EEC E7205: Fundamentals of Computer Engineering

Analysis and Design of Algorithms

Analyzing Algorithms

- ***Analyzing*** an algorithm has come to mean predicting the resources that the algorithm requires.
  - Occasionally, resources such as memory, communication bandwidth, or computer hardware are of primary concern, but most often it is **running time** that we want to measure.
- By analyzing several candidate algorithms for a problem, we can identify a most efficient one.
- The running time depends on the input: an already sorted sequence is easier to sort.
- Parameterize the running time by the size of the input, since short sequences are easier to sort than long ones.
  - Look at ***growth*** as size $\rightarrow \infty$
- As running time depends on the speed of the computer, then ignore machine-dependent constants.

Kinds of Analysis

Best-case: (rarely)

- Cheat with a slow algorithm that works fast on *some* input.

Average-case: (sometimes)

- $T(n) =$ expected time of algorithm over all inputs of size n.
- Need assumption of statistical distribution of inputs.

Worst-case: (usually)

- $T(n) =$ maximum time of algorithm on any input of size n.

Input Size

- ***Input size*** depends on the problem being studied.
- For many problems, such as sorting, the most natural measure is the *number of items in the input*—for example, the array size n for sorting.
- Problems, such as multiplying two integers, the best measure of input size is the *total number of bits* needed to represent the input in ordinary binary notation.
- Sometimes, it is more appropriate to describe the size of the input with two numbers rather than one.
  - For instance, if the input to an algorithm is a graph, the input size can be described by the numbers of vertices and edges in the graph.

Running Time

- The ***running time*** of an algorithm on a particular input is the number of primitive operations or "steps" executed.
- It is convenient to define the notion of step so that it is as machine-independent as possible.
- One line of our pseudocode may take a different amount of time than another line, but we shall assume that each execution of the i^{th} line takes time c_i (where c_i is a constant.)

The Sorting Problem

Input: sequence $\langle a_1, a_2, \ldots, a_n \rangle$ of numbers.

Output: permutation $\langle a'_1, a'_2, \ldots, a'_n \rangle$ such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.

Example:

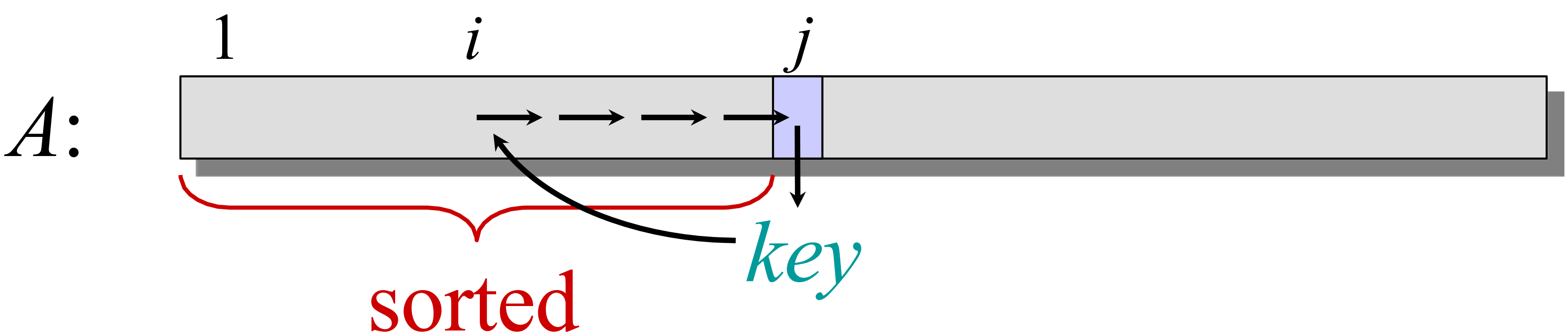
Input: 8 2 4 9 3 6

Output: 2 3 4 6 8 9

Insertion Sort Algorithm

“pseudocode”

INSERTION-SORT (array A , int n)
    for $j \leftarrow 2$ **to** n **do**
        $key \leftarrow A[j]$
        $i \leftarrow j - 1$
        while $i > 0$ and $A[i] > key$ **do**
                $A[i+1] \leftarrow A[i]$
                $i \leftarrow i - 1$
        $A[i+1] = key$



