O(1) = c$, where c is a constant
- $O(n) = c*n = cn$, where c is constant and n is variable
- $c_1*O(1) = c_1*c = c_2 = O(1)$, where c, c_1, c_2 are constants
  - $O(1) + O(1) + O(1) = 3*O(1) = O(1)$
  - $5*O(1) = O(1)$
- $n*O(1) = n*c = cn = O(n)$, where c is constant and n is variable
- $O(m) + O(n) \neq O(m+n)$
- $O(m) * O(n) = c_1mc_2n = (c_1*c_2)(mn) = (c_2)(mn) = O(mn)$
- $O(m)*O(n)*O(p)*O(q) = O(m(n(p(q)))) = O(mnpq)$
  - Example nested for loops
- $O(an^2 + bn + c) = O(n^2)$ where a, b , c are constants

Example #1: carry n books from one bookshelf to another one

- How many operations?
- n pick-ups, n forward moves, n drops and n reverse moves 4 n operations
- 4n operations = c. n = O(c. n) = O(n)
- Similarly, any program that reads n inputs from the user will have minimum time complexity O(n).

Example #2: Locating Roll-Number record in Attendance Sheet

What is the time complexity of search?

- Binary Search algorithm at work
  - O(log n)
- Sequential search?
  - O(n)

Example #3: Teacher of CSE 221 gives gifts to first 10 students

- There are n students in the queue.
- Teacher brings one gift at a time.
- Time complexity = O(c. 10) = O(1)
- Teacher will take exactly same time irrespective of the line length.

Loops with Break

```
for (j = 0; j < n; ++j) {
 // 3 atomics
 if (condition) break;
}
```

- Upper bound = $O(4n) = O(n)$
- Lower bound = $\Omega(4) = \Omega(1)$
- Complexity = $O(n)$

Ques: Why don’t we have a $\Theta(\ldots)$ notation here?

Sequential Search

- Given an unsorted vector/list a[ ], find the location of element X.

```
for (i = 0; i < n; i++) {
        if (a[i] == X) return true;
}
return false;
```

- Input size: n = array size()
- Complexity = O(n)

If-then-else Statement

```
if(condition)
    i = 0;
else
    for ( j = 0; j < n; j++)
        a[j] = j;
```

- Complexity = ??

= O(1) + max (O(1), O(N))

= O(1) + O(N)

= O(N)

Consecutive Statements

```
for (j = 0; j < n; ++j) {
    // 3 atomics
}
for (j = 0; j < n; ++j) {
    // 5 atomics
}
```

- Add the complexity of consecutive statements
- Complexity = $O(3n + 5n) = O(n)$

Nested Loop Statements

- Analyze such statements inside out

```
for (j = 0; j < n; ++j) {
   // 2 atomics
   for (k = 0; k < n; ++k) {
      // 3 atomics
   }
}
```

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

Example

- Code:
- `a = b;`

- Complexity:

Example

- Code:
- `sum = 0;`
- `for (i=1; i <=n; i++)`
- `    sum += n;`

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

- Complexity:

Example

- Code:

```
sum = 0;
for (j=1; j<=n; j++)
    for (i=1; i<=j; i++)
        sum++;
for (k=0; k<n; k++)
    A[k] = k;
```

- Complexity:

Example

- Code:
- `sum1 = 0;`
- `for (i=1; i<=n; i++)`
- `    for (j=1; j<=n; j++)`
- `        sum1++;`
- Complexity:

Example

- Code:
- `sum2 = 0;`
- `for (i=1; i<=n; i++)`
- `    for (j=1; j<=i; j++)`
- `        sum2++;`
- Complexity:

Example

- Code:
- ` sum1 = 0;`
- ` for (k=1; k<=n; k*=2)`
- `    for (j=1; j<=n; j++)`
- `       sum1++;`
- Complexity:

Example

- Code:
- `sum2 = 0;`
- `for (k=1; k<=n; k*=2)`
- `    for (j=1; j<=k; j++)`
- `        sum2++;`
- Complexity:

Recursion

```
long factorial( int n )
{
   if( n <= 1 )
     return 1;
   else
     return n*factorial(n- 1);
}
```

In terms of big-Oh:
$t(1) = 1$
$t(n) = 1 + t(n-1) = 1 + 1 + t(n-2)$
$= \ldots k + t(n-k)$
Choose $k = n-1$
$t(n) = n-1 + t(1) = n-1 + 1 =$
$O(n)$

Consider the following time complexity:
$t(0) = 1$
$t(n) = 1 + 2t(n-1) = 1 + 2(1 + 2t(n-2)) = 1 + 2 + 4t(n-2)$
$= 1 + 2 + 4(1 + 2t(n-3)) = 1 + 2 + 4 + 8t(n-3)$
$= 1 + 2 + \ldots + 2^{k-1} + 2^k t(n-k)$
Choose $k = n$
$t(n) - 1 + 2 + \ldots 2^{n-1} + 2^n = 2^{n+1} - 1$

Binary Search

- Given a sorted vector/list a[ ], find the location of element X

```
unsigned int binary_search(vector<int> a, int X)
{
    unsigned int low = 0, high = a.size()-1;

    while (low <= high) {
        int mid = (low + high) / 2;
        if (a[mid] < X)
            low = mid + 1;
        else if( a[mid] > X )
            high = mid - 1;
        else
            return mid;
    }
    return NOT_FOUND;
}
```

- Input size: n = array size()
- Complexity = O(k iterations x (1 comparison+1 assignment) per loop)

  = O(log(n))

Summary

- Time complexity is a measure of algorithm efficiency
- Efficient algorithm plays the major role in determining the running time.

Q: Is it possible to determine running time based on algorithm's time complexity alone?

- Minor tweaks in the code can cut down the running time by a factor too.
- Other items like CPU speed, memory speed, device I/O speed can help as well.
- For certain problems, it is possible to allocate additional space & improve time complexity.

Summary

- Time complexity is a measure of algorithm efficiency
- Efficient algorithm plays the major role in determining the running time.

Q: Is it possible to determine running time based on algorithm's time complexity alone?

- Minor tweaks in the code can cut down the running time by a factor too.
- Other items like CPU speed, memory speed, device I/O speed can help as well.
- For certain problems, it is possible to allocate additional space & improve time complexity.