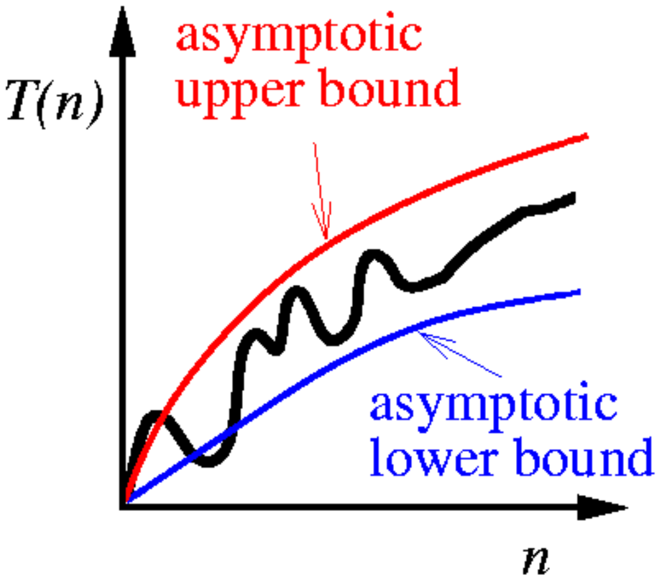
Unit 1: Algorithmic Fundamentals

- Course contents:
  - On algorithms
  - Mathematical foundations
  - Asymptotic notation
  - Growth of functions
  - Recurrences
- Readings:
  - Chapters 1, 2, 3, 4
  - Appendix A

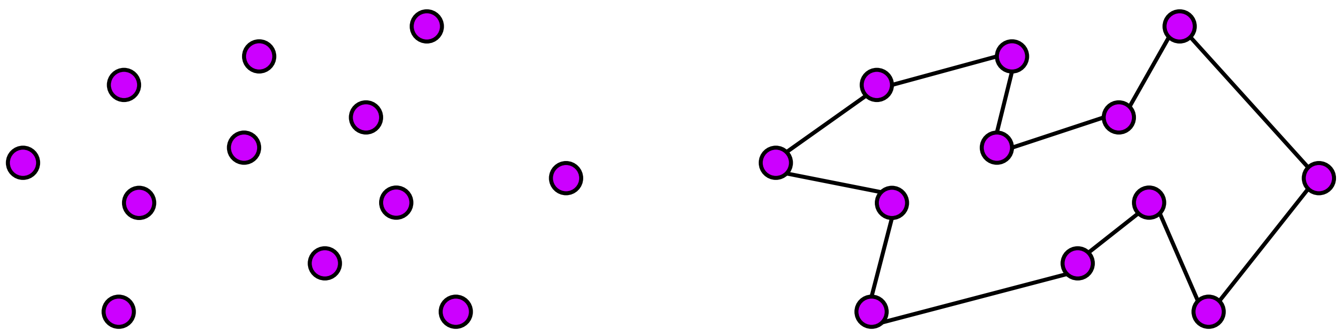


On Algorithms

- **Algorithm:** A well-defined procedure for transforming some **input** to a desired **output**.
- **Major concerns:**
  - **Correctness:** Does it **halt**? Is it **correct**? Is it **stable**?
  - **Efficiency: Time** complexity? **Space** complexity?
    - Worst case? Average case? (Best case?)
- **Better algorithms?**
  - **How: Faster** algorithms? Algorithms with **less space** requirement?
  - **Optimality:** Prove that an algorithm is **best possible/optimal**? Establish a **lower bound**?
- **Applications?**
  - **Everywhere in computing!**

Example: Traveling Salesman Problem (TSP)

- **Input:** A set of points P (cities) together with a distance $d(p, q)$ between any pair $p, q \in P$.
- **Output:** The shortest circular route that starts and ends at a given point and visits all the points.

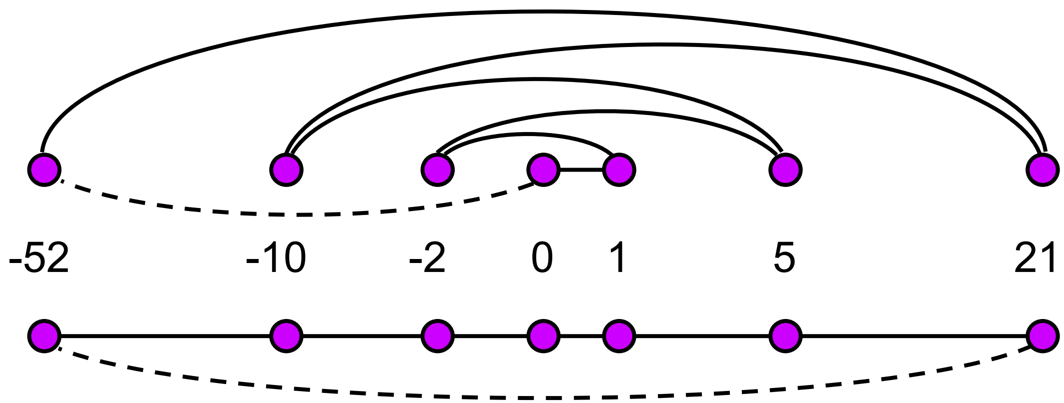


- Correct and efficient algorithms?

Nearest Neighbor Tour

1. pick and visit an initial point p_0
2. $i = 0$
3. **while** there are unvisited points
4. visit p_i's nearest unvisited point p_{i+1}
5. $i = i + 1$
6. return to p_0 from p_i

- Simple to implement and very efficient, but **incorrect!**

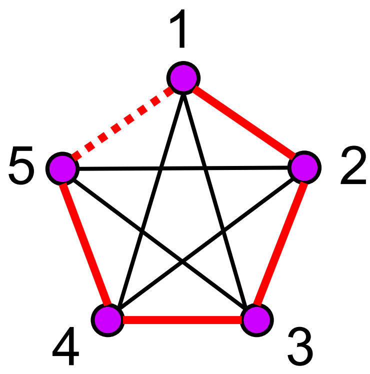


A Correct, But Inefficient Algorithm

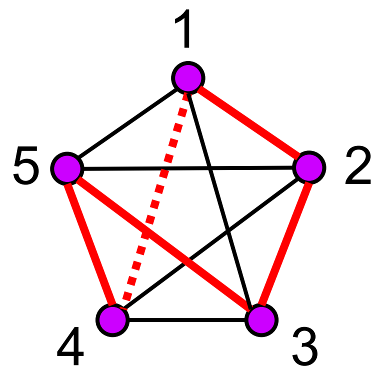
1. $d = \infty$
2. **for** each of the n! permutations π_i of the n points
3.   **if** (cost(π_i) ≤ d)
4.   d = cost(π_i)
5. $T_{min} = \pi_i$
6. **return** T_{min}

- **Correctness?** Tries all possible orderings of the points ⇒ Guarantees to end up with the shortest possible tour.
- **Efficiency?** Tries *n!* possible routes!
  - 120 routes for 5 points, 3,628,800 routes for 10 points, 20 points?
- No known efficient, correct algorithm for TSP!
  - TSP is "**NP-complete**".