Example of insertion sort

8   2   4   9   3   6

Example of insertion sort

8 2 4 9 3 6

Example of insertion sort

8	2	4	9	3	6
2	8	4	9	3	6

Example of insertion sort

8 2 4 9 3 6

2 8 4 9 3 6

Example of insertion sort

8	2	4	9	3	6
2	8	4	9	3	6
2	4	8	9	3	6

Example of insertion sort

8	2	4	9	3	6
2	8	4	9	3	6
2	4	8	9	3	6

Example of insertion sort

8	2	4	9	3	6
2	8	4	9	3	6
2	4	8	9	3	6
2	4	8	9	3	6

Example of insertion sort

8 2 4 9 3 6

2 8 4 9 3 6

2 4 8 9 3 6

2 4 8 9 3 6

Example of insertion sort

8 2 4 9 3 6

2 8 4 9 3 6

2 4 8 9 3 6

2 4 8 9 3 6

2 3 4 8 9 6

Example of insertion sort

8	2	4	9	3	6
2	8	4	9	3	6
2	4	8	9	3	6
2	4	8	9	3	6
2	3	4	8	9	6

Example of insertion sort

8 2 4 9 3 6

2 8 4 9 3 6

2 4 8 9 3 6

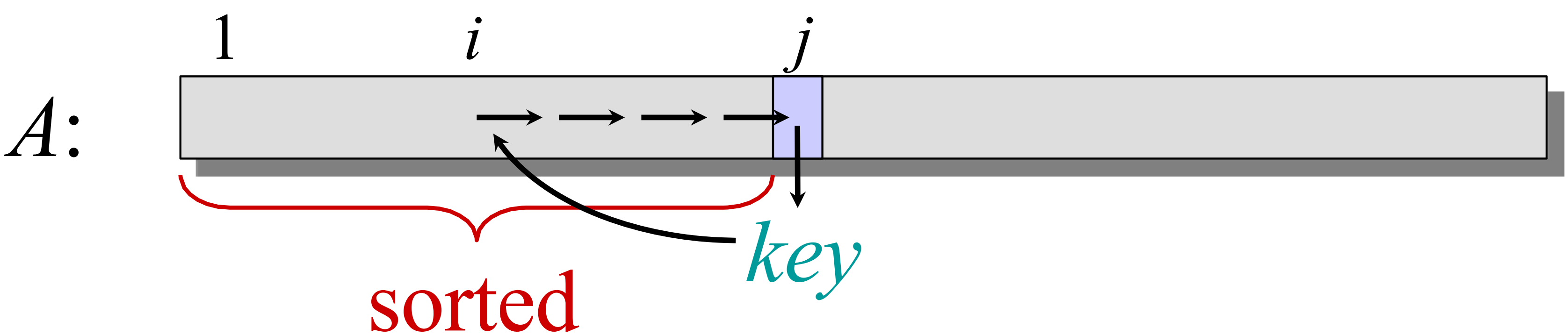
2 4 8 9 3 6

2 3 4 8 9 6

2 3 4 6 8 9 *done*

Analysis of Insertion Sort (1 of 2)

INSERTION-SORT(A)	*cost*	*times*
for $j = 2$ **to** $A.length$	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 $A[i] > key$	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	$n - 1$



Analysis of Insertion Sort (2 of 2)

- When a **for** or **while** loop exits in the usual way (i.e., due to the test in the loop header), the test is executed one time more than the loop body.
- t_j denote the number of times the **while** loop test in line 5 is executed for that value of *j*.
- Comments are not executable statements, and so they take no time.
- To compute *T(n)* , the running time of an input of *n* values, we sum the products of the *cost* and *times* columns, obtaining:

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) \\ &\quad + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1) \, . \end{aligned}$$

Best-Case Running Time

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1) .$$

- In Insertion Sort, the best case occurs if the array is already sorted. For each j = 2,3, …., n, we then find that A[i] ≤ *key* in line 5 when i has its initial value of j-1. Thus $t_j = 1$ for j = 2,3, …., n.
- The best-case running time is the following ***linear function*** of n:

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8) . \\ &= an + b \text{ for } constants\ a \text{ and } b \end{aligned}$$

Worst-Case Running Time (1 of 2)

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1) .$$

- In Insertion Sort, the worst case occurs if the array is in reverse sorted order. For each j = 2,3, …., n, we then find that A[i] > *key* in line 5 for all values of i. Thus $t_j = j$ for j = 2,3, …., n.
- The worst-case running time is the following ***quadratic function*** of n:

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) \\ &\quad + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\ &= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right)n \\ &\quad - (c_2 + c_4 + c_5 + c_8) . \\ &= an^2 + bn + c \text{ for constants } a, b, \text{ and } c \end{aligned}$$

Worst-Case Running Time (2 of 2)

$$an^2 + bn + c$$

- It is the ***rate of growth***, or ***order of growth***, of the running time that really interests us.
- We therefore consider only the leading term of a formula (an^2), since the lower-order terms are relatively insignificant for large values of n.
- We also ignore the leading term's constant coefficient, since constant factors are less significant than the rate of growth in determining computational efficiency for large inputs.
- We are left with the factor of n^2.
- We write that insertion sort has a worst-case running time of $\Theta(n^2)$ (pronounced "theta of n-squared").

Designing Algorithms

- For insertion sort, we used an ***incremental*** approach: having sorted the subarray *A[1 .. j-1]*, we inserted the single element *A[j]* into its proper place, yielding the sorted subarray *A[1 .. j]*.

- An alternative design approach is "***divide-and-conquer***".

Divide-and-Conquer Introduction

- Many useful algorithms are ***recursive*** in structure: to solve a given problem, they call themselves recursively one or more times to deal with closely related sub-problems.
- These algorithms typically follow a ***divide-and-conquer*** approach:
  - They break the problem into several sub-problems that are similar to the original problem but smaller in size,
  - Solve the sub-problems recursively, and
  - Then combine these solutions to create a solution to the original problem.

Divide-and-Conquer Steps

1. **Divide** the problem into a number of sub-problems that are smaller instances of the same problem.

2. **Conquer** the sub-problems by solving them recursively. If the sub-problem sizes are small enough, however, just solve the sub-problems in a straightforward manner.

3. **Combine** the solutions to the sub-problems into the solution for the original problem.

Merge Sort Algorithm (1 of 2)

MERGE-SORT $A[1 \,..\, n]$

1. If $n = 1$, done.
2. Recursively sort $A[\,1 \,..\, \lceil n/2 \rceil\,]$ and $A[\,\lceil n/2 \rceil + 1 \,..\, n\,]$.
3. "***Merge***" the 2 sorted lists.

Merge Sort Algorithm (2 of 2)

- The ***merge sort*** algorithm closely follows the divide-and-conquer paradigm. Intuitively, it operates as follows.
  - **Divide:** Divide the n-element sequence to be sorted into two sub-sequences of n/2 elements each.
  - **Conquer:** Sort the two sub-sequences recursively using merge sort.
  - **Combine:** Merge the two sorted sub-sequences to produce the sorted answer.
- Once the sub-sequences become small enough that we no longer recurse, we say that the recursion "bottoms out".
- The merge-sort recursion "bottoms out" when the sequence to be sorted has length 1, in which case there is no work to be done, since every sequence of length 1 is already in sorted order.

