Basics of Algorithm

Kinds of Analyses

- Worst case
  - Provides an upper bound on running time
  - An absolute guarantee
- Best case – not very useful
- Average case
  - Provides the expected running time
  - Very useful, but treat with care: what is “average”?
    - Random (equally likely) inputs
    - Real-life inputs

How to measure complexity?

- Accurate running time is not a good measure.
- It depends on the machine you used.
- It depends on input.

Machine-independent

- A generic uniprocessor random-access machine (RAM) model
  - No concurrent operations
  - Each **simple** operation (e.g. +, -, =, *, if, for) takes 1 step.
    - **Loops** and **subroutine** calls are ***not*** simple operations.
  - All memory equally expensive to access
    - Constant word size
    - Unless we are explicitly manipulating bits

Asymptotic Analysis

- How does algorithm behave as the problem size gets very large?

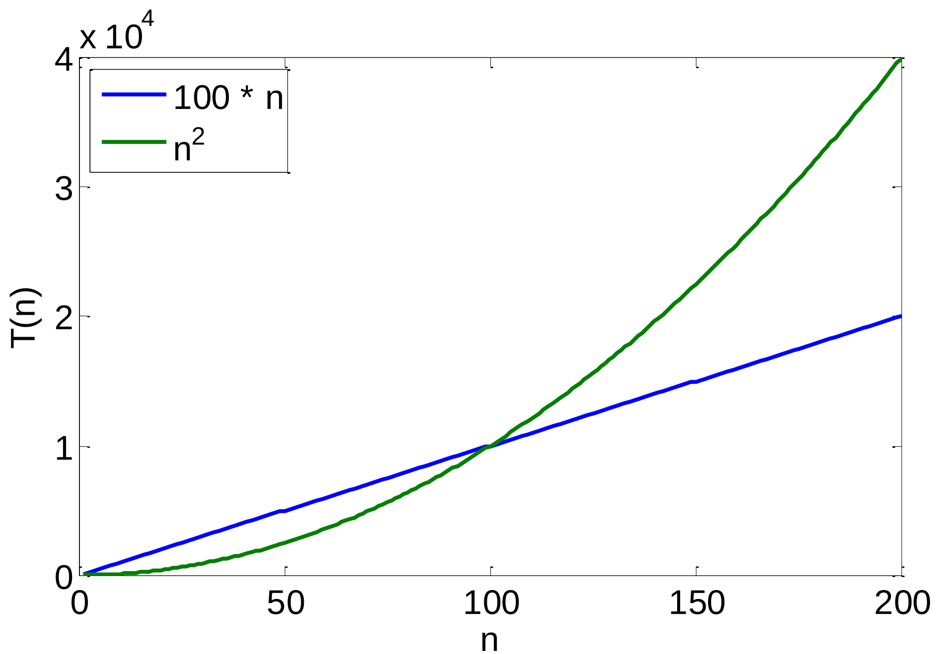
- Running time depends on the size of the input
  - Larger array takes more time to sort
  - To compare two algorithms with running times *f(n)* and *g(n),* we need a **rough measure** that characterizes **how fast each function grows**
  - Look at ***growth*** of $T(n)$ as $n \to \infty$.

Asymptotic Analysis

- Order of Growth
  - The low order terms in a function are relatively insignificant for **large** *n*

$$n^4 + 100n^2 + 10n + 50 \quad \sim \quad n^4$$

i.e., we say that $n^4 + 100n^2 + 10n + 50$ and n^4 have the same **order of growth**



Input Size

- 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

Example

- Associate a "cost" with each statement.
- Find the "total cost“ by finding the total number of times each statement is executed.