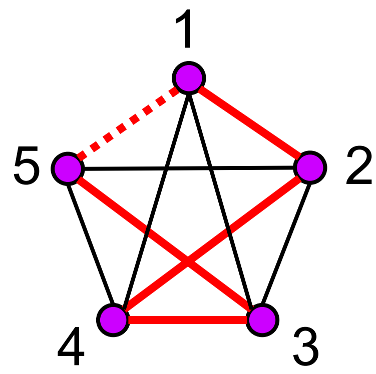
Example of Permutations



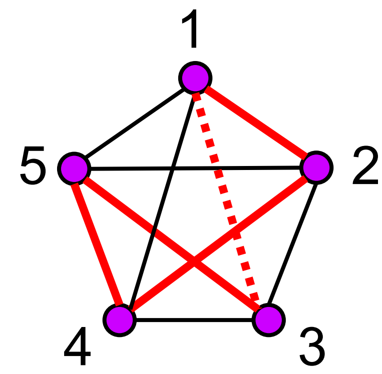
<1,2,3,4,5>



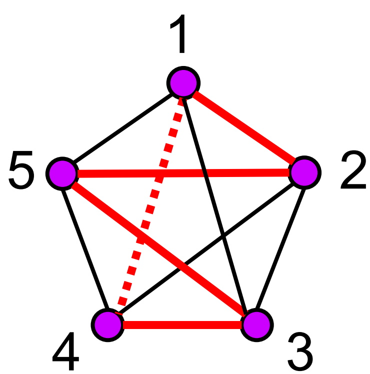
<1,2,3,5,4>



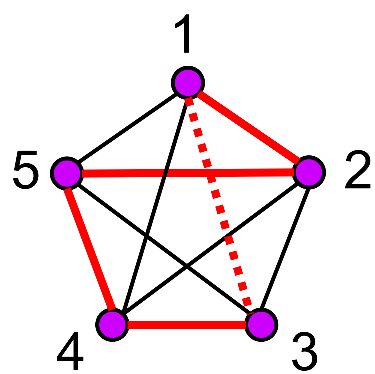
<1,2,4,3,5>



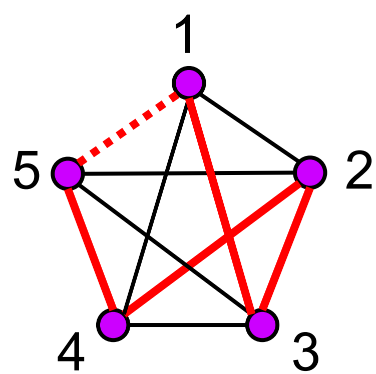
<1,2,4,5,3>



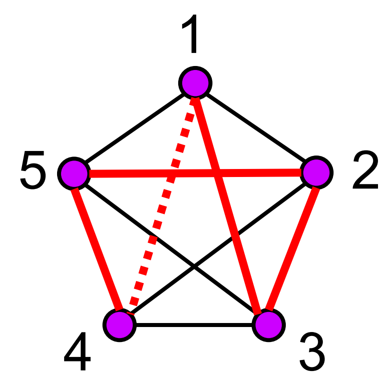
<1,2,5,3,4>



<1,2,5,4,3>



<1,3,2,4,5>



<1,3,2,5,4> …

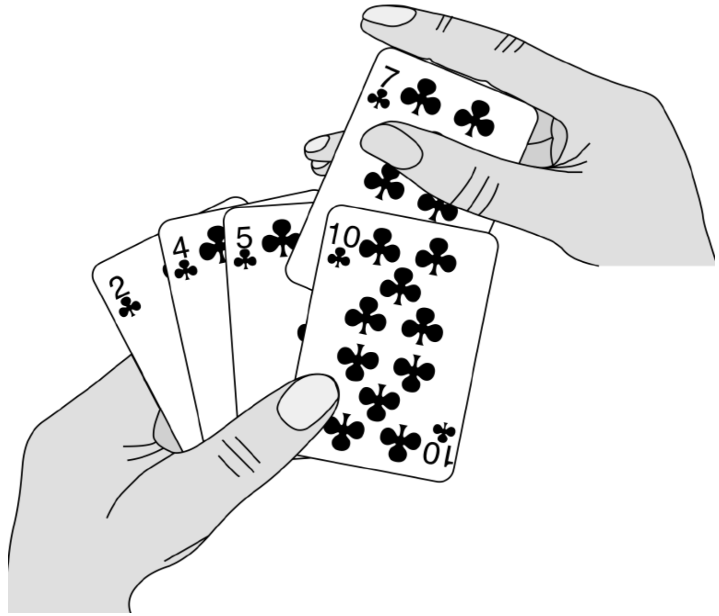
5! = 120 permutations for only 5 points!!

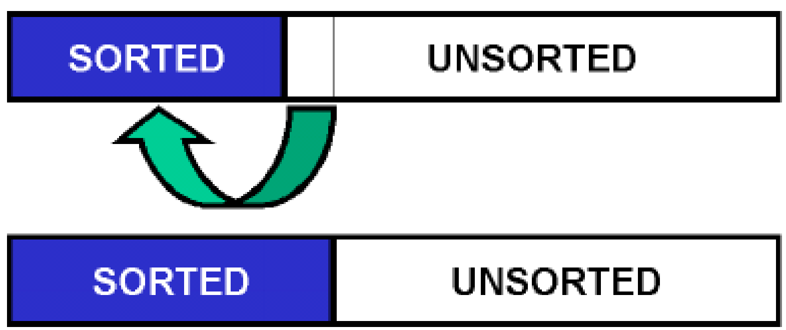
Insertion Sort and Asymptotic Analysis

Sorting

- **Input:** A sequence of n numbers $<a_1, a_2, \ldots, a_n>$.
- **Output:** A permutation $<a_1', a_2', \ldots, a_n'>$ such that $a_1' \leq a_2' \leq \ldots \leq a_n'$.
- Example:
  - Input: <8, 6, 9, 7, 5, 2, 3>
  - Output: <2, 3, 5, 6, 7, 8, 9 >
- Correct and efficient algorithms?

Insertion Sort Illustration





What is the invariant of this sort?

Insertion Sort

$j = 4$
$Key = 7$
$i = 3$
$A[3] = 8$
$i = 1$
$A[2] = 7$

```
InsertionSort(A)
for j = 2 to A.length do
    key = A[j]
    /* Insert A[j] into the sorted sequence A[1..j-1] */
    i = j - 1
    while i > 0 and A[i] > key do
        A[i+1] = A[i]
        i = i - 1
    A[i+1] = key
```

8	6	9	7	5	2
6	8	9	7	5	2
6	8	9	7	5	2
6	7	8	9	5	2
5	6	7	8	9	2
2	5	6	7	8	9

Correctness?

❑ *Loop invariant*

– At the start of each iteration of the for loop of lines 1--8, subarray *A*[1..*j*-1] consists of the elements originally in *A*[1..*j*-1] but in sorted order.

A[1..1]	8	6	9	7	5	2	j = 2
A[1..2]	6	8	9	7	5	2	j = 3
A[1..3]	6	8	9	7	5	2	j = 4
A[1..4]	6	7	8	9	5	2	j = 5
	5	6	7	8	9	2	j = 6
	2	5	6	7	8	9	j = 7

Loop Invariant for Proving Correctness

```
InsertionSort(A)
for j = 2 to A.length do
    key = A[j]
    /* Insert A[j] into the sorted sequence A[1..j-1] */
    i = j - 1
    while i > 0 and A[i] > key do
        A[i+1] = A[i]
        i = i - 1
    A[i+1] = key
```

- **We may use loop invariants to prove the correctness.**
  - **Initialization:** True before the 1st iteration.
  - **Maintenance:** If it is true before an iteration, it remains true before the next iteration.
  - **Termination:** When the loop terminates, the invariant leads to the correctness of the algorithm.

Loop Invariant of Insertion Sort

```
InsertionSort(A)
for j = 2 to A.length do
    key = A[j]
    /* Insert A[j] into the sorted sequence A[1..j-1] */
    i = j - 1
    while i > 0 and A[i] > key do
        A[i+1] = A[i]
        i = i - 1
    A[i+1] = key
```

- **Loop invariant:** subarray $A[1..j\text{-}1]$ consists of the elements originally in $A[1..j\text{-}1]$ but in sorted order.
  - **Initialization:** $j = 2 \Rightarrow A[1]$ is sorted.
  - **Maintenance:** Move $A[j\text{-}1]$, $A[j\text{-}2]$,… one position to the right until the position for $A[j]$ is found.
  - **Termination:** $j = \boldsymbol{n+1} \Rightarrow A[1..n]$ is sorted. Hence the entire array is sorted!

Exact Analysis of Insertion Sort

InsertionSort(*A*)	cost	time
1. **for** *j* = 2 **to** *A.length* **do**	c_1	n
2. *key* = *A*[*j*]	c_2	n-1
3. /* Insert *A*[*j*] into the sorted sequence *A*[1..*j*-1] */	0	n-1
4. *i* = *j* – 1	c_4	n-1
5. **while** *i* > 0 and *A*[*i*] > *key* **do**	c_5	$\sum_{j=2}^{n} t_j$
6. *A*[*i*+1] = *A*[*i*]	c_6	$\sum_{j=2}^{n}(t_j - 1)$
7. *i* = *i* - 1	c_7	$\sum_{j=2}^{n}(t_j - 1)$
8. *A*[*i*+1] = *key*	c_8	n-1

- The **for** loop is executed (*n*-1) + 1 times. (why?)
- t_j: # of times the while loop test for value *j* (i.e., 1 + # of elements that have to be slid right to insert the *j*-th item).
- Step 5 is executed $t_2 + t_3 + \ldots + t_n$ times.
- Step 6 and 7 are executed $(t_2 - 1) + (t_3 - 1) + \ldots + (t_n - 1)$ times.
- Runtime: $T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n}(t_j - 1) + c_7 \sum_{j=2}^{n}(t_j - 1) + c_8(n-1)$

Exact Analysis of Insertion Sort

InsertionSort(*A*)	cost	time
for *j* = 2 **to** *A.length* **do**	c_1	n
key = *A*[*j*]	c_2	n-1
/* Insert *A*[*j*] into the sorted sequence *A*[1..*j*-1] */	0	n-1
i = *j* – 1	c_4	n-1
while *i* > 0 and *A*[*i*] > *key* **do**	c_5	$\sum_{j=2}^{n} t_j$
A[*i*+1] = *A*[*i*]	c_6	$\sum_{j=2}^{n}(t_j - 1)$
i = *i* - 1	c_7	$\sum_{j=2}^{n}(t_j - 1)$
A[*i*+1] = *key*	c_8	*n-1*

- $T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n}(t_j - 1) + c_7 \sum_{j=2}^{n}(t_j - 1) + c_8(n-1)$
- **Best case:** If the input is already sorted, all t_j's are 1.
  - Linear: $T(n) = (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8)$
- **Worst case:** If the array is in reverse sorted order, $t_j = j$, $\forall j$.
  - Quadratic: $T(n) = \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right)n - (c_2 + c_4 + c_5 + c_8)$

Exact analysis is often hard (and tedious)!

