Growth of Functions

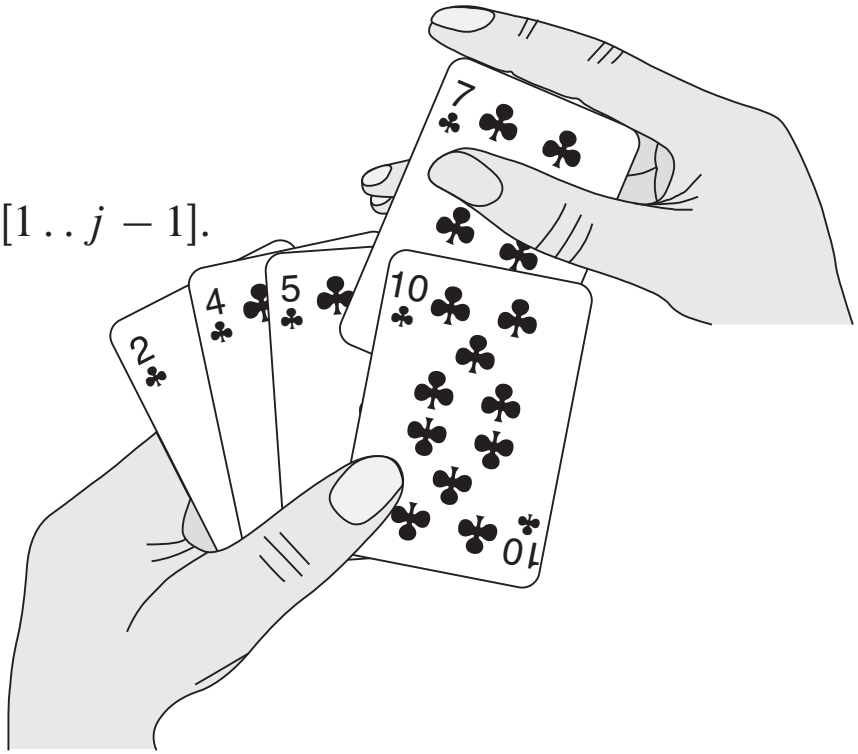
How to evaluate an algorithm?

- Metrics:
  - Correctness
  - Time efficiency: Aka time complexity, indicates how fast an algorithm runs.
  - Space efficiency: Aka space complexity, refers to the amount of memory units required in addition to that for input and output.
  - Message complexity: for distributed and/or network algorithms.
  - Processor complexity: for parallel algorithms.
- Approaches:
  - Formal proof for correctness
  - Experimental studies (run programs)
  - Asymptotic algorithm analysis of complexity

Our main focus

Correctness of Insertion Sort

```
INSERTION-SORT(A)
1  for j = 2 to A.length
2      key = A[j]
3      // Insert A[j] into the sorted sequence A[1 .. j − 1].
4      i = j − 1
5      while i > 0 and A[i] > key
6          A[i + 1] = A[i]
7          i = i − 1
8      A[i + 1] = key
```



Is it correct?

Correctness of Insertion Sort

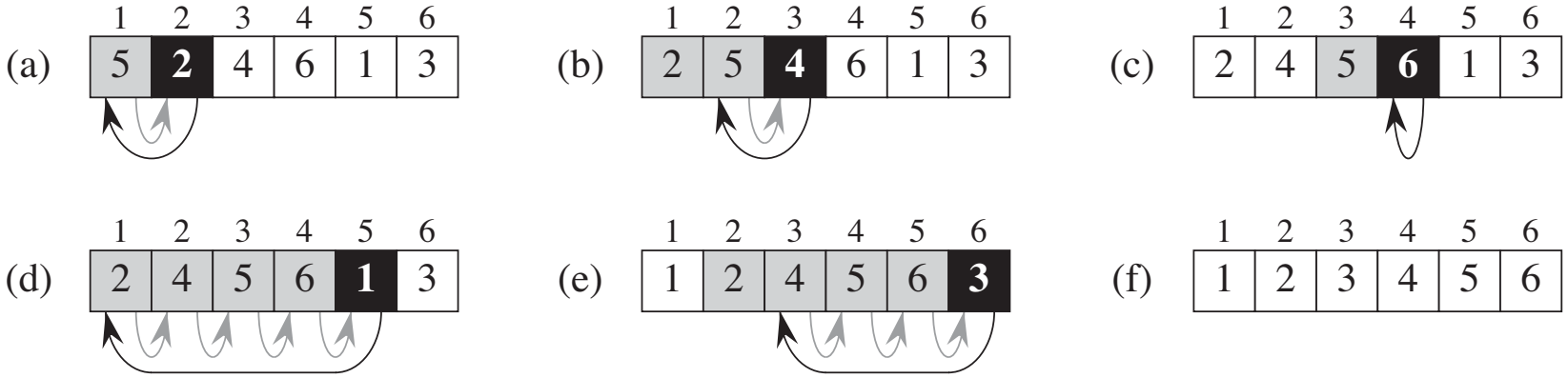


Figure 2.2 The operation of INSERTION-SORT on the array $A = \langle 5, 2, 4, 6, 1, 3 \rangle$. Array indices appear above the rectangles, and values stored in the array positions appear within the rectangles. **(a)–(e)** The iterations of the **for** loop of lines 1–8. In each iteration, the black rectangle holds the key taken from $A[j]$, which is compared with the values in shaded rectangles to its left in the test of line 5. Shaded arrows show array values moved one position to the right in line 6, and black arrows indicate where the key moves to in line 8. **(f)** The final sorted array.

Correctness of Insertion Sort

- **Loop invariant**: A statement about a loop that is true *before* the loop begins and *after each iteration* of the loop.
- Upon termination of the loop, the invariant should help you show something useful about the algorithm.

```
INSERTION-SORT(A)
1  for j = 2 to A.length
2      key = A[j]
3      // Insert A[j] into the sorted sequence A[1 .. j − 1].
4      i = j − 1
5      while i > 0 and A[i] > key
6          A[i + 1] = A[i]
7          i = i − 1
8      A[i + 1] = key
```

Loop invariant?