Merging Two Sorted Arrays

20	12
13	11
7	9
2	1

Merging Two Sorted Arrays

20	12
13	11
7	9
2	1

1

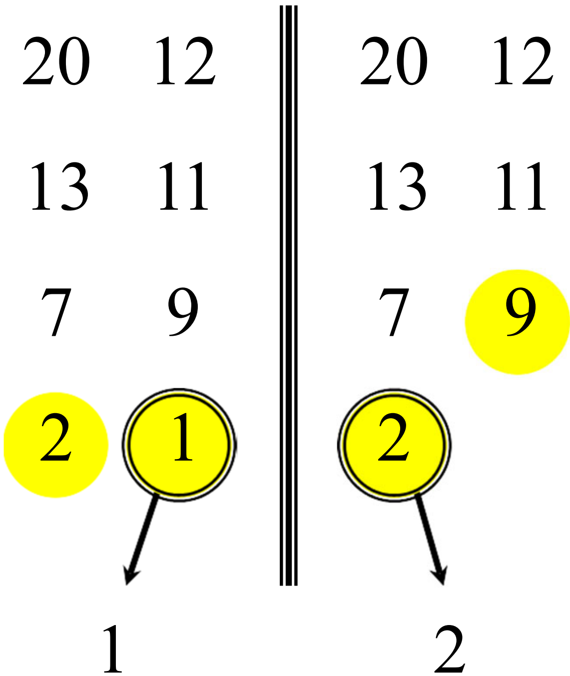
Merging Two Sorted Arrays

20	12
13	11
7	9
2	1

1

20	12
13	11
7	9
2	

Merging Two Sorted Arrays



Merging Two Sorted Arrays

20	12	‖	20	12	‖	20	12
13	11	‖	13	11	‖	13	11
7	9	‖	7	9	‖	7	9
2	1	‖	2		‖		
1			2				

Merging Two Sorted Arrays

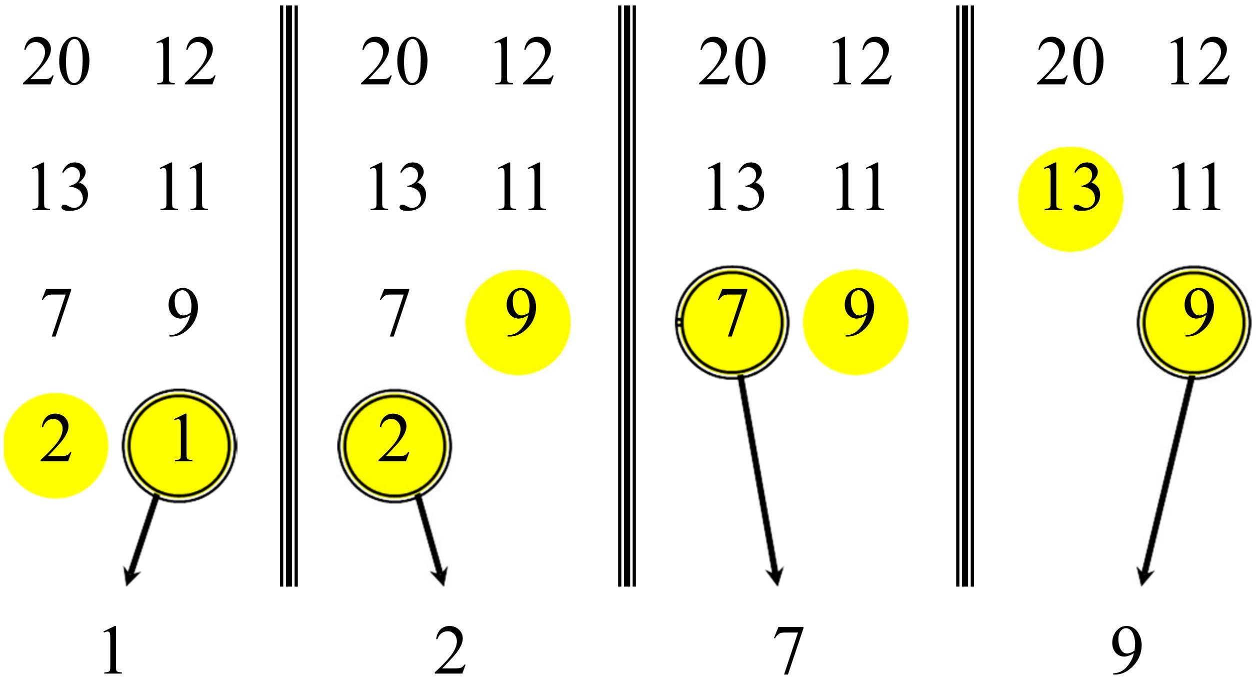
20	12	20	12	20	12
13	11	13	11	13	11
7	9	7	9	7	9
2	1	2			