Asymptotic Analysis

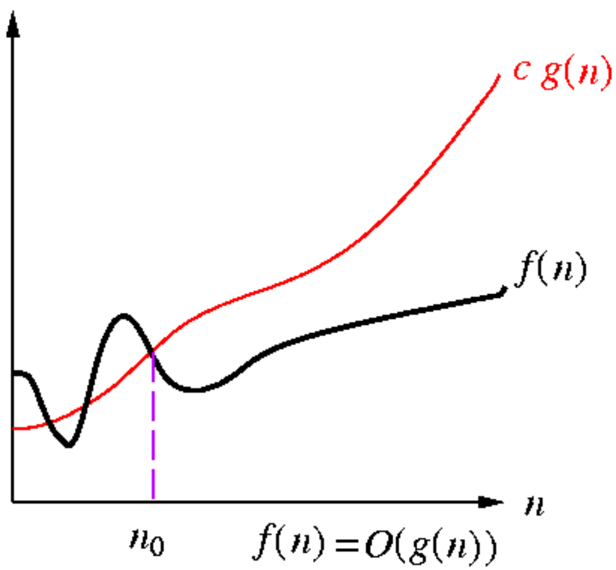
- Asymptotic analysis looks at growth of $T(n)$ as $n \rightarrow \infty$.
- θ notation: drop low-order terms and ignore the leading constant.
  - E.g., $T(n) = 8n^3 - 4n^2 + 5n - 2 = \theta(n^3)$.
- As *n* grows large, lower-order θ algorithms outperform higher-order ones.
  - Ex: for large inputs, a $\theta(n^2)$ algorithm will run more quickly in the worst case than a $\theta(n^3)$ algorithm.
- Asymptotic analysis of insertion sort
  - **Worst case**: input reverse sorted, **while** loop is executed *j* times each iteration: $T(n) = \sum_{j=2}^{n} j = \Theta(\sum_{j=2}^{n} j) = \Theta(n^2)$
  - **Average case**: **while** loop is executed about *j* /2 times each iteration: $T(n) = \sum_{j=2}^{n} \frac{j}{2} = \Theta(\sum_{j=2}^{n} \frac{j}{2}) = \Theta(n^2)$

$$\sum_{j=2}^{n} j = \frac{(n+2)(n-1)}{2} = \frac{n^2+n-2}{2} = \Theta(n^2)$$

$$\sum_{j=2}^{n} \frac{j}{2} = \frac{1}{2}\sum_{j=2}^{n} j = \Theta(n^2)$$

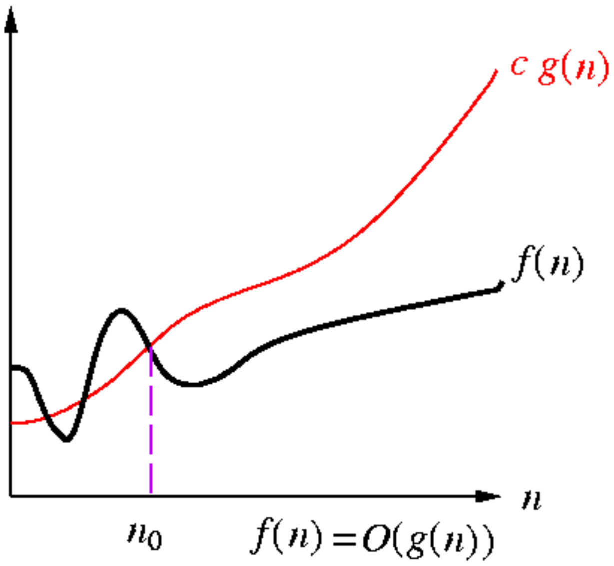
O: Upper Bounding Function

- **Def:** $f(n) = O(g(n))$ if $\exists\ c > 0$ and $n_0 > 0$ such that **$0 \le f(n) \le cg(n)$** for all $n \ge n_0$.
- Intuition: $f(n)$ " **≤** " $g(n)$ when we ignore constant multiples and small values of n.
- How to **verify** O (Big-Oh) relationships?
  - $f(n) = O(g(n))$ implies that $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$ for some $c \ge 0$, if the limit exists.



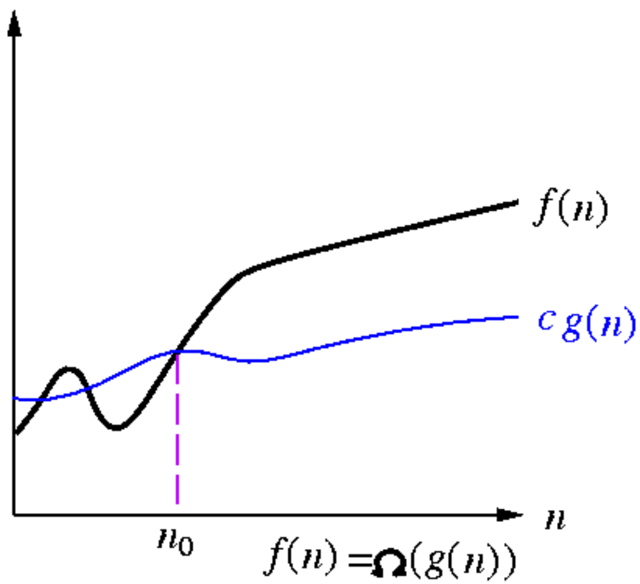
Big-Oh Examples

- **Def:** $f(n) = O(g(n))$ if $\exists\ c > 0$ and $n_0 > 0$ such that **$0 \le f(n) \le cg(n)$** for all $n \ge n_0$.
- Examples
  - $3n^2 + n = O(n^2)$? Yes! $\lim_{n\to\infty} \frac{3n^2+n}{n^2} = 3$
  - $3n^2 + n = O(n)$? No! $\lim_{n\to\infty} \frac{3n^2+n}{n} = \infty$
  - $3n^2 + n = O(n^3)$? Yes! $\lim_{n\to\infty} \frac{3n^2+n}{n^3} = 0$
  - $3n^2 + n \le cn^2$?
    - Take $c = 4$, $n_0 = 1$



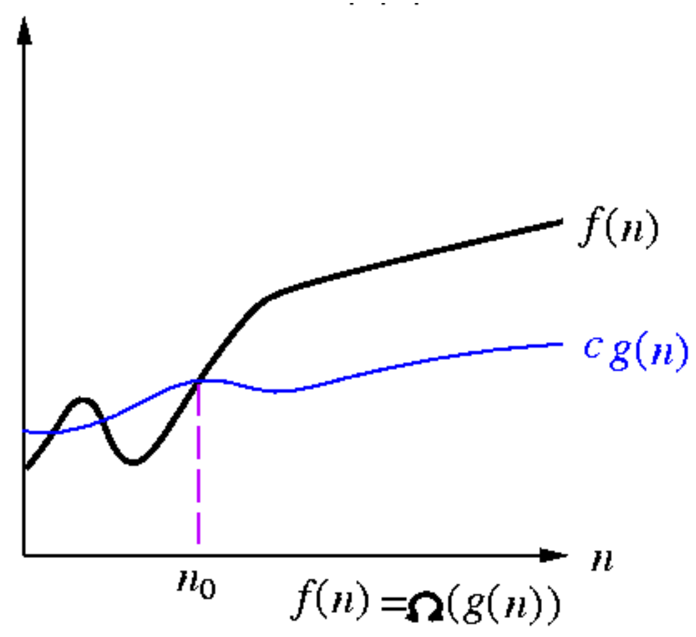
Ω : Lower Bounding Function

- **Def:** $f(n) = \Omega(g(n))$ if $\exists\ c > 0$ and $n_0 > 0$ such that **$0 \leq cg(n) \leq f(n)$** for all $n \geq n_0$.
- Intuition: $f(n)$ " **≥** " $g(n)$ when we ignore constant multiples and small values of n.
- How to **verify** Ω (Big-Omega) relationships?
  - $f(n) = \Omega(g(n))$ implies that $\lim_{n\to\infty} \frac{g(n)}{f(n)} = c$ for some $c \geq 0$, if the limit exists.