Correctness of Insertion Sort

- **Loop invariant**: A statement about a loop that is true *before* the loop begins and *after each iteration* of the loop.
- At the start of each iteration of the for loop of lines 1-8 the subarray $A[1..j-1]$ is the sorted version of the original elements of $A[1..j-1]$

```
INSERTION-SORT(A)
1  for j = 2 to A.length
2      key = A[j]
3      // Insert A[j] into the sorted sequence A[1 . . j − 1].
4      i = j − 1
5      while i > 0 and A[i] > key
6          A[i + 1] = A[i]
7          i = i − 1
8      A[i + 1] = key
```

Proof?

Correctness of Insertion Sort

- At the start of each iteration of the for loop of lines 1-8 the subarray $A[1..j-1]$ is the sorted version of the original elements of $A[1..j-1]$.
- Proof by induction
  - Base case: invariant is true before loop
  - Inductive case: it is true after each iteration

```
INSERTION-SORT(A)
1  for j = 2 to A.length
2      key = A[j]
3      // Insert A[j] into the sorted sequence A[1 . . j − 1].
4      i = j − 1
5      while i > 0 and A[i] > key
6          A[i + 1] = A[i]
7          i = i − 1
8      A[i + 1] = key
```

Insertion Sort

```
INSERTION-SORT(A)
1  for j = 2 to A.length
2      key = A[j]
3      // Insert A[j] into the sorted sequence A[1 .. j − 1].
4      i = j − 1
5      while i > 0 and A[i] > key
6          A[i + 1] = A[i]
7          i = i − 1
8      A[i + 1] = key
```

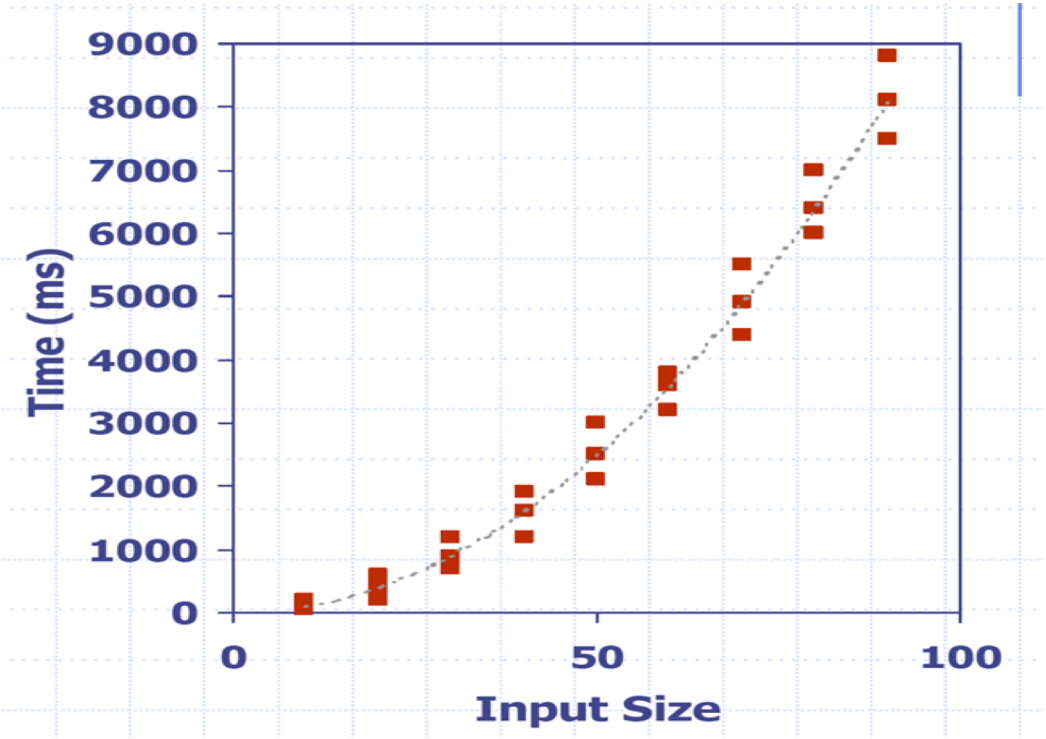
Time complexity?

(Additional) Space complexity?

Analyzing Time Complexity

Experimental studies

- Write a program implementing the algorithm
- Run the program with inputs of varying size and composition
- Use a function, like the built-in clock() function, to get an accurate measure of the actual running time
- Plot the results



Limitations of Experiments

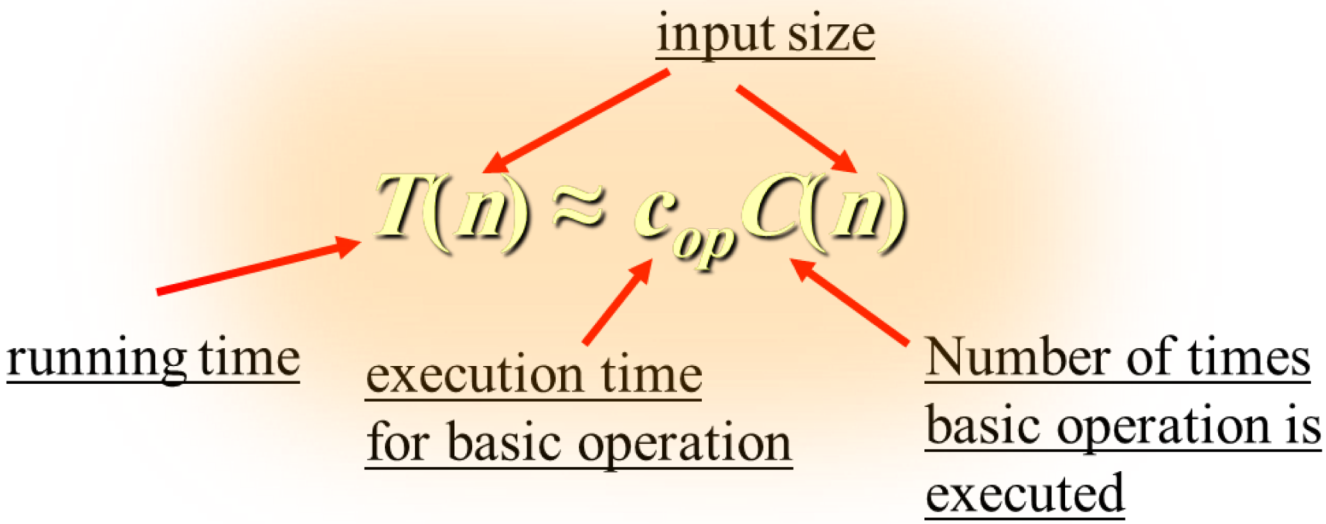
- It is necessary to implement the algorithm, which may be difficult.
- Results may not be indicative of the running time on other inputs not included in the experiment.
- In order to compare two algorithms, the same hardware and software environments must be used.