1	2	7

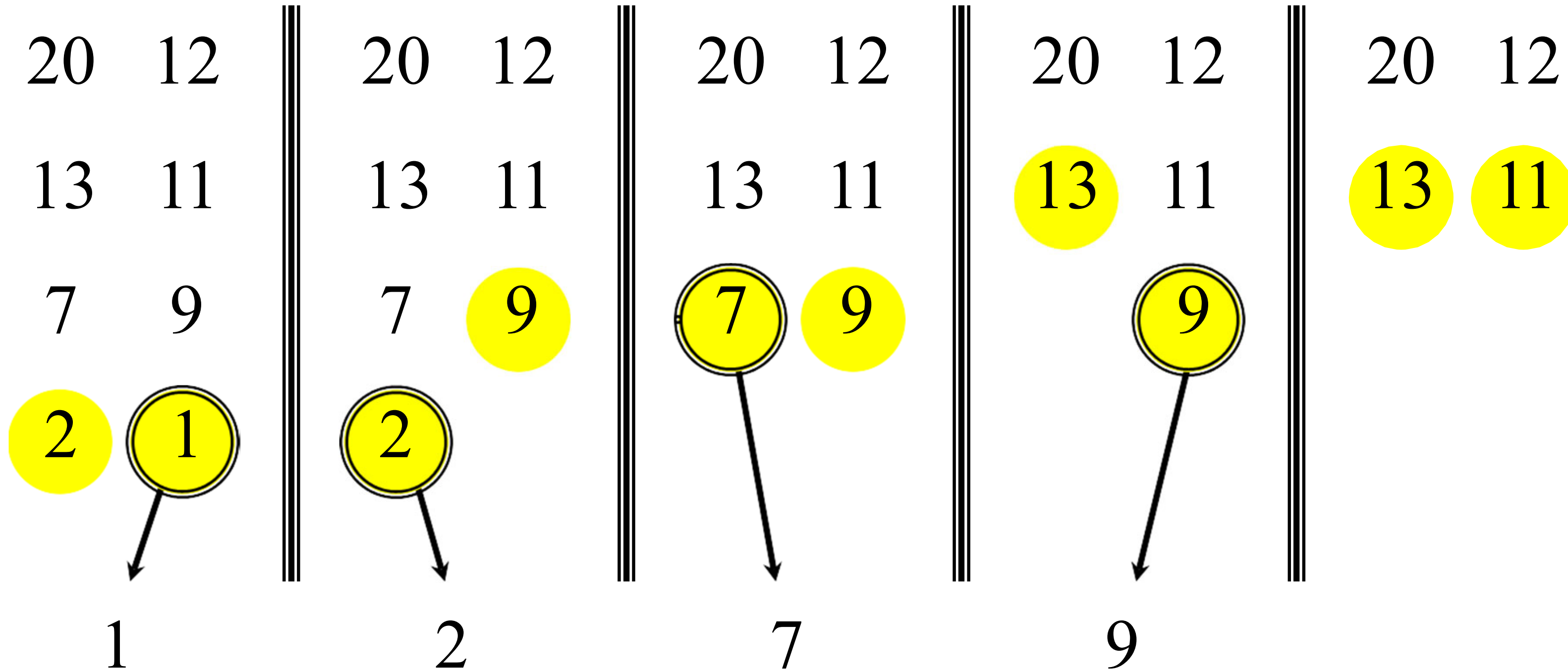
Merging Two Sorted Arrays

20	12	20	12	20	12	20	12
13	11	13	11	13	11	13	11
7	9	7	9	7	9		9
2	1	2					
1		2		7			