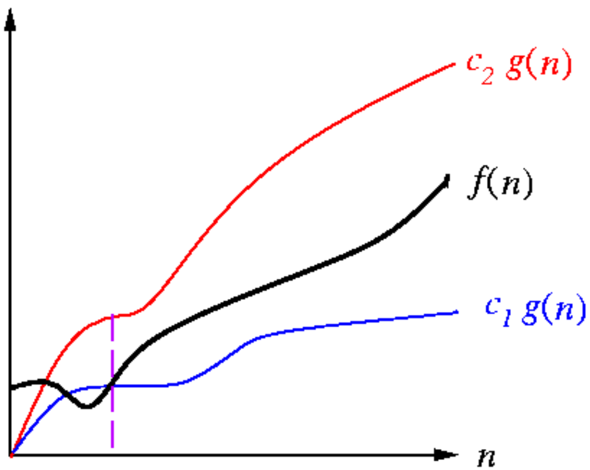
Big-Omega Examples

- **Def:** $f(n) = \Omega(g(n))$ if $\exists$ $c > 0$ and $n_0 > 0$ such that **$0 \leq cg(n) \leq f(n)$** for all $n \geq n_0$.
- Examples
  - $3n^2 + n = \Omega(n^2)$? Yes! $\lim_{n\to\infty} \frac{n^2}{3n^2+n} = \frac{1}{3}$
  - $3n^2 + n = \Omega(n)$? Yes! $\lim_{n\to\infty} \frac{n}{3n^2+n} = 0$
  - $3n^2 + n = \Omega(n^3)$? No! $\lim_{n\to\infty} \frac{n^3}{3n^2+n} = \infty$
  - $3n^2 + n \geq cn^2$?
    - Take $c = 2$, $n_0 = 1$



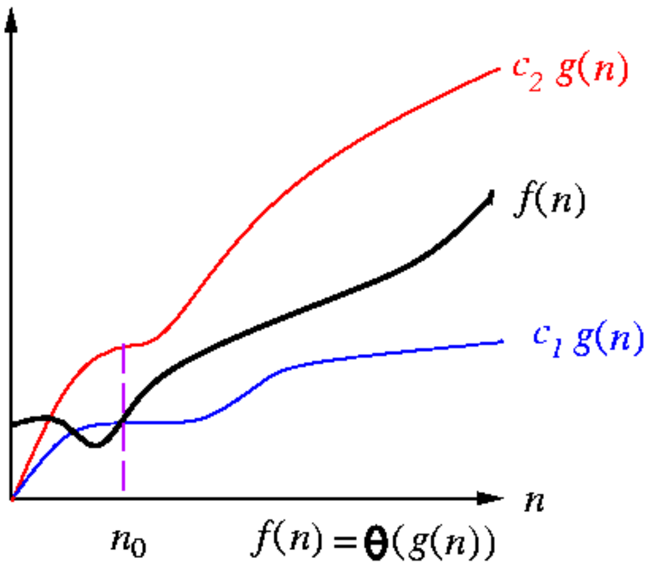
θ: Tightly Bounding Function

- **Def:** $f(n) = \theta(g(n))$ if $\exists\ c_1, c_2 > 0$ and $n_0 > 0$ such that **$0 \le c_1 g(n) \le f(n) \le c_2 g(n)$** for all $n \ge n_0$.
- Intuition: $f(n)$ " **=** " $g(n)$ when we ignore constant multiples and small values of n.
- How to **verify** θ relationships?
  - Show both "big Oh" (O) and "Big Omega" (Ω) relationships.
  - $f(n) = \theta(g(n))$ implies that $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$ for some **$c > 0$,** if the limit exists.



Theta Examples

- **Def:** $f(n) = \theta(g(n))$ if $\exists\ c_1, c_2 > 0$ and $n_0 > 0$ such that $\mathbf{0 \le c_1 g(n) \le f(n) \le c_2 g(n)}$ for all $n \ge n_0$.
- Examples
  - $3n^2 + n = \theta(n^2)$? Yes! $\lim_{n\to\infty} \frac{3n^2+n}{n^2} = 3$
  - $3n^2 + n = \theta(n)$? No! $\lim_{n\to\infty} \frac{3n^2+n}{n} = \infty$
  - $3n^2 + n = \theta(n^3)$? No! $\lim_{n\to\infty} \frac{3n^2+n}{n^3} = 0$
  - $c_1 n^2 \le 3n^2 + n \le c_2 n^2$?
    - Take $c_1 = 2$, $c_2 = 4$, $n_0 = 1$



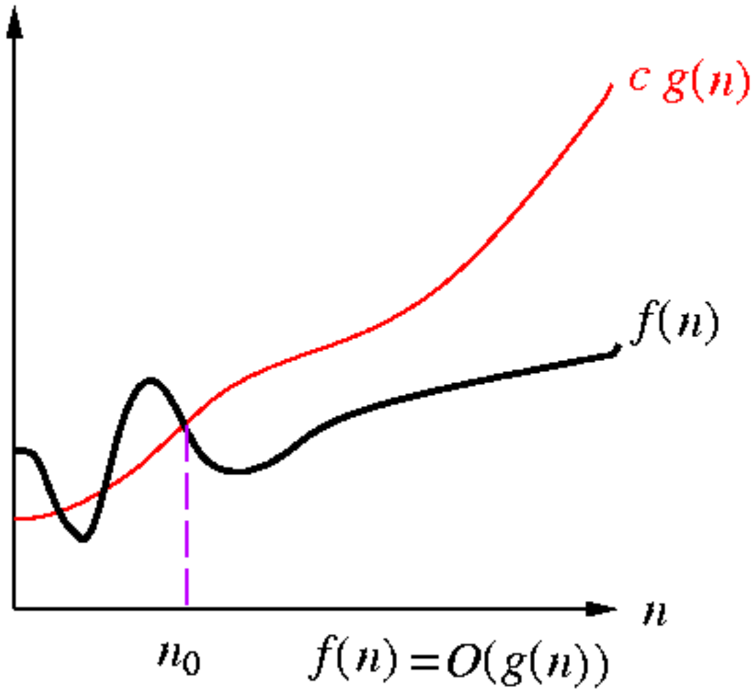
o, ω : Untightly Upper, Lower Bounding Functions

- **Little Oh *o*:** $f(n) = o(g(n))$ if $\forall$ **$c > 0$**, $\exists\, n_0 > 0$ such that $0 \le$ **$f(n) < cg(n)$** for all $n \ge n_0$.
- Intuition: $f(n)$ "**<** " **any constant multiple of** $g(n)$ when we ignore small values of n.
- **Little Omega ω :** $f(n) = \omega(g(n))$ if $\forall$ **$c > 0$**, $\exists\, n_0 > 0$ such that $0 \le$ **$cg(n) < f(n)$** for all $n \ge n_0$.
- Intuition: $f(n)$ is "**>**" any constant multiple of $g(n)$ when we ignore small values of n.
- How to **verify** *o* (Little-Oh) and ω (Little-Omega) relationships (if the limit exists)?
  - $f(n) = o(g(n))$ implies that $\lim_{n \to \infty} \dfrac{f(n)}{g(n)} = 0$.
  - $f(n) = \omega(g(n))$ implies that $\lim_{n \to \infty} \dfrac{f(n)}{g(n)} = \infty$.

Little-Oh Examples

- **Little Oh *o*:** $f(n) = o(g(n))$ if $\forall$ **$c > 0$**, $\exists\ n_0 > 0$ such that $0 \leq f(n) < cg(n)$ for all $n \geq n_0$.