Algorithm 1

	Cost
arr[0] = 0;	c_1
arr[1] = 0;	c_1
arr[2] = 0;	c_1
...	...
arr[N-1] = 0;	c_1

$c_1+c_1+...+c_1 = c_1 \times N$

Algorithm 2

	Cost
for(i=0; i<N; i++)	c_2
arr[i] = 0;	c_1

$(N+1) \times c_2 + N \times c_1 = (c_2 + c_1) \times N + c_2$

Example

Algorithm 3	*Cost*
`sum = 0;`	c_1
`for(i=0; i<N; i++)`	c_2
`  for(j=0; j<N; j++)`	c_2
`  sum += arr[i][j];`	c_3

$c_1 + c_2 \text{ x } (N+1) + c_2 \; x \; N \text{ x } (N+1) + c_3 \text{ x } N^2$

Order of growth

$1 << \log_2 n << n << n\log_2 n << n^2 << n^3 << 2^n << n!$

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}

Asymptotic Notation

- **O notation:** asymptotic "less than":
  - f(n) is O(g(n)) if f(n) is asymptotically **less than or equal** to g(n)
- **Ω notation:** asymptotic "greater than":
  - f(n) is Ω(g(n)) if f(n) is asymptotically **greater than or equal** to g(n)
- **Θ notation:** asymptotic "equality":
  - f(n) is Θ(g(n)) if f(n) is asymptotically **equal** to g(n)

Big O

- Informally, O (g(n)) is the set of all functions with a smaller or same order of growth as g(n), within a constant multiple
- If we say f(n) is in O(g(n)), it means that g(n) is an asymptotic upper bound of f(n)
  - Intuitively, it is like $f(n) \le g(n)$
- What is $O(n^2)$?
  - The set of all functions that grow slower than or in the same order as n^2

- For example

$n \in O(n^2)$
$n^2 \in O(n^2)$
$1000n \in O(n^2)$
$n^2 + n \in O(n^2)$
$100n^2 + n \in O(n^2)$