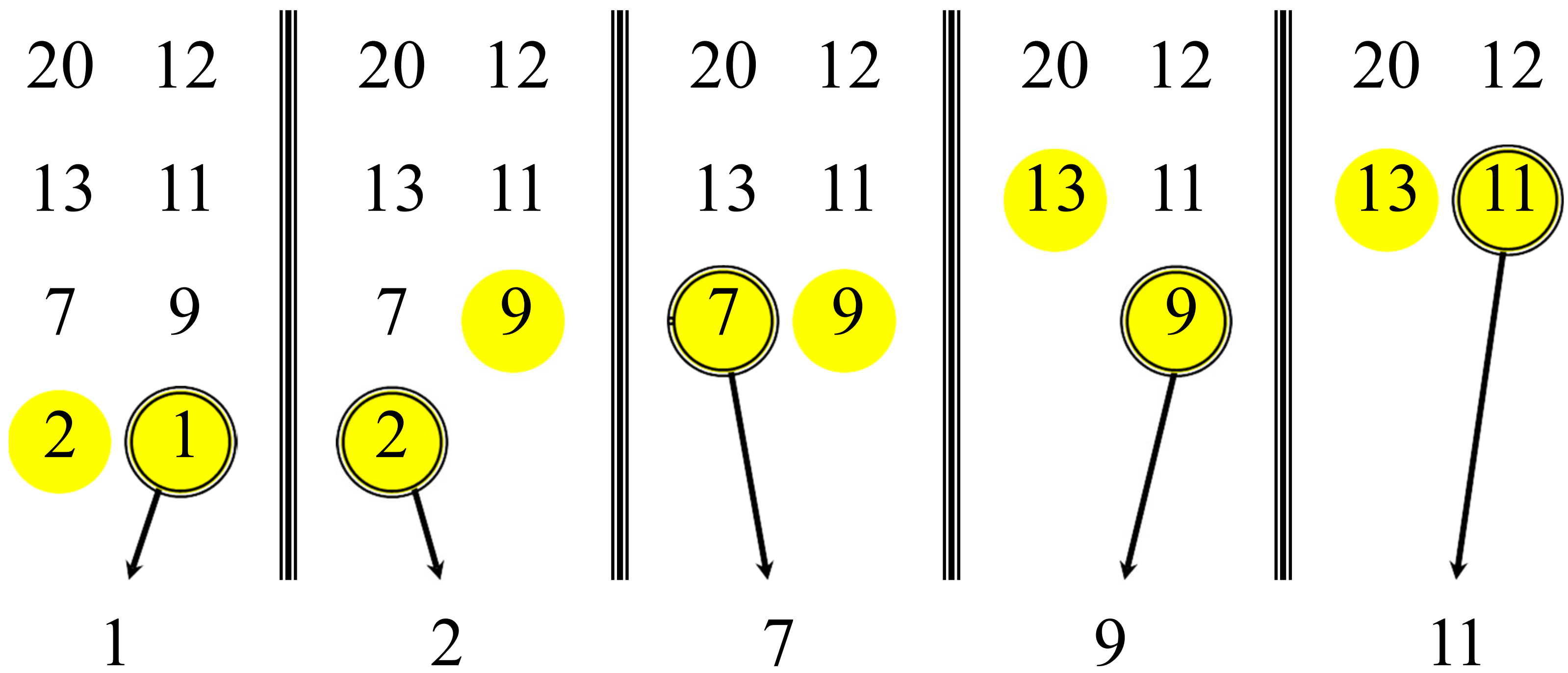
Merging Two Sorted Arrays



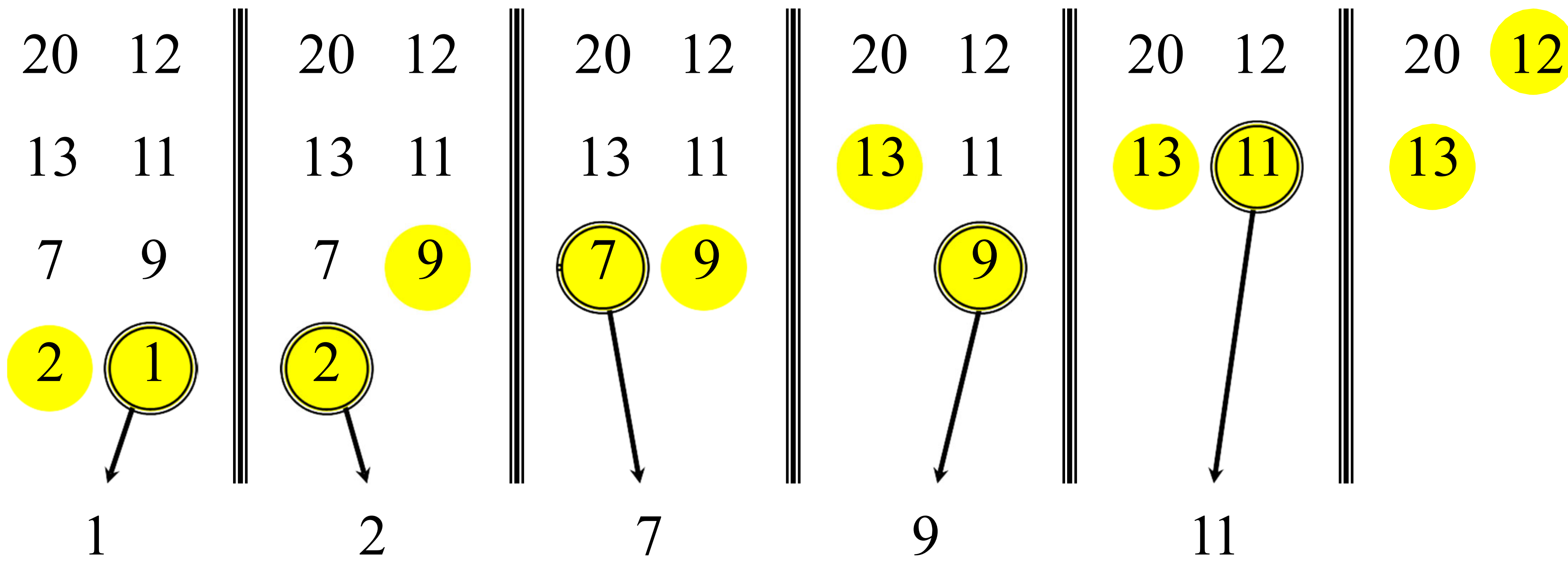
Merging Two Sorted Arrays



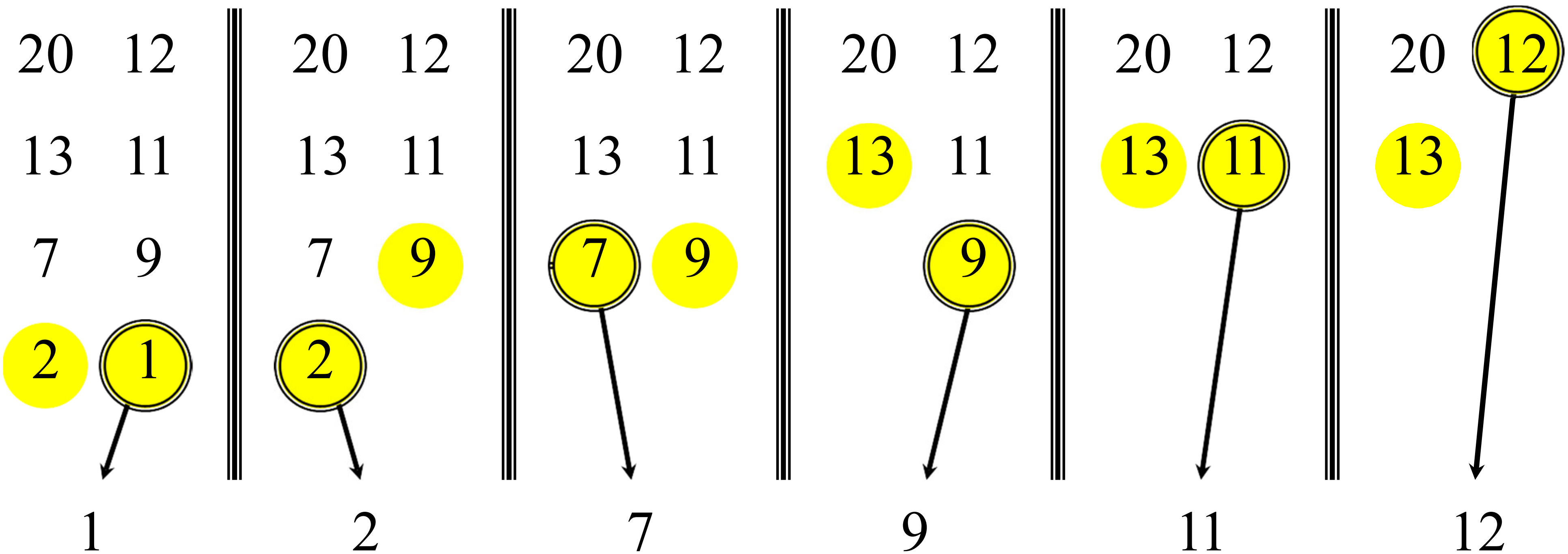
Merging Two Sorted Arrays



Merging Two Sorted Arrays



Merging Two Sorted Arrays



Analyzing Merge Sort

$T(n)$	**MERGE-SORT** $A[1 \,.\,.\, n]$
c	1. If $n = 1$, done.
$2T(n/2)$	2. Recursively sort $A[\,1 \,.\,.\, \lceil n/2 \rceil\,]$ and $A[\,\lceil n/2 \rceil + 1 \,.\,.\, n\,]$.
cn	3. ***"Merge"*** the 2 sorted lists

Should be $T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor)$, but it turns out not to matter for worst case analysis.

It is unlikely that the same constant exactly represents both the time to solve problems of size 1 and the time per array element of the divide and combine steps. We can get around this problem by letting c be the larger of these times.