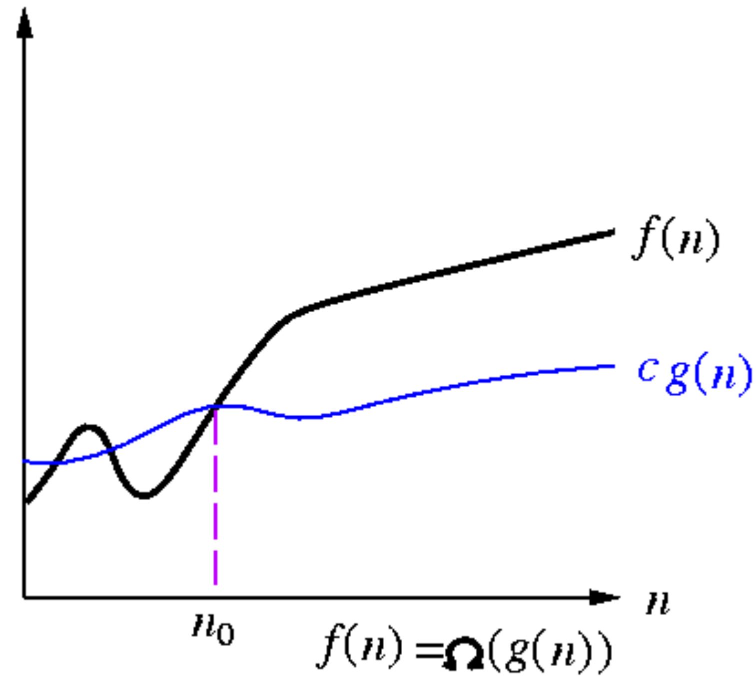
1. **$3n^2 + n = o(n^2)$?** No
2. **$3n^2 + n = o(n)$?** No
3. **$3n^2 + n = o(n^3)$?** Yes



$f(n) = o(g(n))$ implies that $\lim_{n \to \infty} \dfrac{f(n)}{g(n)} = 0$, if the limit exists

Little-Omega Examples

- **Little Omega** ω **:** $f(n) = \omega(g(n))$ if $\forall$ $\mathbf{c > 0}$, $\exists$ $n_0 > 0$ such that $0 \le \mathbf{cg(n) < f(n)}$ for all $n \ge n_0$.



1. **$3n^2 + n = \omega(n^2)$?** No
2. **$3n^2 + n = \omega(n)$?** Yes
3. **$3n^2 + n = \omega(n^3)$?** No

$f(n) = \omega(g(n))$ implies that $\lim_{n \to \infty} \dfrac{f(n)}{g(n)} = \infty$, if the limit exists

Algorithms with Asymptotic Notation

- "Insertion sort is a $O(n^2)$ algorithm" or "The running time of insertion sort is $O(n^2)$" **=> correct!**
  - The worst-case running time of insertion sort is $O(n^2)$
  - For any input of size n, the running time is at most cn^2
- "Insertion sort is a $\Omega(n)$ algorithm" or "The running time of insertion sort is $\Omega(n)$" **=> correct!**
  - The best-case running time of insertion sort is $\Omega(n)$
  - For any input of size n, the running time is at least cn
- "Insertion sort is a $\theta(n^2)$ algorithm" or "The running time of insertion sort is $\theta(n^2)$" **=> wrong!!**
  - For a sorted input, insertion sort runs in $\theta(n)$

Asymptotic Functions

- ***Polynomial-time complexity:*** $O(p(n))$,
  - n: the **input size.**
  - $p(n)$: a polynomial function of n **($p(n) = n^{O(1)}$).**

low → high: complexity

1	Constant
$\lg n$	Logarithmic
$\lg^{O(1)} n = (\lg n)^{O(1)}$	Polylogarithmic
$\sqrt{n}$	Sublinear
n	Linear
$n \lg n$	Loglinear
n^2	Quatratic
n^3	Cubic
n^4	Quartic
$2^n, 3^n, \ldots$	Exponential
$n!$	Factorial
n^n	-

Polynomial-time complexity

An Example

- Rank the following functions by the order of growth:
  - n^2
  - $n^{\lg n}$

Computational Complexity

- Computational complexity: an abstract measure of the **time** and **space** necessary to execute an algorithm as functions of its "input size".
  - Sort n words of bounded length ⇒ input size: n
  - The input is the graph $G(V, E)$ ⇒ input size: $|V|$ and $|E|$
- Runtime comparison

Time	Big-Oh	n = 10	n = 100	$n = 10^4$	$n = 10^6$	$n = 10^8$
500	O(1)	$5*10^{-7}$ sec	$5*10^{-7}$ sec	$5*10^{-7}$ sec	$5*10^{-7}$ sec	$5*10^{-7}$ sec
3n	O(n)	$3*10^{-8}$ sec	$3*10^{-7}$ sec	$3*10^{-5}$ sec	0.003 sec	0.3 sec
nlgn	O(n lg n)	$3*10^{-8}$ sec	$6*10^{-7}$ sec	$1*10^{-4}$ sec	0.018 sec	2.5 sec
n^2	$O(n^2)$	$1*10^{-7}$ sec	$1*10^{-5}$ sec	0.1 sec	16.7 min	116 days
n^3	$O(n^3)$	$1*10^{-6}$ sec	0.001 sec	16.7 min	31.7 yr	∞
2^n	$O(2^n)$	$1*10^{-6}$ sec	$4*10^{11}$ cent.	∞	∞	∞
n!	O(n!)	0.003 sec	∞	∞	∞	∞

Runtime Analysis

- Two rules
  - A number of operations are performed in an algorithm, the runtime is dominated by the most expensive operation
  - If an operation is repeatedly performed a number of times, the total runtime is the runtime of the operation multiplied by the iteration count
- Example

if (condition) **then**	$O(1)$
op1	$T_1(n)$
else	
op2	$T_2(n)$

  - $T(n) = O(\max(T_1(n), T_2(n)))$

Runtime Analysis (cont'd)

- Example

```
for i = 1 to n
    if A[i] > maxVal then          O(1)
        maxVal = A[i]              O(1)
        maxIdx = i                 O(1)
```

  – T(n) = O(n)

```
for i = 1 to n
    for j = 1 to n
        sum += A[i][j]             O(1)
```

  – $T(n) = O(n^2)$

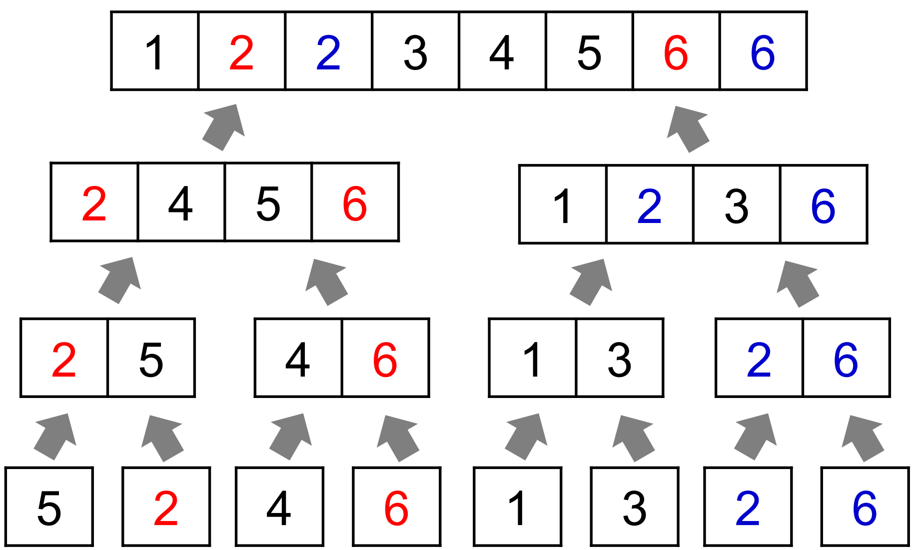
Merge Sort and Asymptotic Analysis

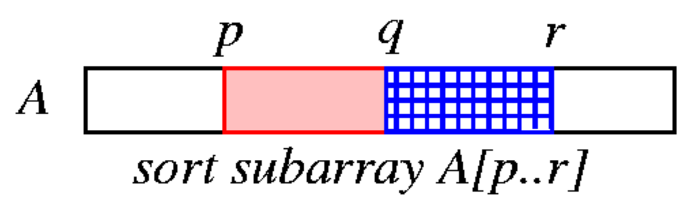
Divide-and-Conquer Algorithms

- **The divide-and-conquer paradigm**
  - **Divide** the problem into a number of subproblems.
  - **Conquer** the subproblems (solve them).
  - **Combine** the subproblem solutions to get the solution to the original problem.
- **Merge sort:** a divide-and-conquer algorithm
  - **Divide** the n-element sequence to be sorted into two $n/2$-element sequences.
  - **Conquer:** sort the subproblems, recursively using merge sort.
  - **Combine:** merge the resulting two sorted $n/2$-element sequences.

Merge Sort

MergeSort(A, p, r)	$T(n)$
1. **if** $p < r$	$\theta(1)$
2. $q = \lfloor (p+r)/2 \rfloor$	$\theta(1)$
3. MergeSort (A, p, q)	$T(n/2)$
4. MergeSort (A, q +1, r)	$T(n/2)$
5. Merge(A, p, q, r)	$\theta(n)$





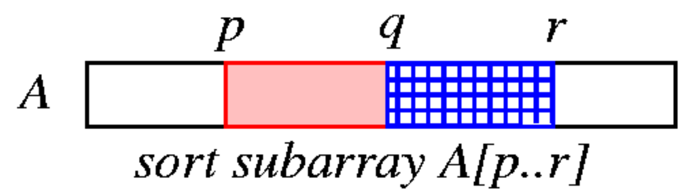
Merge(A, p, q, r): merge two sorted subarrays $A[p..q]$ and $A[q+1..r]$ into sorted $A[p..r]$.