The Analysis Framework

- It is logical to investigate an algorithm's efficiency as a function of some parameter *n* indicating the input size.
  - For example: the size of the list for problems of sorting, searching, and most other problems dealing with lists.
- Single processor
- RAM model: instructions are executed one after another, with no concurrent operations.

Units for Measuring Running Time

- Time efficiency is analyzed by determining # of repetitions of the *basic operation* as a function of *input size.*
- *Basic operation*: the operation that contributes most towards the running time of the algorithm.
- We can estimate the running time T(n) by the formula:



Units for Measuring Running Time (cont.)

Input size and basic operation examples:

Problem	Input Size Measure	Basic Operation
Searching for key in a list of n items	Number of list's items, i.e. n	Key comparison
Multiplication of two matrices	Matrix dimensions or total number of elements	Multiplication of two numbers
Checking primality of a given integer n	n's size = number of digits (in binary representation)	Division
Typical graph problem	#vertices and/or edges	Visiting a vertex or traversing an edge

Units for Measuring Running Time (cont.)

- This formula should be used with caution:
  - The count C(n) does not contain any information about operations that are not basic.
    - the count itself is often computed only approximately.
  - The constant c_{op} is also an approximation which is not always easy to assess.

- Still, the formula can give a reasonable estimate of the algorithm's running time.
- It also makes it possible to answer such questions as "How much faster would this algorithm run on a machine that is 10 times faster than the one we have?"

Units for Measuring Running Time (cont.)

- Assuming that *C(n) = 1/2 n(n − 1)*, how much longer will the algorithm run if we double its input size?

$$C(n) = \frac{1}{2}n(n-1) = \frac{1}{2}n^2 - \frac{1}{2}n \approx \frac{1}{2}n^2$$

$$\frac{T(2n)}{T(n)} \approx \frac{c_{op}C(2n)}{c_{op}C(n)} \approx \frac{\frac{1}{2}(2n)^2}{\frac{1}{2}n^2} = 4.$$

- notes:
  - we were able to answer the question without actually knowing the value of c_{op}: *it was neatly cancelled out in the ratio.*
  - *1/2* , the multiplicative constant, *was also cancelled out.*
- Thus we ignore multiplicative constants and concentrate on the count's *order of growth*.

Orders of Growth

- How fast running time grows as input size increases?
- A difference in exe times on small inputs is not what really distinguishes efficient algorithms from inefficient ones.
- For large values of *n,* it is the function's order of growth that counts.
- Asymptotic analysis: The limiting behavior of the execution time of an algorithm when the size of the problem goes to infinity.

Orders of Growth (Cont.)

- The magnitude of the numbers in this table has a profound significance for the analysis of algorithms.

n	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
10	3.3	10^1	$3.3 \cdot 10^1$	10^2	10^3	10^3	$3.6 \cdot 10^6$
10^2	6.6	10^2	$6.6 \cdot 10^2$	10^4	10^6	$1.3 \cdot 10^{30}$	$9.3 \cdot 10^{157}$
10^3	10	10^3	$1.0 \cdot 10^4$	10^6	10^9		
10^4	13	10^4	$1.3 \cdot 10^5$	10^8	10^{12}		
10^5	17	10^5	$1.7 \cdot 10^6$	10^{10}	10^{15}		
10^6	20	10^6	$2.0 \cdot 10^7$	10^{12}	10^{18}		

  - The slowest among these is the logarithmic function.
  - The exponential function 2^n *and the* factorial function *$n!$ grow so fast that their values become* astronomically large even for small n.

Orders of Growth (Cont.)

- How these functions react to twofold increases in the value of their arguments n:
  - The function $\log_2 n$ *increases in* value by just 1