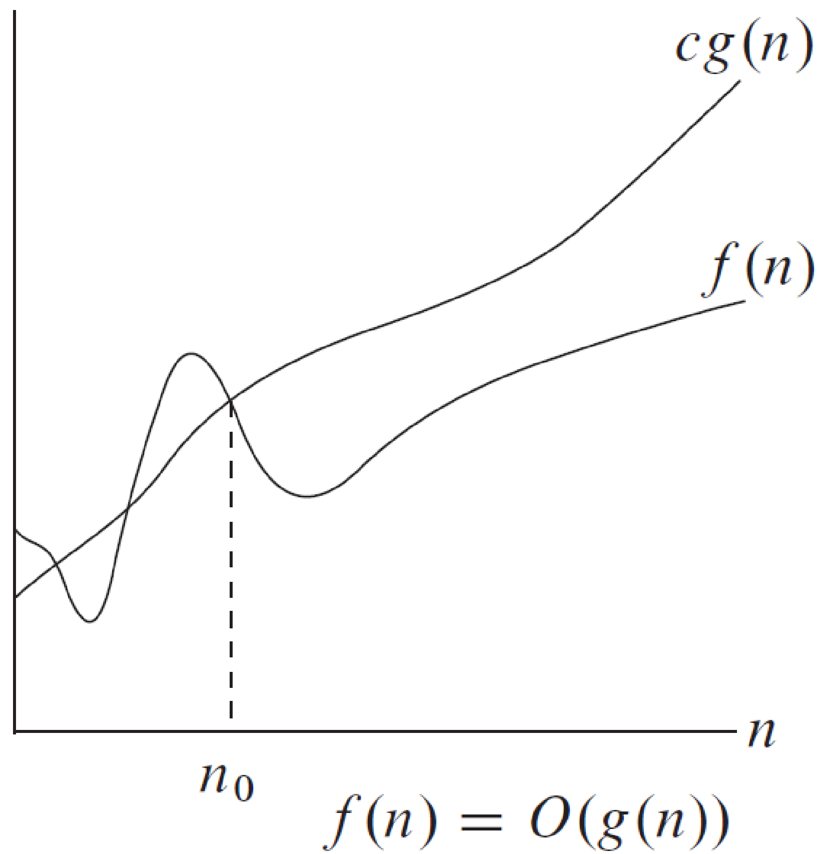
But: $1/1000\ n^3 \notin O(n^2)$

Big-O

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

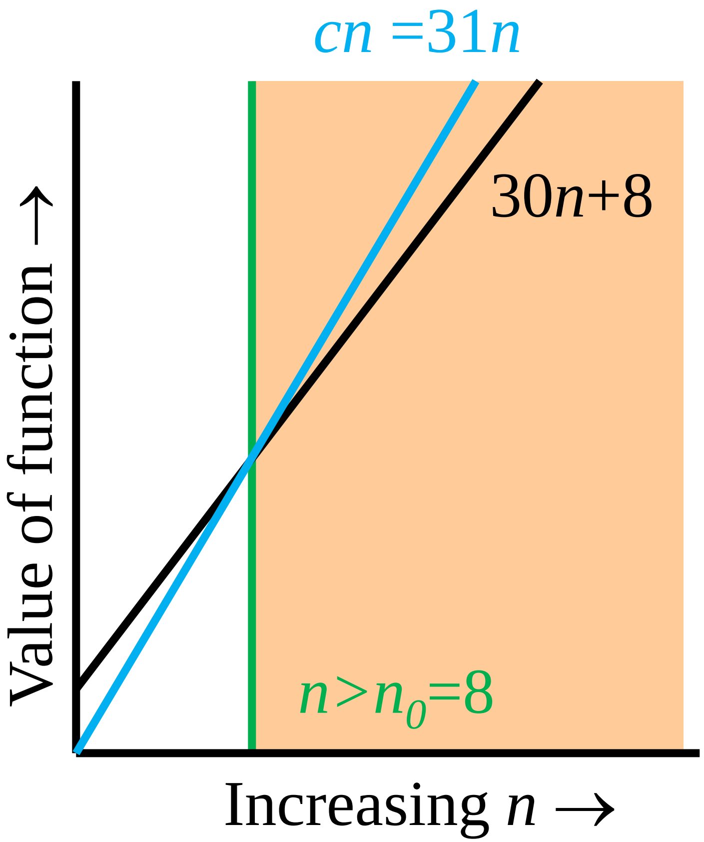
- What does it mean?
  - If $f(n) = O(n^2)$, then $f(n)$ can be larger than n^2 sometimes, **but…**
    - We can choose some constant c and some value n_0 such that for **every** value of n larger than n_0 : $f(n) \leq cn^2$
    - That is, for values larger than n_0, $f(n)$ is never more than a constant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

Big-O Visualization



Examples

- Show that $30n+8$ is $O(n)$.
  - Show $\exists c, n_0$: $30n+8 \leq cn$, $\forall n \geq n_0$.
  - Let $c=31$, $n_0=8$
    $cn = 31n = 30n + n \geq 30n+8$,
  - so $30n+8 \leq cn$.



$30n+8 \in O(n)$

Back to Example

Algorithm 1		*Algorithm 2*	
	Cost		**Cost**
arr[0] = 0;	c_1	for(i=0; i<N; i++)	c_2
arr[1] = 0;	c_1	arr[i] = 0;	c_1
arr[2] = 0;	c_1		
...	...		
arr[N-1] = 0;	c_1		
	-----------		------------
	$c_1+c_1+...+c_1 = c_1 \times N$		$(N+1) \times c_2 + N \times c_1 = (c_2 + c_1) \times N + c_2$

Both algorithms are of the same order: *O(N)*

Back to Example

Algorithm 3	*Cost*
sum = 0;	c_1
for(i=0; i<N; i++)	c_2
for(j=0; j<N; j++)	c_2
sum += arr[i][j];	c_3

$c_1 + c_2 \text{ x } (N{+}1) + c_2 \ x\ N \text{ x } (N{+}1) + c_3 \text{ x } N^2$

This algorithm is of the order $O(N^2)$.