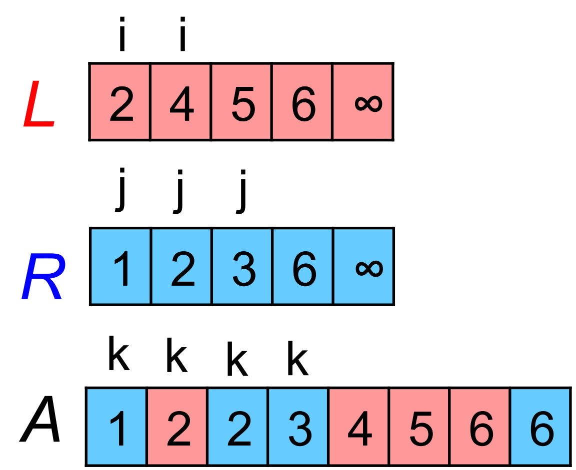
Merge

```
Merge (A, p, q, r)
n1 = q - p + 1
n2 = r - q
let L[1..n1+1] and R[1..n2+1] be new arrays
for i = 1 to n1
    L[i] = A[p + i - 1]
for j = 1 to n2
    R[j] = A[q + j]
L[n1+1] = ∞
R[n2+1] = ∞
i = 1
j = 1
for k = p to r
    if L[i] ≤ R[j]
        A[k] = L[i]
        i = i + 1
    else A[k] = R[j]
        j = j + 1
```

$\Theta(n)$ time!

Merge(*A*, *p*, *q*, *r*): merge two sorted subarrays $A[p..q]$ and $A[q+1..r]$ into sorted $A[p..r]$.

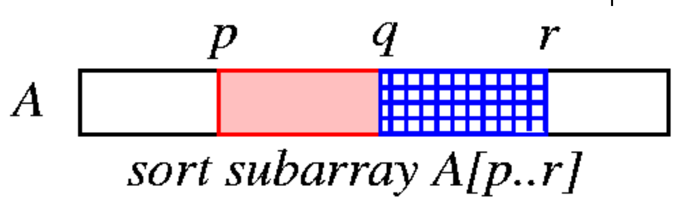


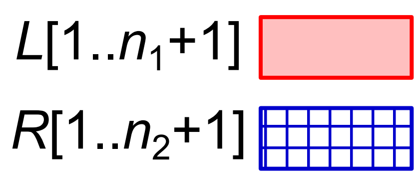


Merge (cont'd)

Merge (A, p, q, r)

1. $n_1 = q - p + 1$
2. $n_2 = r - q$
3. let $L[1..n_1+1]$ and $R[1..n_2+1]$ be new arrays
4. **for** $i = 1$ **to** n_1
5. $L[i] = A[p + i - 1]$
6. **for** $j = 1$ **to** n_2
7. $R[j] = A[q + j]$
8. $L[n_1+1] = \infty$
9. $R[n_2+1] = \infty$
10. $i = 1$
11. $j = 1$
12. **for** $k = p$ **to** r
13.   **if** $L[i] \leq R[j]$
14. $A[k] = L[i]$
15. $i = i + 1$
16.   **else** $A[k] = R[j]$
17. $j = j + 1$

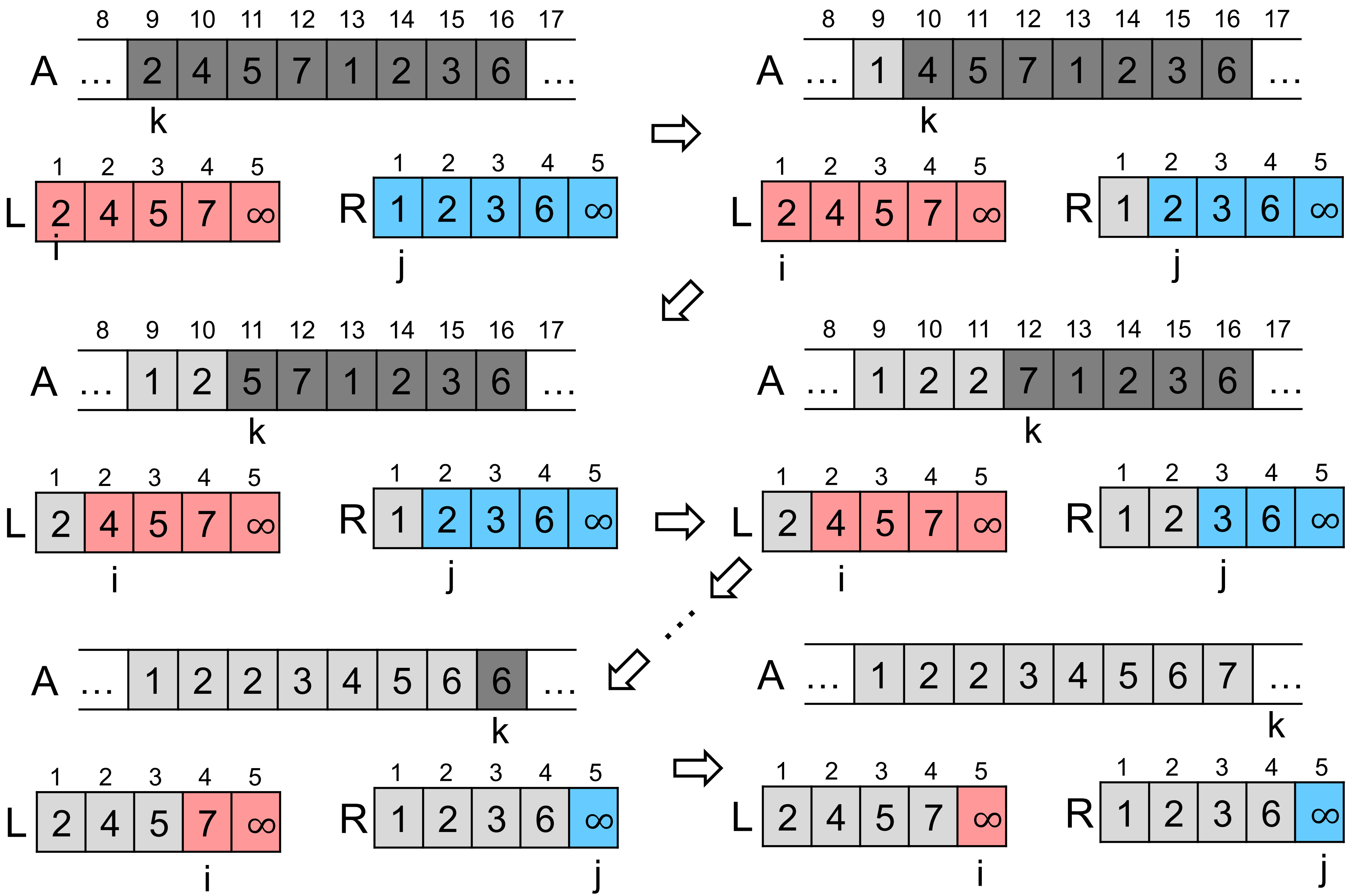




Loop invariant:
At the start of each iteration of the **for** loop (lines 12—17):

- The subarray $A[p..k-1]$ contains the $k-p$ smallest elements of $L[1..n_1+1]$ and $R[1..n_2+1]$, in sorted order.
- $L[i]$ and $R[j]$ are the smallest elements of their arrays that have not been copied back into A.