$$\log_2 2n = \log_2 2 + \log_2 n = 1 + \log_2 n$$

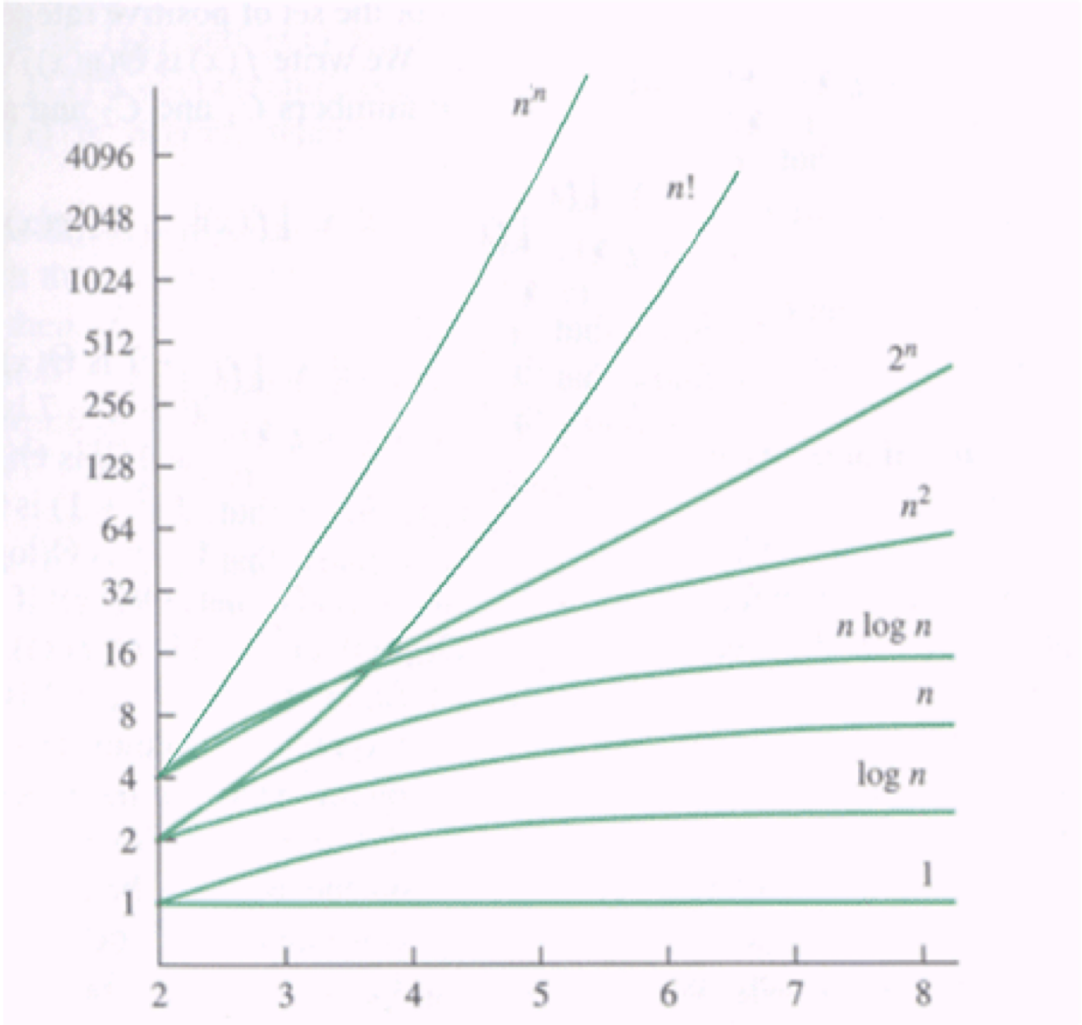
  - The linear function increases twofold
  - The linearithmic function $n \log_2 n$ *increases slightly more than* twofold.
  - The quadratic function n^2 and cubic function n^3 *increase fourfold and* eightfold, respectively

$$(2n)^2 = 4n^2 \text{ and } (2n)^3 = 8n^3$$

  - The value of 2^n gets squared $2^{2n} = (2^n)^2$; and $n!$ increases much more than that.

Orders of Growth (Cont.)

C - constant
log n - logarithmic
$\log^2 n$ - Log-squared
$\log^k n$ - Poly-logarithmic
n - Linear
n^2 - Quadratic
n^3 - Cubic
2^n - Exponential



We mostly use the above classes of growth functions.

Orders of Growth (Cont.)

- Example:
  - Order the following functions by asymptotic growth rate from least to greatest:

$4n\log n+2n$, 2^{10}, $3n+100\log n$, $4n$, 2^n, n^2+10n, n^3, $n\log n$

Orders of Growth (Cont.)

- Example:
  - Order the following functions by asymptotic growth rate from least to greatest:

$4n\log n+2n,\quad 2^{10},\quad 3n+100\log n,\quad 4n,\quad 2^n,\quad n^2+10n,\quad n^3,\quad n\log n$

- Solution:

2^{10} , $3n+100\log n$, $4n$, $n\log n$, $4n\log n+2n$, n^2+10n, n^3, 2^n

Types of Analysis

- For input size n, three types of analysis:
  - Worst-Case Analysis: time efficiency of input that results in the longest running time for given input size.
    - Worst case: $C_{worst}(n)$ – maximum over inputs of size n
  - Best-Case Analysis: time efficiency of input that results in fastest running time for given input size.
    - Best case: $C_{best}(n)$ – minimum over inputs of size n
  - Average-Case Analysis: "typical" time efficiency of input of given input size.
    - not average of best and worst case.
    - Average case: $C_{avg}(n)$ – "average" over inputs of size n

Example: Sequential Search

ALGORITHM *SequentialSearch*$(A[0..n-1], K)$

```
//Searches for a given value in a given array by sequential search
//Input: An array A[0..n − 1] and a search key K
//Output: The index of the first element in A that matches K
//              or −1 if there are no matching elements
i ← 0
while i < n and A[i] ≠ K do
    i ← i + 1
if i < n return i
else return −1
```

- Time Efficiencies:
  - Worst-Case?
  - Best-Case?
  - Average-Case?

Asymptotic notations

- Asymptotic Order of Growth: To compare and rank orders of growth, computer scientists use three notations.

  - $O(g(n))$: class of functions $f(n)$ that grow no faster than $g(n)$.
    Knuth traces the origin of O in number theory text by Bachman in 1892 and in calculus.