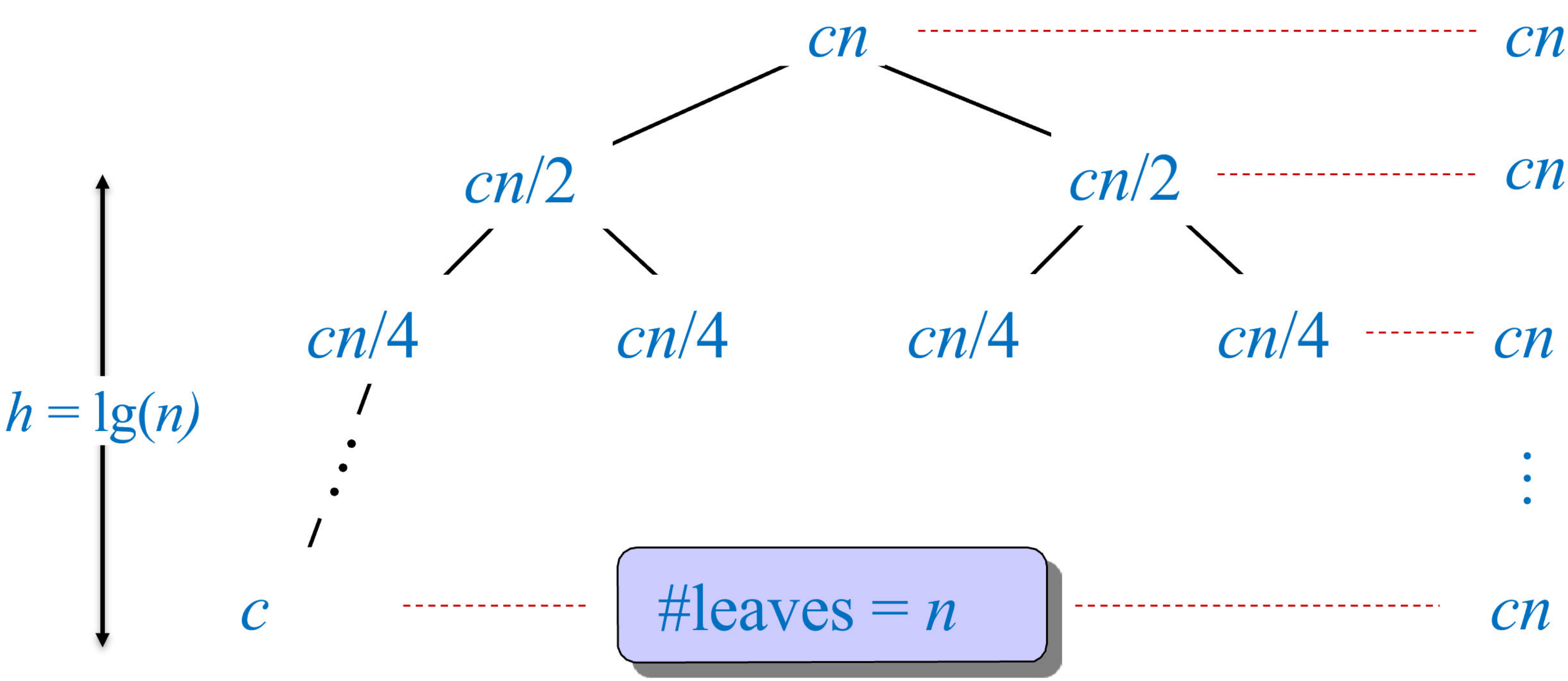
Recurrence for Merge Sort

- When an algorithm contains a recursive call to itself, we can often describe its running time by a ***recurrence equation*** or ***recurrence***.
- The recurrence equation for merge sort is:

$$T(n) = \begin{cases} c & \text{if } n = 1; \\ 2T(n/2) + cn & \text{if } n > 1. \end{cases}$$

Merge Sort Recursion Tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant and assuming n is power of 2



- Assuming $n = 2^h$, then we need h times of dividing n by 2 to reach 1.
- Total: $cn \lg n + cn$ → Worst-case running time of merge sort is $\Theta(n \log n)$
- *Note:* All logarithms are within constant factors of each other: $\log_b n = (\log_c n) / (\log_c b)$, which is a constant times $\log_c n$, for any base b & c
- So, we can use $O(\log n)$ without specifying a base such as 2 in $\lg(n)$ or e in $\ln(n)$

Merge Sort vs. Insertion Sort

- $\Theta(n \log n)$ grows more slowly than $\Theta(n^2)$.
- Therefore, merge sort beats insertion sort in the worst case.
- In practice, merge sort beats insertion sort for $n > 30$ or so.

Merging Two Sorted Lists

- The shown pseudocode merges two sorted lists.
- To avoid having to check whether either list is empty in each basic step, a ***sentinel*** value of ∞ is placed at the end of each list.

		p			q				r	
	8	9	10	11	12	13	14	15	16	17
A	…	2	4	5	7	1	2	3	6	…
		k								

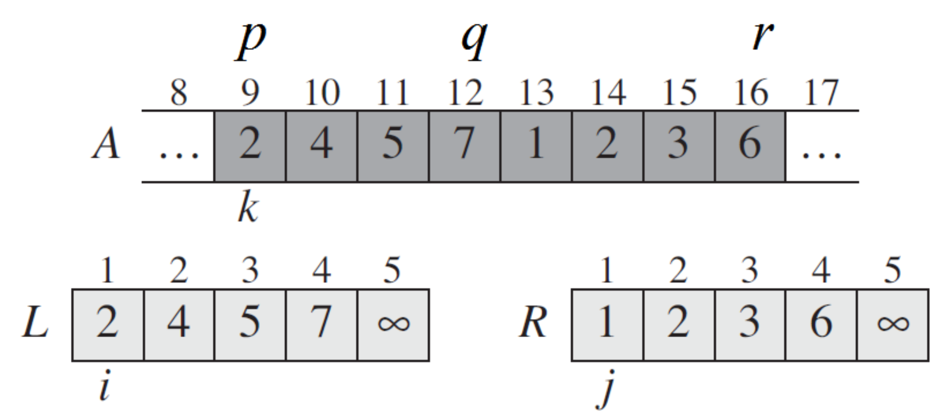
	1	2	3	4	5
L	2	4	5	7	∞
	i				

	1	2	3	4	5
R	1	2	3	6	∞
	j				

```
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
```

Updated Merge Sort Algorithm

- We can now use the MERGE procedure as a subroutine in the merge sort algorithm as shown




```
MERGE-SORT(A, p, r)
if p < r
    q = ⌊(p + r)/2⌋
    MERGE-SORT(A, p, q)
    MERGE-SORT(A, q + 1, r)
    MERGE(A, p, q, r)
```

- We make the initial call MERGE-SORT(*A, 1, A.length*).
- The number of elements in the sub-array to be sorted is r-p+1. So, r-p+1>1 is the condition to continue calling the merge sort recursively. r-p+1>1 is equivalent to $p<r$

Order of Growth

- The order of growth of the running time of an algorithm gives a simple characterization of the algorithm's efficiency and also allows us to compare the relative performance of alternative algorithms.
- Once the input size n becomes large enough, merge sort, with its $\Theta(n \lg n)$ worst-case running time, beats insertion sort, whose worst-case running time is $\Theta(n^2)$.
- Although we can sometimes determine the exact running time of an algorithm, the extra precision is not usually worth the effort of computing it.
- For large enough inputs, the multiplicative constants and lower-order terms of an exact running time are dominated by the effects of the input size itself.
- When we look at input sizes large enough to make only the order of growth of the running time relevant, we are studying the ***asymptotic*** efficiency of algorithms.
  - Usually, an algorithm that is asymptotically more efficient will be the best choice for all but very small inputs.