Merge (cont'd)



A … 2 4 5 7 1 2 3 6 …
k
L 2 4 5 7 ∞
R 1 2 3 6 ∞
i
j
A … 1 4 5 7 1 2 3 6 …
A … 1 2 5 7 1 2 3 6 …
A … 1 2 2 7 1 2 3 6 …
A … 1 2 2 3 4 5 6 6 …
A … 1 2 2 3 4 5 6 7 …

Recurrence

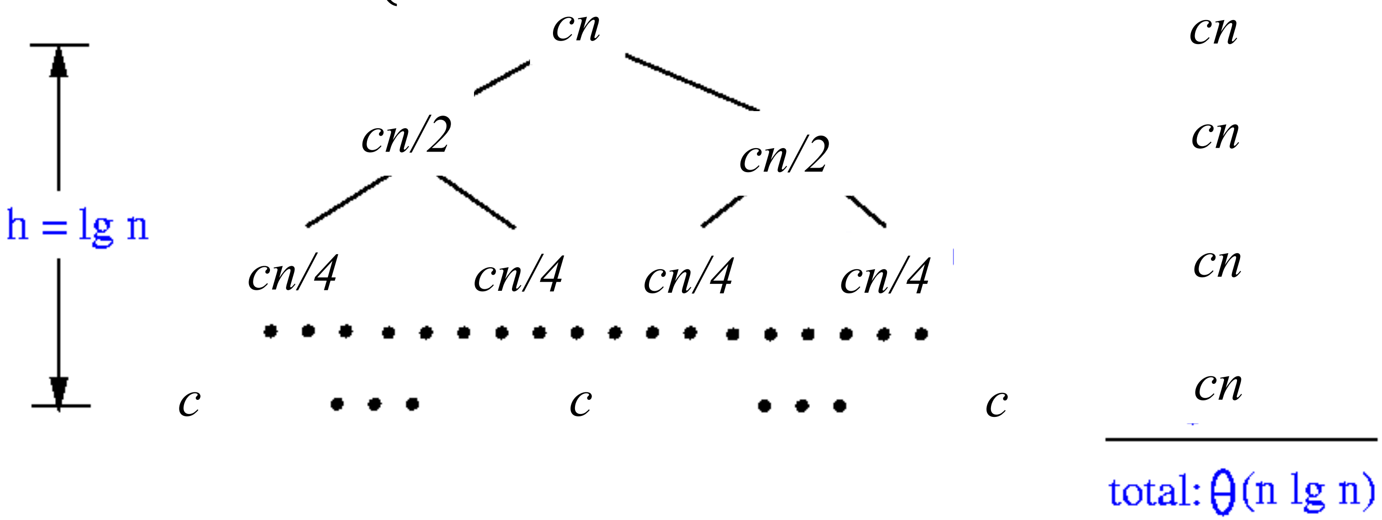
- Describes a function recursively in terms of itself.
- Describes performance of recursive algorithms.
- Recurrence for merge sort

$$T(n) = \begin{cases} \Theta(1), & if\ n = 1 \\ 2T\left(\frac{n}{2}\right) + \Theta(n), & if\ n > 1 \end{cases}$$

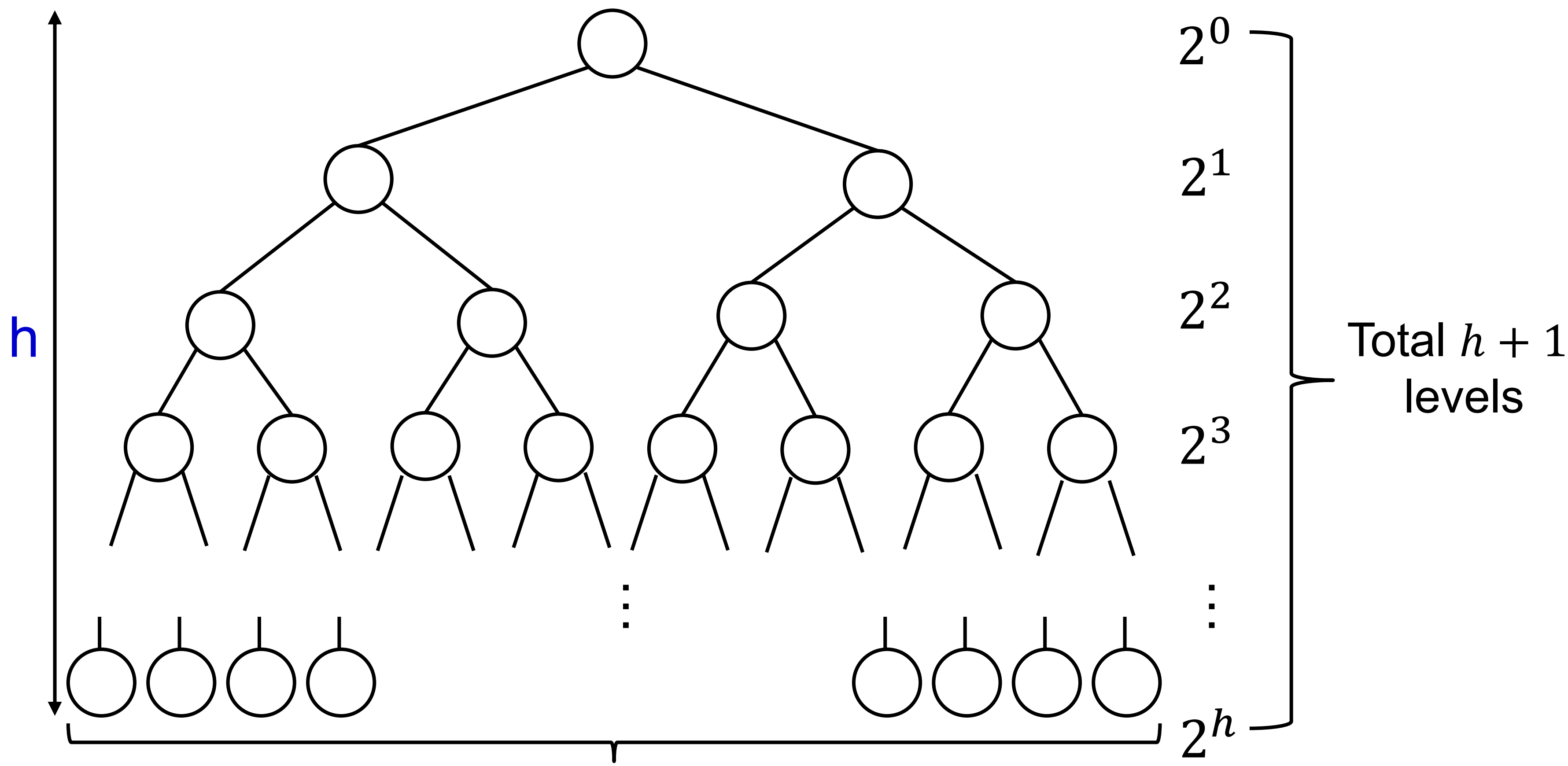
MergeSort(*A*, *p*, *r*)	*T*(*n*)
1. **if** *p* < *r*	θ(1)
2. *q* **=** ⌊(*p*+*r*)/2⌋	θ(1)
3. MergeSort (*A*, *p*, *q*)	*T*(*n*/2)
4. MergeSort (*A*, *q* +1, *r*)	*T*(*n*/2)
5. Merge(*A*, *p*, *q*, *r*)	θ(*n*)

Recursion Tree for Asymptotic Analysis

$$T(n) = \begin{cases} \Theta(1), & if\ n = 1 \\ 2T\left(\frac{n}{2}\right) + \Theta(n), & if\ n > 1 \end{cases}$$



- $\theta(n \lg n)$ grows more slowly than $\theta(n^2)$.
- Thus merge sort asymptotically beats insertion sort in the **worst case**.
  - Insertion sort: stable, in-place
  - Merge sort: stable, not in-place



$n = 2^h \Rightarrow h = \lg n$

Insertion Sort vs. Merge Sort

	Insertion Sort	**Merge Sort**
Approach	Incremental	Divide-and-conquer
Runtime Complexity	$O(n^2)$	$\Theta(n \lg n)$
Stable?	Y	Y
In-place?	Y	N

- **In-place**: Only a constant # of variables are stored outside the working array
- **Stable**: Numbers with the same value in the output array are in the same order as the input array

Analyzing Divide-and-Conquer Algorithms

- Recurrence for a divide-and-conquer algorithms

$$T(n) = \begin{cases} \Theta(1), & if\ n \leq c \\ aT\left(\frac{n}{b}\right) + D(n) + C(n), & otherwise \end{cases}$$

  - a: # of subproblems
  - $\frac{n}{b}$: size of the subproblems
  - $D(n)$: time to divide the problem of size *n* into subproblems
  - $C(n)$: time to combine the subproblem solutions to get the answer for the problem of size *n*
- Merge sort:

$$T(n) = \begin{cases} \Theta(1), & if\ n \leq c \\ 2T\left(\frac{n}{2}\right) + \Theta(n), & otherwise \end{cases}$$

  - $D(n) = \Theta(1)$: compute midpoint of array
  - $C(n) = \Theta(n)$: merging by scanning sorted subarrays

Solving Recurrences

- Three general methods for solving recurrences
  - **Iteration:** Convert the recurrence into a summation by expanding some terms and then bound the summation.
  - **Substitution:** Guess a solution and verify it by induction.
  - **Master Theorem:** if the recurrence has the form

$$T(n) = aT(n/b) + f(n),$$

    then **most likely** there is a formula that can be applied.
- Two **simplifications** that won't affect asymptotic analysis
  - Ignore floors and ceilings.
  - Assume base cases are constant, i.e., $T(n) = \theta(1)$ for small n.