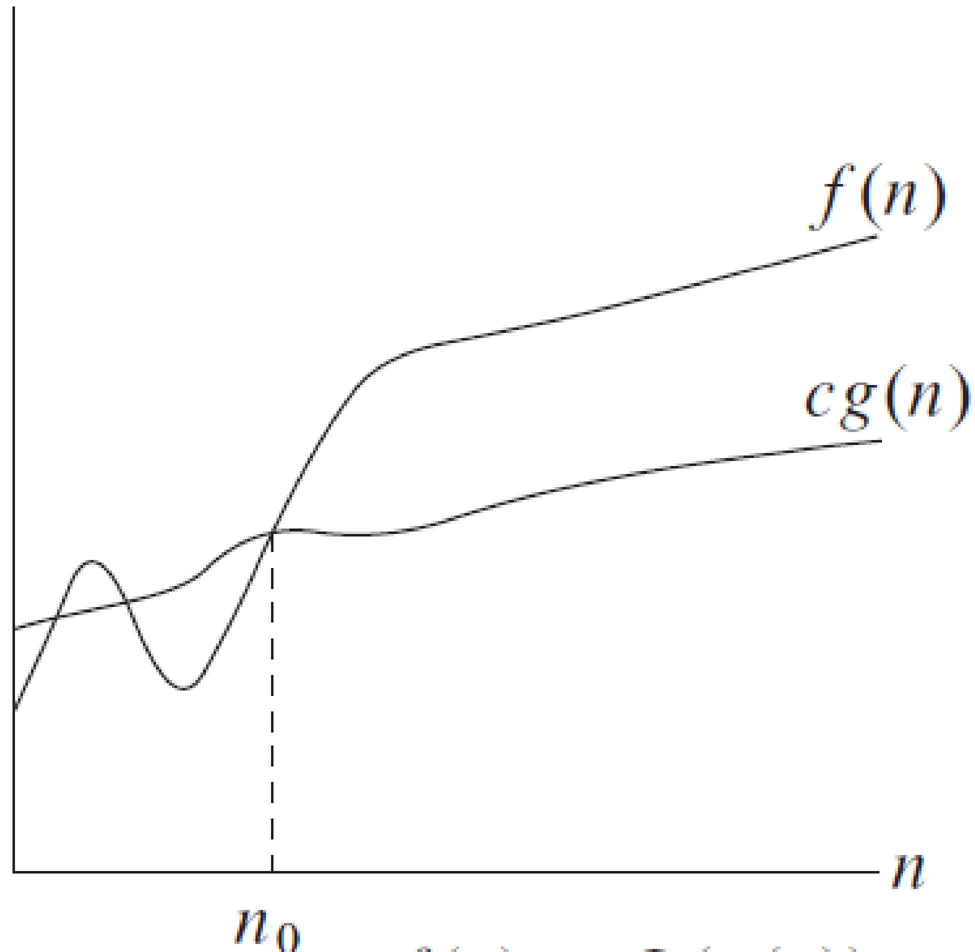
Big Omega – Notation

- $\Omega()$ – A **lower** bound

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

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

Big Omega Visualization



$f(n) = \Omega(g(n))$

Θ-notation

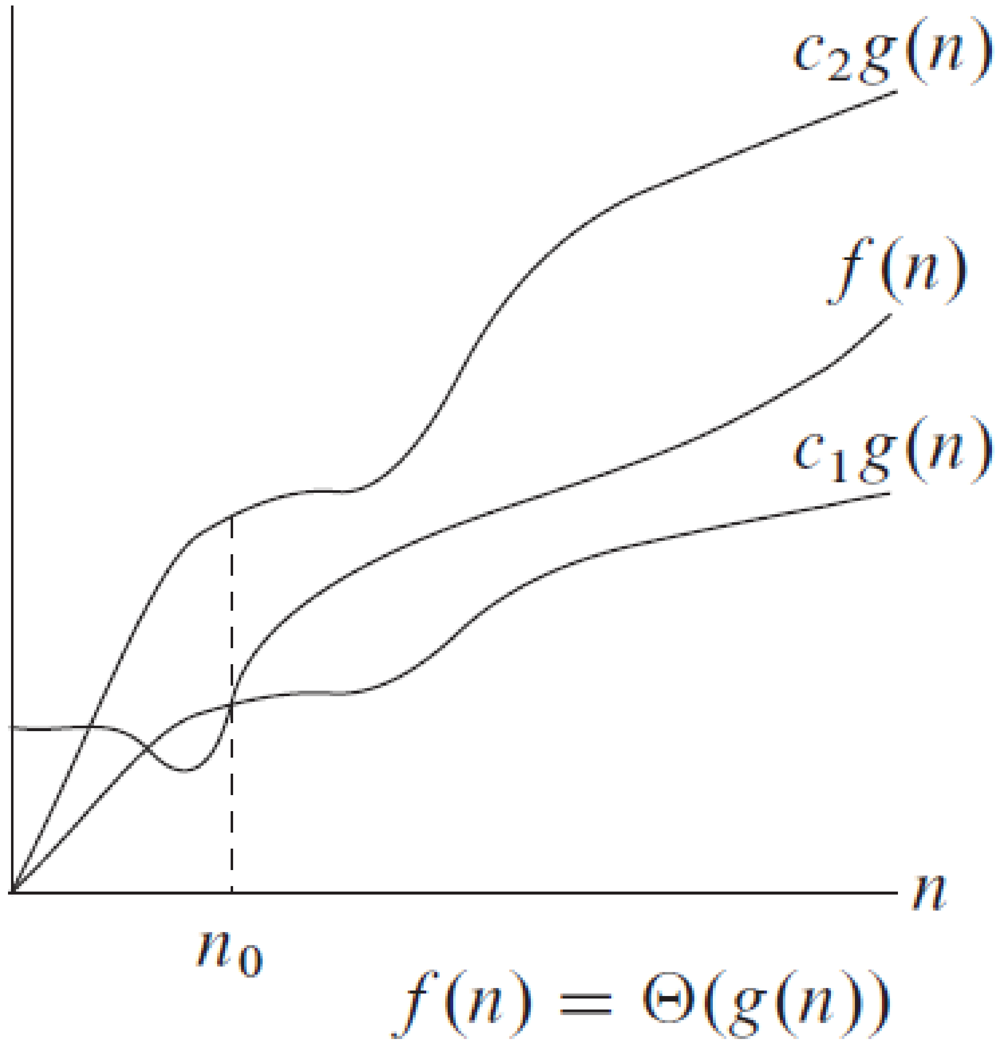
- Big-O is not a tight upper bound. In other words $n = O(n^2)$
- Θ provides a tight bound

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

- In other words,

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

Θ Visualization



Example

- Prove that: $$20n^3 + 7n + 1000 = \Theta\left(n^3\right)$$

- Let $c = 21$ and $n_0 = 10$
- $21n^3 \geq 20n^3 + 7n + 1000$ for all $n > 10$

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

  TRUE, but we also need…
- Let $c = 20$ and $n_0 = 10$
- $20n^3 \leq 20n^3 + 7n + 1000$ for all $n \geq 10$

  TRUE

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 $f_1(n) * f_2(n) = O(g_1(n) * g_2(n))$

Example

- Code:

```
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:

Recursive evaluation of n!

Definition: $n\,! = 1 * 2 * \ldots *(n\text{-}1) * n$ for $n \geq 1$ and $0! = 1$

Recursive definition of n!: $F(n) = F(n\text{-}1) * n$ for $n \geq 1$ and
$F(0) = 1$

Size: n

Basic operation: Multiplication

Recurrence relation: $M(n) = M(n\text{-}1) + 1$

$M(0) = 0$

Solving the recurrence for M(*n*)

$M(n) = M(n\text{-}1) + 1, \;\; M(0) = 0$

$$
\begin{aligned}
M(n) &= M(n-1) + 1 \\
&= (M(n-2) + 1) + 1 \quad = \quad M(n-2) + 2 \\
&= (M(n-3) + 1) + 2 \quad = \quad M(n-3) + 3 \\
&\ldots \\
&= M(n-i) + i \\
&= M(0) + n \\
&= n
\end{aligned}
$$