Asymptotic Notation

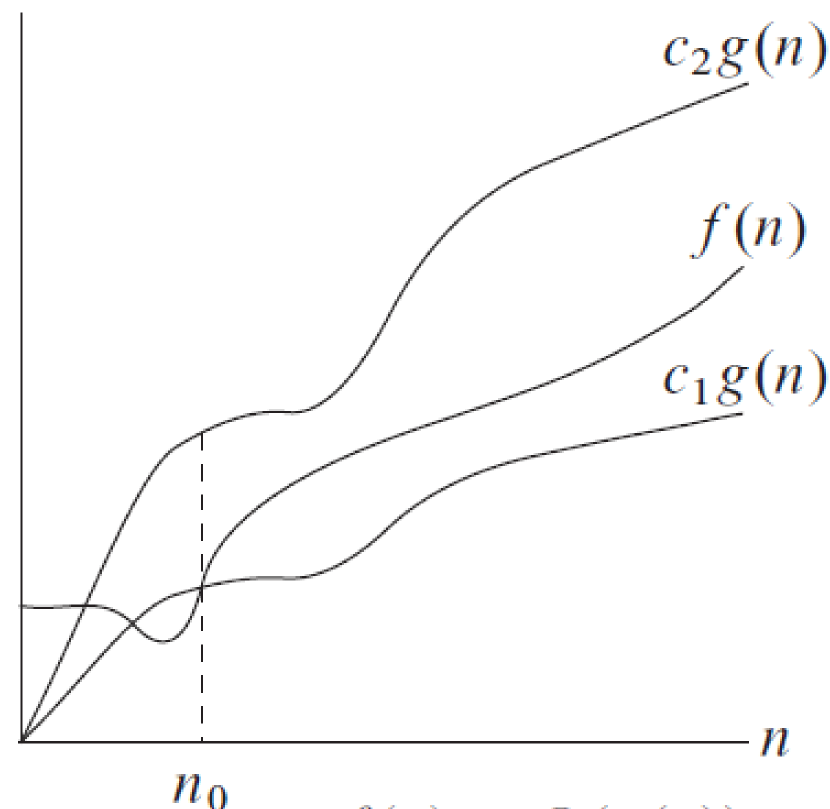
- Asymptotic notation is primarily used to describe the running times of algorithms, as when we wrote that insertion sort's worst-case running time is $\Theta(n^2)$.
- Asymptotic notation can apply to functions that characterize some other aspect of algorithms (the amount of space they use, for example).
- Asymptotic notation actually applies to functions.
  - What we were writing as $\Theta(n^2)$ was the function $an^2+bn + c$, which in that case happened to characterize the worst-case running time of insertion sort

Θ (big-theta) Notation

- For a given function *g(n)*, we denote by $\Theta(g(n))$ the *set of functions:*

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

- Because $\Theta(g(n))$ is a set, we could write "$f(n) \in \Theta(g(n))$"
  - We will usually write "$f(n) = \Theta(g(n))$" to express the same notion.
- The figure gives an intuitive picture of functions $f(n)$. It is "sandwiched" between $c_1g(n)$ and $c_2g(n)$, for all values of n at and to the right of n_0



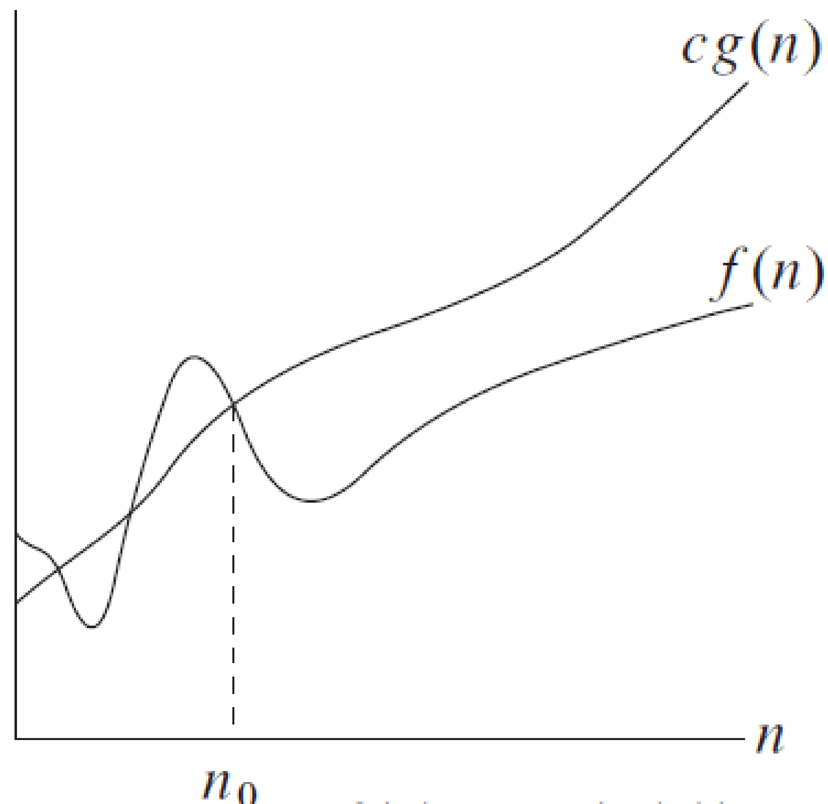
$f(n) = \Theta(g(n))$

O (big-oh) Notation

- For a given function *g(n)*, we denote by $O(g(n))$ the *set of functions:*

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

- O-notation gives an upper bound on a function.
- As shown $f(n)$ is below $cg(n)$, for all values of n at and to the right of n_0
- Note that $f(n) = \Theta(g(n))$ implies $f(n) = O(g(n))$, since Θ-notation is a stronger notion than O-notation and hence: $\Theta(g(n)) \subseteq O(g(n))$