  - $\Theta(g(n))$: class of functions $f(n)$ that grow at same rate as $g(n)$

  - $\Omega(g(n))$: class of functions $f(n)$ that grow at least as fast as $g(n)$

O (big oh)

- Informally, *O(g(n)) is the set of all functions with a lower or same order of growth* as *g(n) (to within a constant multiple, as n goes to infinity).*

- Thus, to give a few examples, the following assertions are all true:

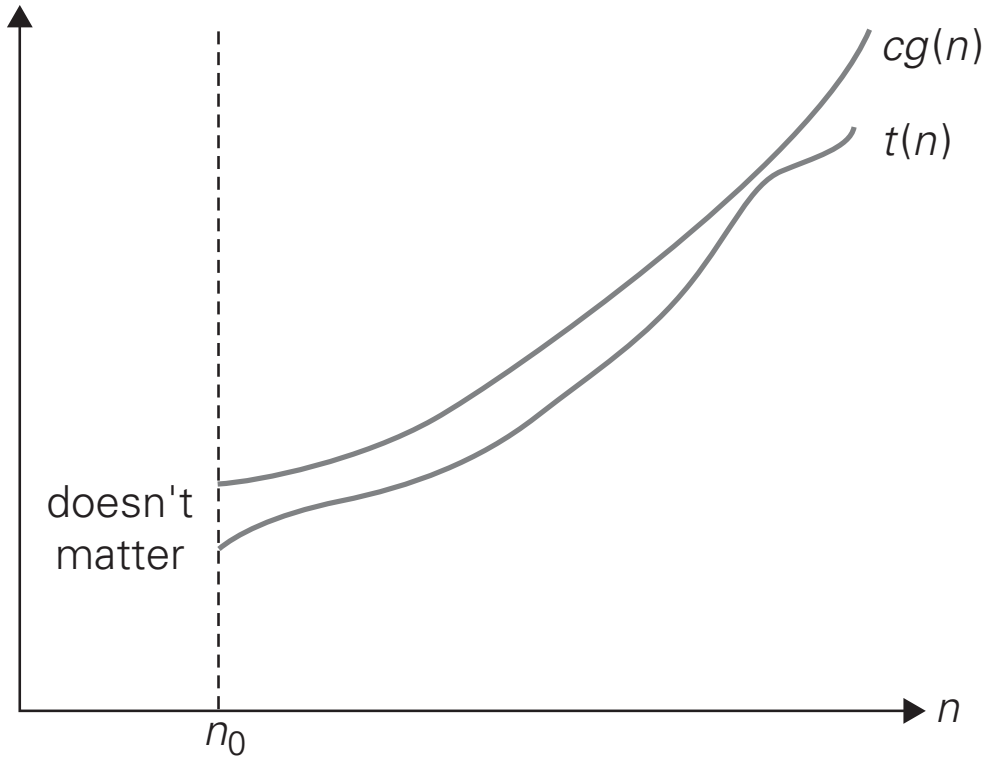
$$n \in O(n^2), \qquad 100n + 5 \in O(n^2), \qquad \frac{1}{2}n(n-1) \in O(n^2).$$

$$n^3 \notin O(n^2), \qquad 0.00001n^3 \notin O(n^2), \qquad n^4 + n + 1 \notin O(n^2).$$

O (big oh) [cont.]

DEFINITION A function $t(n)$ is said to be in $O(g(n))$, denoted $t(n) \in O(g(n))$, if $t(n)$ is bounded above by some constant multiple of $g(n)$ for all large n, i.e., if there exist some positive constant c and some nonnegative integer n_0 such that

$$t(n) \leq cg(n) \quad \text{for all } n \geq n_0.$$



O (big oh) [cont.]

- $t(n) = O(g(n))$ if there are positive constants c and n_0 such that $t(n) \leq cg(n)$ when $n \geq n_0$.
- upper bound on $t(n)$.

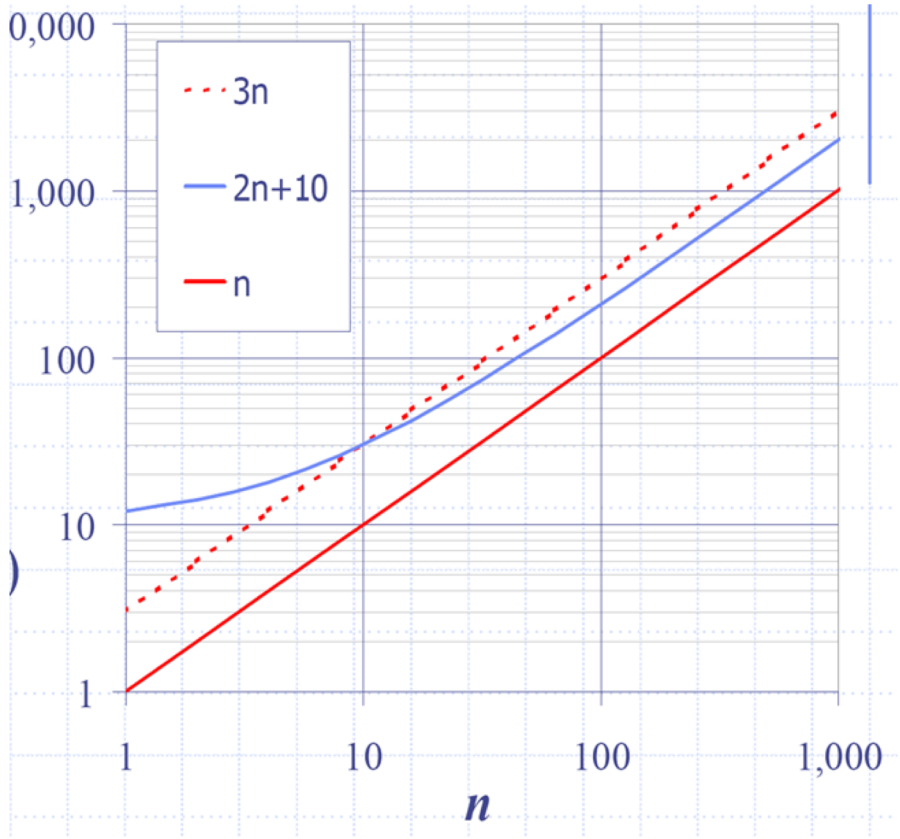
Example: $2n + 10 = O(n)$

$2n + 10 \leq cn$

$(c - 2)\, n \geq 10$

$n \geq 10/(c - 2)$

Pick $c = 3$ and $n_0 = 10$



O (big oh) (cont.)