Solving Recurrences: Iteration

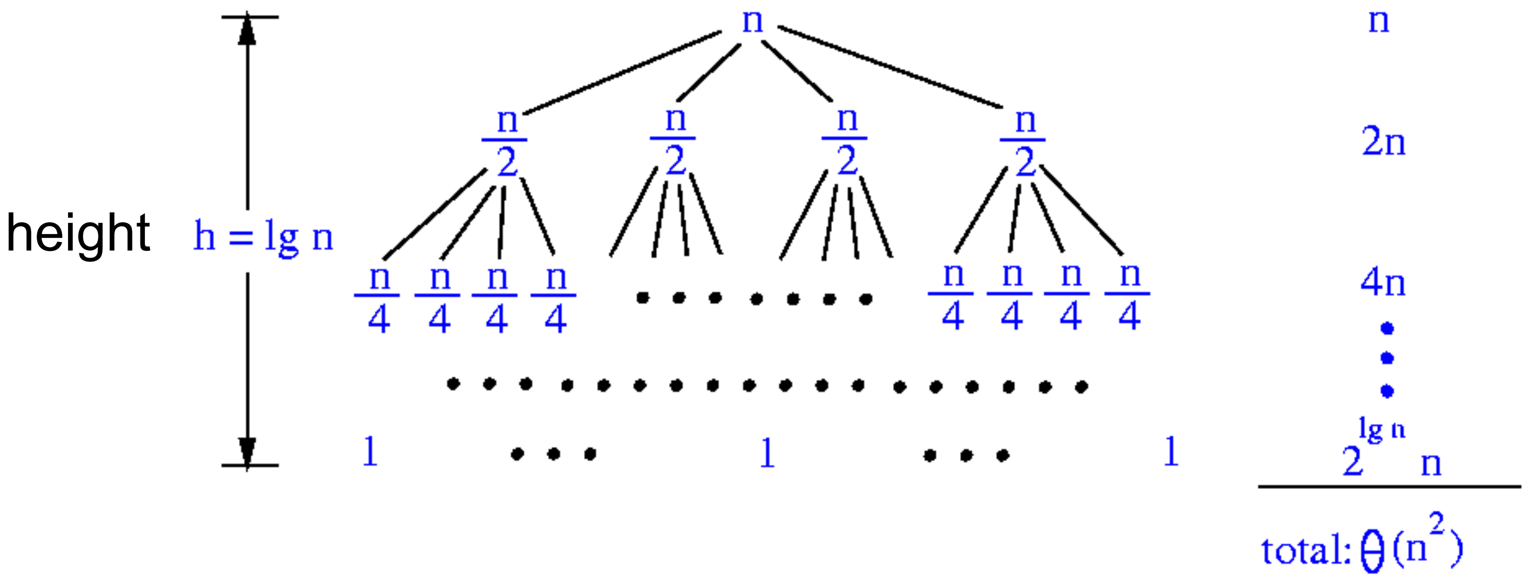
- **Example:** $T(n) = 4T(n/2) + n$.

$$
\begin{aligned}
T(n) &= 4T(n/2) + n \\
&= 4(4T(n/4) + n/2) + n & \text{/* expand */} \\
&= 16T(n/4) + 2n + n & \text{/* simplify */} \\
&= 16(4T(n/8) + n/4) + 2n + n & \text{/* expand */} \\
&= 64T(n/8) + 4n + 2n + n & \text{/* simplify */} \\
&= 4^{\lg n}T(1) + \ldots + 4n + 2n + n & \text{/* \#level } = \lg n \text{ */} \\
&= 4^{\lg n}c + n\sum_{k=0}^{\lg n - 1} 2^k & \text{/* convert to summation */} \\
&= cn^{\lg 4} + n\left(\frac{2^{\lg n} - 1}{2 - 1}\right) & \text{/* } a^{\lg b} = b^{\lg a} \text{ */} \\
&= cn^2 + n(n^{\lg 2} - 1) & \text{/* } 2^{\lg n} = n^{\lg 2} \text{ */} \\
&= (c+1)n^2 - n \\
&= \Theta(n^2)
\end{aligned}
$$

$$\boxed{S_n = \frac{a_1(r^n - 1)}{r - 1}}$$

Iteration by Using Recursion Trees

- Root: computation ($D(n) + C(n)$) at top level of recursion.
- Node at level i: Subproblem at level i in the recursion.
- Height of tree: (#levels - 1) in the recursion.
- $T(n)$ = sum of all nodes in the tree, $T(1) = 1$.
- $T(n) = 4T(n/2) + n = n + 2n + 4n + \ldots + 2^{\lg n}n = \theta(n^2)$.



Solving Recurrences: Substitution (Guess & Verify)

- Step 1 Guess form of a solution.
- Step 2 Apply math. induction to find the constant and verify the solution.
- Step 3 Use to find an upper or a lower bound.
- **Example**: Guess $T(n) = 2T(n/2) + n = O(n^2)$ ($T(1) = 1$)
  - Show $T(n) \le cn^2$ for some $c > 0$ (**we must find** c).
  - Step 1 Basis: $T(2) = 2T(1) + 2 = 4 \le 2^2c$
  - Step 2 Assume $T(k) \le ck^2$ for $k < n$, and prove $T(n) \le cn^2$

$$\begin{aligned} T(n) &= 2T(n/2) + n \\ &\le 2(c(n/2)^2) + n \\ &= cn^2/2 + n \\ &= cn^2 - (cn^2/2-n) \\ &\le cn^2, \end{aligned}$$

where $c \ge 2$ and $n \ge 1$.

Solving Recurrences: Master Theorem

- Let $a \geq 1$ and $b > 1$ be constants, $f(n)$ be an asymptotically **positive** function, and $T(n)$ be defined on nonnegative integers as

$$T(n) = aT(n/b) + f(n).$$

- Then, $T(n)$ can be bounded asymptotically as follows:

1. $T(n) = \Theta(n^{\log_b a})$ if $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$.
2. $T(n) = \Theta(n^{\log_b a} \lg n)$ if $f(n) = \Theta(n^{\log_b a})$.
3. $T(n) = \Theta(f(n))$ if $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$ and $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n.

- **Intuition: compare $f(n)$ with $\Theta(n^{\log_b a})$**
  - Case 1: $f(n)$ is polynomially smaller than $\Theta(n^{\log_b a})$
  - Case 2: $f(n)$ is asymptotically equal to $\Theta(n^{\log_b a})$
  - Case 3: $f(n)$ is polynomially larger than $\Theta(n^{\log_b a})$

Examples of Master Theorem

- **Ex. 1:** $T(n) = 4T(\frac{n}{2}) + n$
  - Compare $f(n) = n$ with $n^{\log_b a} = n^2$: $f(n) = n = O(n^{2-\epsilon})$.
  - Case 1 applies: $T(n) = \Theta(n^{\log_b a}) = \Theta(n^2)$.
- **Ex. 2:** $T(n) = 2T(\frac{n}{2}) + n$
  - Compare $f(n) = n$ with $n^{\log_b a} = n$: $f(n) = n = \Theta(n)$.
  - Case 2 applies: $T(n) = \Theta(n^{\log_b a} \lg n) = \Theta(n \lg n)$.

Examples of Master Theorem (cont'd)

- **Ex. 3:** $T(n) = 3T(\frac{n}{4}) + n \lg n$
  - Compare $f(n) = n \lg n$ with $n^{\log_b a} = n^{0.79}$: $f(n) = n \lg n = \Omega(n^{0.79+\epsilon})$.
  - Case 3 **could** apply: Need to **check for "regularity" condition** that $af(n/b) \leq cf(n)$.
    - Find $c < 1$ s.t. $af(n/b) \leq cf(n)$ for large enough n.
    - $3\frac{n}{4} \lg \frac{n}{4} \leq cn \lg n$ which is true for $c = \frac{3}{4}$.
  - Case 3 applies: $T(n) = \Theta(f(n)) = \Theta(n \lg n)$.

Examples of Master Theorem (cont'd)

- $T(n) = 2^n T\left(\frac{n}{2}\right) + n^n$
  - Master theorem cannot be applied (a is not constant).
- $T(n) = 0.5T\left(\frac{n}{2}\right) + \frac{1}{n}$
  - Master theorem cannot be applied ($a < 1$).
- $T(n) = 64T\left(\frac{n}{8}\right) - n^2 \log n$
  - Master theorem cannot be applied ($f(n)$ is not positive).
- $T(n) = 4T\left(\frac{n}{2}\right) + n^2 \lg n$
  - Master theorem cannot be applied
    - $n^{\log_b a} = n^2$, $f(n) = n^2 \lg n$, case 3 could apply
    - Regularity check: $4\left(\frac{n}{2}\right)^2 \lg\left(\frac{n}{2}\right) \leq cn^2 \lg n$, for some $c < 1$ and large n?