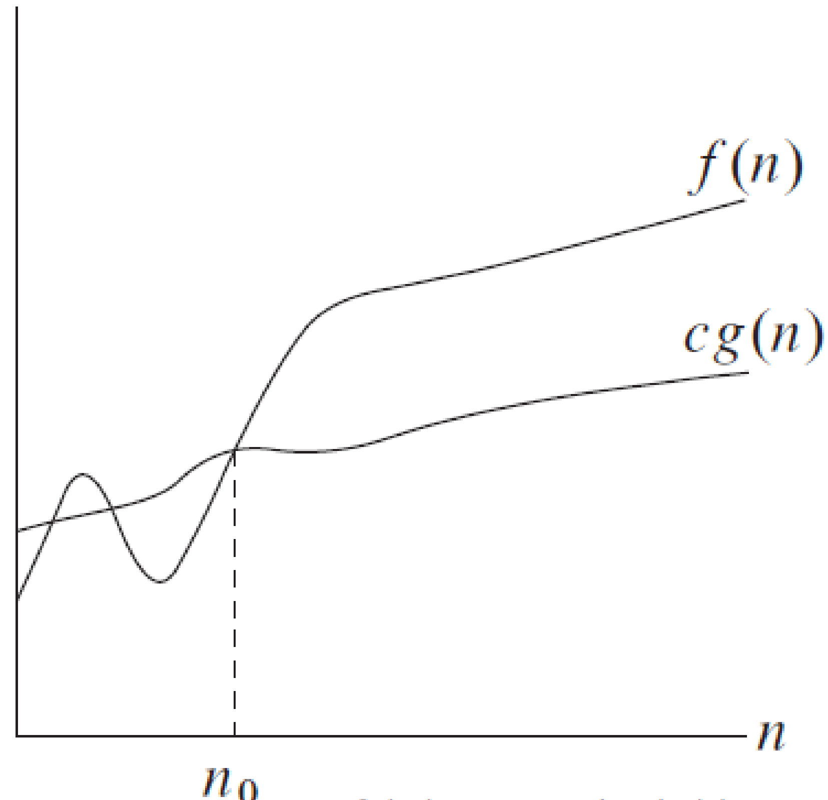
$f(n) = O(g(n))$

Ω (big-omega) Notation

- For a given function *g(n)*, we denote by $\Omega(g(n))$ the *set of functions:*

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

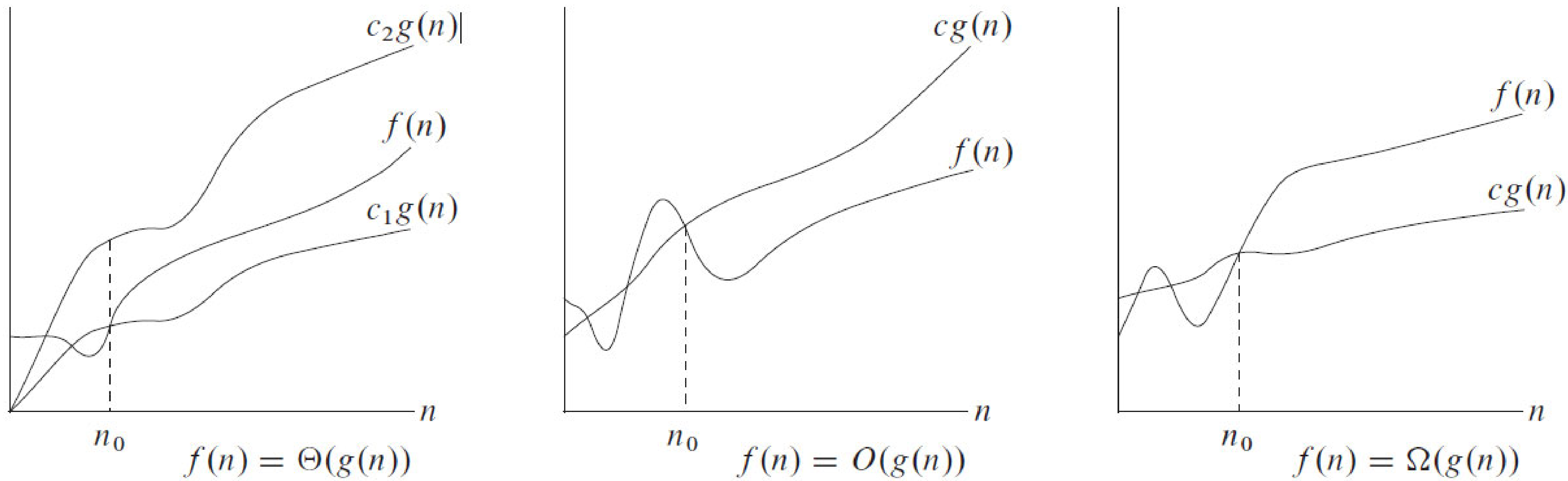
- Ω-notation gives a lower bound on a function.
- As shown $f(n)$ is above $cg(n)$, for all values of n at and to the right of n_0
- Note that $f(n) = \Theta(g(n))$ implies $f(n) = \Omega(g(n))$, since Θ-notation is a stronger notion than Ω-notation and hence: $\Theta(g(n)) \subseteq \Omega(g(n))$



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

Theorem

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



- *Note:* The running time of insertion sort therefore belongs to both $\Omega(n)$ and $O(n^2)$, since it falls anywhere between a linear function of n and a quadratic function of n.

o (little-oh) Notation

- For a given function *g(n)*, we denote by $o(g(n))$ the *set of functions:*

$$o(g(n)) = \{ f(n) : \text{for any positive constant } c > 0, \text{ there exists a constant } n_0 > 0 \text{ such that } 0 \le f(n) < cg(n) \text{ for all } n \ge n_0 \}$$

- The definitions of O-notation and o-notation are similar. The main difference is that $f(n)=O(g(n))$, the bound $0 \le f(n) \le cg(n)$ *holds* for **some** constant $c > 0$, but in $f(n)=o(g(n))$, the bound $0 \le f(n) < cg(n)$ holds for **all** constants $c > 0$.

ω (little-omega) Notation

- For a given function $g(n)$, we denote by $\omega(g(n))$ the *set of functions:*

$$\omega(g(n)) = \{f(n) : \text{for any positive constant } c > 0 \text{, there exists a constant } n_0 > 0 \text{ such that } 0 \le cg(n) < f(n) \text{ for all } n \ge n_0\}$$

- The definitions of Ω-notation and ω-notation are similar. The main difference is that $f(n)=\Omega(g(n))$, the bound $0 \le cg(n) \le f(n)$ *holds* for **some** constant $c > 0$, but in $f(n)=\omega(g(n))$, the bound $0 \le cg(n) < f(n)$ holds for **all** constants $c > 0$.