- $7n-2$

  $7n-2 = O(n)$

  need $c > 0$ and $n_0 \geq 1$ such that $7n-2 \leq c \bullet n$ for $n \geq n_0$

  this is true for $c = 7$ and $n_0 = 1$

- $3n^3 + 20n^2 + 5$

  $3n^3 + 20n^2 + 5 = O(n^3)$

  need $c > 0$ and $n_0 \geq 1$ such that $3n^3 + 20n^2 + 5 \leq c \bullet n^3$ for $n \geq n_0$

  this is true for $c = 4$ and $n_0 = 21$

- $3 \log n + \log \log n$

  $3 \log n + \log \log n = O(\log n)$

  need $c > 0$ and $n_0 \geq 1$ such that $3 \log n + \log \log n \leq c \bullet \log n$ for $n \geq n_0$

  this is true for $c = 4$ and $n_0 = 2$

O (big oh) (cont.)

If $t(n) = 2n^2$ then:

$t(n) = O(n^4)$

$t(n) = O(n^3)$

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

Technically all of them are correct,

But $t(n) = O(n^2)$ is the best answer!

O (big oh) (cont.)

We don't say:

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

Or

$$t(n) = O(\ n^2 + n)$$

The correct form is :

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

Lower order terms can be ignored and constants can be thrown away. Why?

In other words, what is asymptotic analysis?

Ω(big omega)

- Informally, *Ω(g(n)),* stands for the set of all functions with a higher or same order of growth as *g(n) (*to within a constant multiple, as n goes to infinity).

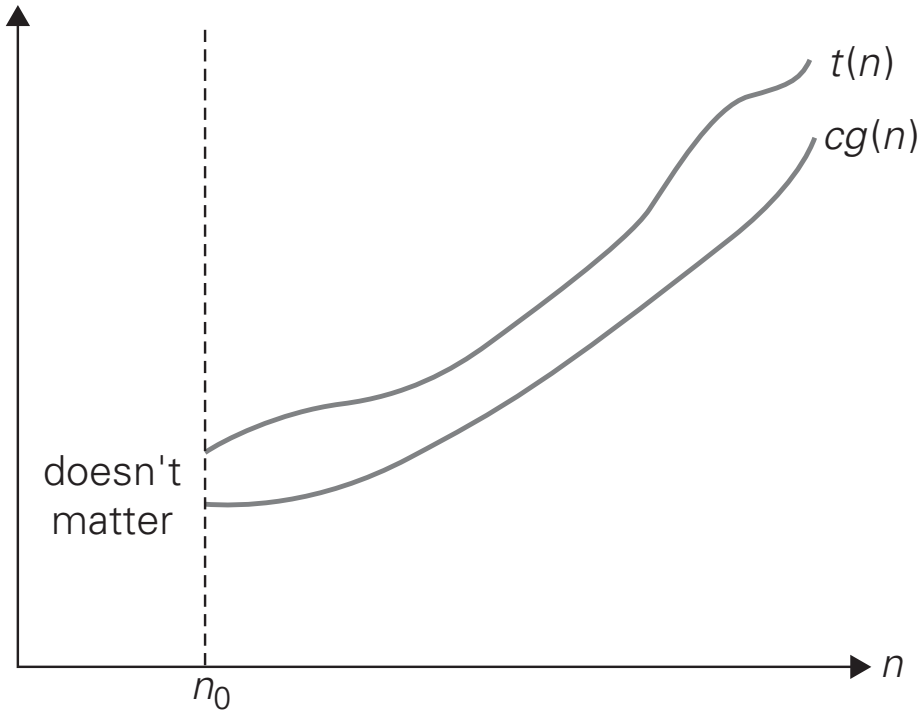
- For example,

$$n^3 \in \Omega(n^2), \qquad \frac{1}{2}n(n-1) \in \Omega(n^2), \qquad \text{but } 100n + 5 \notin \Omega(n^2).$$

Ω(big omega) (cont.)

DEFINITION A function $t(n)$ is said to be in $\Omega(g(n))$, denoted $t(n) \in \Omega(g(n))$, if $t(n)$ is bounded below by some positive constant multiple of $g(n)$ for all large n, i.e., if there exist some positive constant c and some nonnegative integer n_0 such that

$$t(n) \geq cg(n) \quad \text{for all } n \geq n_0.$$



Ω (big omega) (cont.)

Here is an example of the formal proof that $n^3 \in \Omega(n^2)$:

$$n^3 \geq n^2 \quad \text{for all } n \geq 0,$$

i.e., we can select $c = 1$ and $n_0 = 0$.

Ω(big omega) (cont.)

- When we say that $t(n) = O(g(n))$, we are guaranteeing that the function $t(n)$ grows at a rate no faster than $g(n)$

$g(n)$ is *an upper bound* on $t(n)$.

- Since this implies that $g(n) = \Omega(t(n))$, we say that $t(n)$ is *a lower bound* on $g(n)$

Ω(big omega) (cont.)

Example1:

n^3 *grows faster than* n^2 , so we can say:

$$n^2 = O(n^3)$$

or

$$n^3 = \Omega(n^2)$$

Example2:

$t(n)=n^2$ and $g(n)=2n^2$ *grow at the same rate*, so we can say:

$$t(n)= O(g(n))$$

and

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

Θ (big theta)

- Informally, Θ*(g(n))* is the set of all functions that have the same order of growth as *g(n) (to within a constant multiple, as n goes to infinity)*.

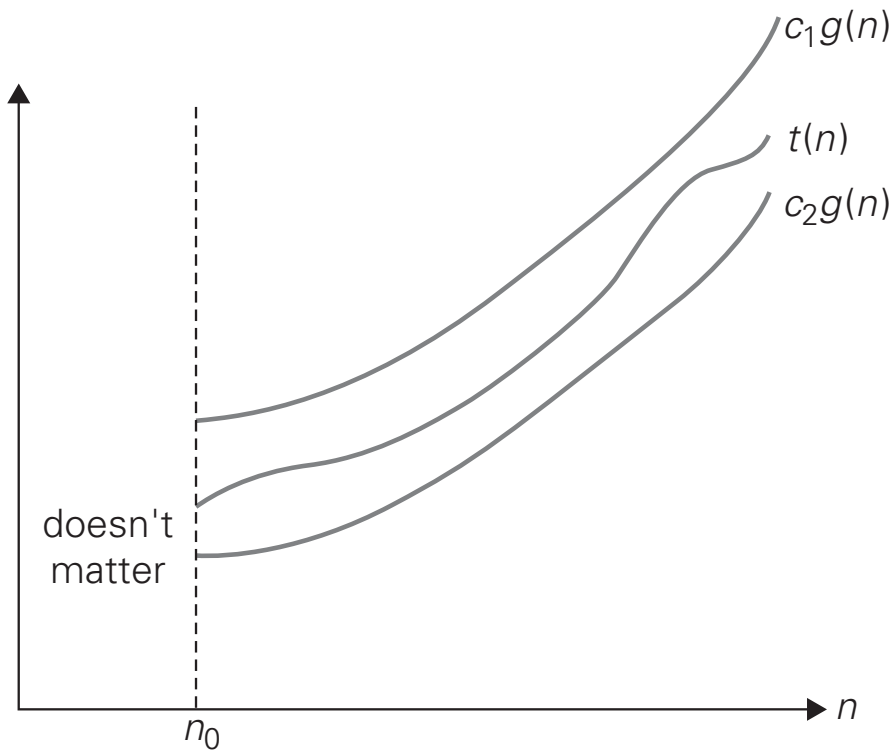
- For example,

$$an^2 + bn + c \text{ with } a > 0 \text{ is in } \Theta(n^2)$$

Θ (big theta) (cont.)

DEFINITION A function $t(n)$ is said to be in $\Theta(g(n))$, denoted $t(n) \in \Theta(g(n))$, if $t(n)$ is bounded both above and below by some positive constant multiples of $g(n)$ for all large n, i.e., if there exist some positive constants c_1 and c_2 and some nonnegative integer n_0 such that

$$c_2 g(n) \leq t(n) \leq c_1 g(n) \quad \text{for all } n \geq n_0.$$



Θ (big theta) (cont.)

For example, let us prove that $\frac{1}{2}n(n-1) \in \Theta(n^2)$. First, we prove the right inequality (the upper bound):

$$\frac{1}{2}n(n-1) = \frac{1}{2}n^2 - \frac{1}{2}n \leq \frac{1}{2}n^2 \quad \text{for all } n \geq 0.$$

Second, we prove the left inequality (the lower bound):

$$\frac{1}{2}n(n-1) = \frac{1}{2}n^2 - \frac{1}{2}n \geq \frac{1}{2}n^2 - \frac{1}{2}n\frac{1}{2}n \text{ (for all } n \geq 2) = \frac{1}{4}n^2.$$

Hence, we can select $c_2 = \frac{1}{4}$, $c_1 = \frac{1}{2}$, and $n_0 = 2$.

Useful Property Involving the Asymptotic notations

- The following property, in particular, is useful in analyzing algorithms that comprise two consecutively executed parts.

THEOREM If $t_1(n) \in O(g_1(n))$ and $t_2(n) \in O(g_2(n))$, then

$$t_1(n) + t_2(n) \in O(\max\{g_1(n), g_2(n)\}).$$

(The analogous assertions are true for the Ω and Θ notations as well.)

Useful Property Involving the Asymptotic notations

- For example, we can check whether an array has equal elements by the following two-part algorithm:
  - first, sort the array by applying some known sorting algorithm;
  - second, scan the sorted array to check its consecutive elements for equality.
- If a sorting algorithm used in the first part makes no more than $1/2\ n(n-1)$ comparisons (and hence is in $O(n^2)$)
- *The second part makes no more than $n-1$ comparisons (and hence is in $O(n)$).*
- *The efficiency of the entire algorithm* will be in

$$O(\max\{n^2, n\}) = O(n^2)$$

Using Limits for Comparing Orders of Growth

- A much more convenient method for comparing the orders of growth of two specific functions is based on computing the limit of the ratio of two functions in question. Three principal cases may arise:

$$\lim_{n\to\infty} \frac{t(n)}{g(n)} = \begin{cases} 0 & \text{implies that } t(n) \text{ has a smaller order of growth than } g(n), \\ c & \text{implies that } t(n) \text{ has the same order of growth as } g(n), \\ \infty & \text{implies that } t(n) \text{ has a larger order of growth than } g(n). \end{cases}$$

- note that the first two cases mean that *$t(n) \in O(g(n))$, the last two mean that $t(n) \in \Omega(g(n))$, and the second case means that $t(n) \in \Theta(g(n))$.*

Using Limits for Comparing Orders of Growth (cont.)

The limit-based approach is often more convenient than the one based on the definitions because it can take advantage of the powerful calculus techniques developed for computing limits, such as L'Hôpital's rule

$$\lim_{n\to\infty} \frac{t(n)}{g(n)} = \lim_{n\to\infty} \frac{t'(n)}{g'(n)}$$

and Stirling's formula

$$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \quad \text{for large values of } n.$$

Using Limits for Comparing Orders of Growth (cont.)

$$\lim_{n\to\infty} \frac{\frac{1}{2}n(n-1)}{n^2} = \frac{1}{2}\lim_{n\to\infty} \frac{n^2-n}{n^2} = \frac{1}{2}\lim_{n\to\infty}(1-\frac{1}{n}) = \frac{1}{2}.$$

$$\lim_{n\to\infty} \frac{\log_2 n}{\sqrt{n}} = \lim_{n\to\infty} \frac{(\log_2 n)'}{(\sqrt{n})'} = \lim_{n\to\infty} \frac{(\log_2 e)\frac{1}{n}}{\frac{1}{2\sqrt{n}}} = 2\log_2 e \lim_{n\to\infty} \frac{1}{\sqrt{n}} = 0.$$

$$\lim_{n\to\infty} \frac{n!}{2^n} = \lim_{n\to\infty} \frac{\sqrt{2\pi n}\left(\frac{n}{e}\right)^n}{2^n} = \lim_{n\to\infty} \sqrt{2\pi n}\frac{n^n}{2^n e^n} = \lim_{n\to\infty} \sqrt{2\pi n}\left(\frac{n}{2e}\right)^n = \infty.$$

Mathematical Analysis of nonrecursive Algorithms

EXAMPLE 1 Consider the problem of finding the value of the largest element in a list of n numbers. For simplicity, we assume that the list is implemented as an array. The following is pseudocode of a standard algorithm for solving the problem.

ALGORITHM *MaxElement*$(A[0..n-1])$

//Determines the value of the largest element in a given array
//Input: An array $A[0..n-1]$ of real numbers
//Output: The value of the largest element in A
$maxval \leftarrow A[0]$
for $i \leftarrow 1$ **to** $n-1$ **do**
    if $A[i] > maxval$
        $maxval \leftarrow A[i]$
return $maxval$

Mathematical Analysis of nonrecursive Algorithms [cont.]

- Note that the number of comparisons will be the same for all arrays of size n; *therefore, in terms of this metric, there is no need to* distinguish among the worst, average, and best cases here.

$$C(n) = \sum_{i=1}^{n-1} 1.$$

This is an easy sum to compute because it is nothing other than 1 repeated $n - 1$ times. Thus,

$$C(n) = \sum_{i=1}^{n-1} 1 = n - 1 \in \Theta(n).$$ ■

Mathematical Analysis of nonrecursive Algorithms [cont.]

EXAMPLE 2 Consider the ***element uniqueness problem***: check whether all the elements in a given array of n elements are distinct. This problem can be solved by the following straightforward algorithm.

ALGORITHM *UniqueElements*$(A[0..n-1])$

```
//Determines whether all the elements in a given array are distinct
//Input: An array A[0..n − 1]
//Output: Returns "true" if all the elements in A are distinct
//                and "false" otherwise
for i ← 0 to n − 2 do
    for j ← i + 1 to n − 1 do
        if A[i] = A[j] return false
return true
```

Mathematical Analysis of nonrecursive Algorithms [cont.]

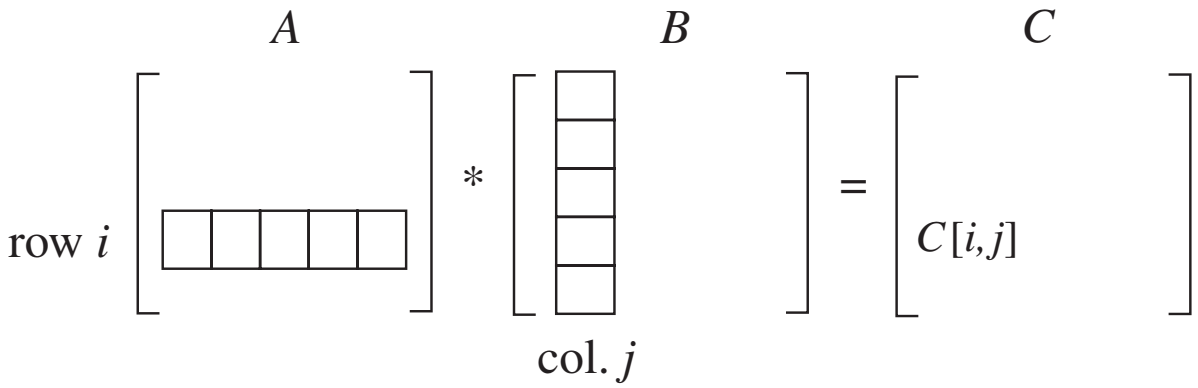
$$C_{worst}(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1] = \sum_{i=0}^{n-2} (n-1-i)$$

$$= \sum_{i=0}^{n-2} (n-1) - \sum_{i=0}^{n-2} i = (n-1) \sum_{i=0}^{n-2} 1 - \frac{(n-2)(n-1)}{2}$$

$$= (n-1)^2 - \frac{(n-2)(n-1)}{2} = \frac{(n-1)n}{2} \approx \frac{1}{2} n^2 \in \Theta(n^2).$$

Mathematical Analysis of nonrecursive Algorithms [cont.]

EXAMPLE 3 Given two $n \times n$ matrices A and B, find the time efficiency of the definition-based algorithm for computing their product $C = AB$. By definition, C is an $n \times n$ matrix whose elements are computed as the scalar (dot) products of the rows of matrix A and the columns of matrix B:



where $C[i, j] = A[i, 0]B[0, j] + \cdots + A[i, k]B[k, j] + \cdots + A[i, n-1]B[n-1, j]$ for every pair of indices $0 \le i, j \le n-1$.

Mathematical Analysis of nonrecursive Algorithms [cont.]

ALGORITHM *MatrixMultiplication*$(A[0..n-1, 0..n-1], B[0..n-1, 0..n-1])$

//Multiplies two square matrices of order n by the definition-based algorithm
//Input: Two $n \times n$ matrices A and B
//Output: Matrix $C = AB$
for $i \leftarrow 0$ **to** $n-1$ **do**
    for $j \leftarrow 0$ **to** $n-1$ **do**
        $C[i, j] \leftarrow 0.0$
        for $k \leftarrow 0$ **to** $n-1$ **do**
            $C[i, j] \leftarrow C[i, j] + A[i, k] * B[k, j]$
return C

Mathematical Analysis of nonrecursive Algorithms [cont.]

- Let us set up a sum for the total number of multiplications *M(n) executed by the algorithm.*
- *Since this* count depends only on the size of the input matrices, we do not have to investigate the worst-case, average-case, and best-case efficiencies separately.
- The number of multiplications made for every pair of specific values of variables *i and j is*

$$\sum_{k=0}^{n-1} 1,$$

- The total number of multiplications *M(n) is expressed by the following* triple sum:

$$M(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} 1 = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} n = \sum_{i=0}^{n-1} n^2 = n^3.$$