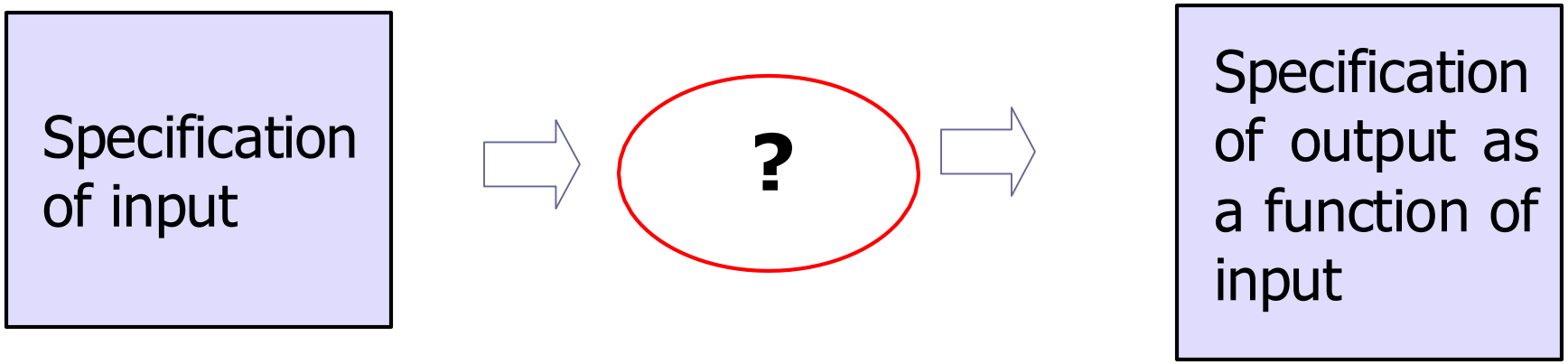
Analysis of Algorithms

- Motivations
  - primitive operations
  - Best/Worst/Average Case
- Asymptotic Analysis
  - The Big O notation
  - Big Omega and Big Theta
  - Rules
- Analyzing insertion sort

Yanlin Zhang & Wei Wang | DSAA 2043 Spring 2025

Motivations

Algorithmic Problem



- Infinite number of input *instances* satisfying the specification.
  - E.g., a sorted, non-decreasing sequence of natural numbers of non-zero, finite length:
    - 1, 20, 908, 909, 100000, 1000000000
    - 3

Algorithmic Solution

- Algorithm describes actions on the input instance
- Many correct algorithms for the same algorithmic problem

What is a Good Algorithm?

- Efficient
  - Running time
  - Space used

- Efficiency as a function of input size
  - The number of bits in an input number
  - Number of data elements (numbers, points)

What is Analysis of Algorithms

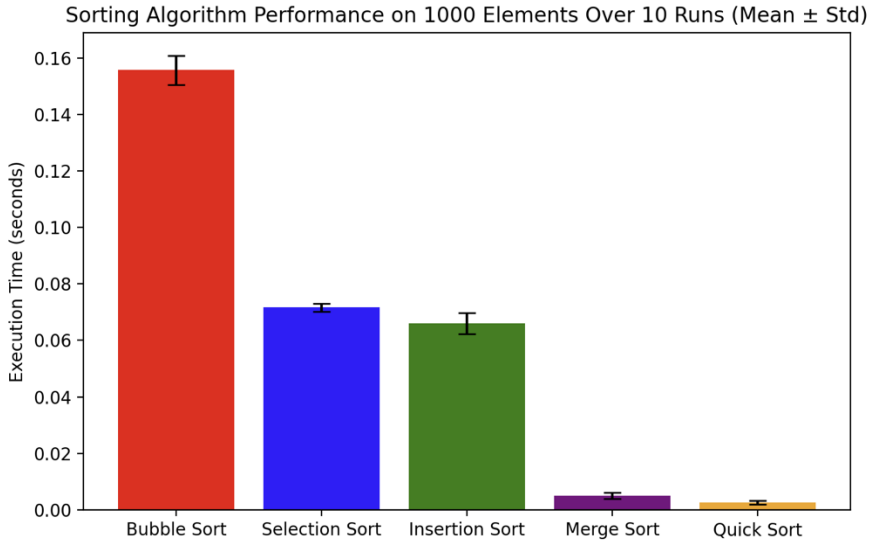
Estimate the running time.

Estimate the memory space required.

Time and space depend on the input size.

Measuring the Running Time Experimentally

How should we measure the running time of an algorithm?



Experimental Study

- Write a program that implements the algorithm
- Run the program with data sets of varying size and composition
- Use a system call to get an accurate measure of the actual running time

Limitations of Experimental Studies

- Must implement and test the algorithm to determine its running time
- Experiments done only on a limited set of inputs
  - May not be indicative of the running time on other inputs not included in the experiment
- To compare two algorithms, the same hardware and software environments needed

Beyond Experimental Studies

香港科技大学(广州)
THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY (GUANGZHOU)

- We will develop a general methodology for analyzing running time of algorithms. This approach
  - Uses a high-level description (pseudocode) of the algorithm instead of testing one of its implementations
  - Considers all possible inputs
  - Evaluates the efficiency of any algorithm being independent of the hardware and software environment
- To achieve that, we need to
  - Make simplifying assumptions about the running time of each basic (primitive) operations
  - Study how the number of primitive operations depends on the size of the problem solved

Primitive Operations

Simple computer operation that can be performed in time that is always the same, independent of the size of the bigger problem solved (we say: constant time)

- **Assigning a value to a variable:** x ←1 — T_{assign}
- **Calling a method:** Expos.addWin() — T_{call}
  - Note: doesn't include the time to execute the method
- **Returning from a method**: return x; — T_{return}
- **Arithmetic operations on primitive types** — T_{arith}

  x + y, r*3.1416, x/y, etc.
- **Comparisons on primitive types:** x==y — T_{comp}
- **Conditionals:** if (...) then.. else... — T_{cond}
- **Indexing into an array:** A[i] — T_{index}
- **Following object reference:** Expos.losses — T_{ref}

Note: Multiplying two Large Integers is *not* a primitive operation, because the running time depends on the size of the numbers multiplied.

More assumptions

- Counting each type of primitive operations is tedious
- The running time of each operation is roughly comparable:

$$T_{assign} \approx T_{comp} \approx T_{arith} \approx \ldots \approx T_{index} = 1 \text{ primitive operation}$$

- We are only interested in the **number** of primitive operations performed

Estimating Running Time for FindMin

```
Algorithm findMin(A, start, stop)
Input: Array A, index start & stop
Output: Index of the smallest element of A[start:stop]
minvalue ← A[start]
minindex ← start
index ← start + 1
while ( index <= stop ) do {
   if (A[index]<minvalue)
   then {
        minvalue ← A[index]
        minindex ← index
   }
   index = index + 1
}
return minindex
```

Running time	
$T_{index} + T_{assign}$	
T_{assign}	
$T_{arith} + T_{assign}$	
$T_{comp} + T_{cond}$	repeated stop-start times
$T_{index} + T_{comp} + T_{cond}$	
$T_{index} + T_{assign}$	
T_{assign}	
$T_{assign} + T_{arith}$	
$T_{comp} + T_{cond}$ (last check of loop)	
T_{return}	

Estimating Running Time for FindMin

- Running time depends on *n* = stop – start + 1
- $T(n) = 8 + 10 * (n-1)$ primitive operations
  - 8 primitive operations outside the loop
  - 10 primitive operations inside the loop
- How will the running time change for the following two input instances?

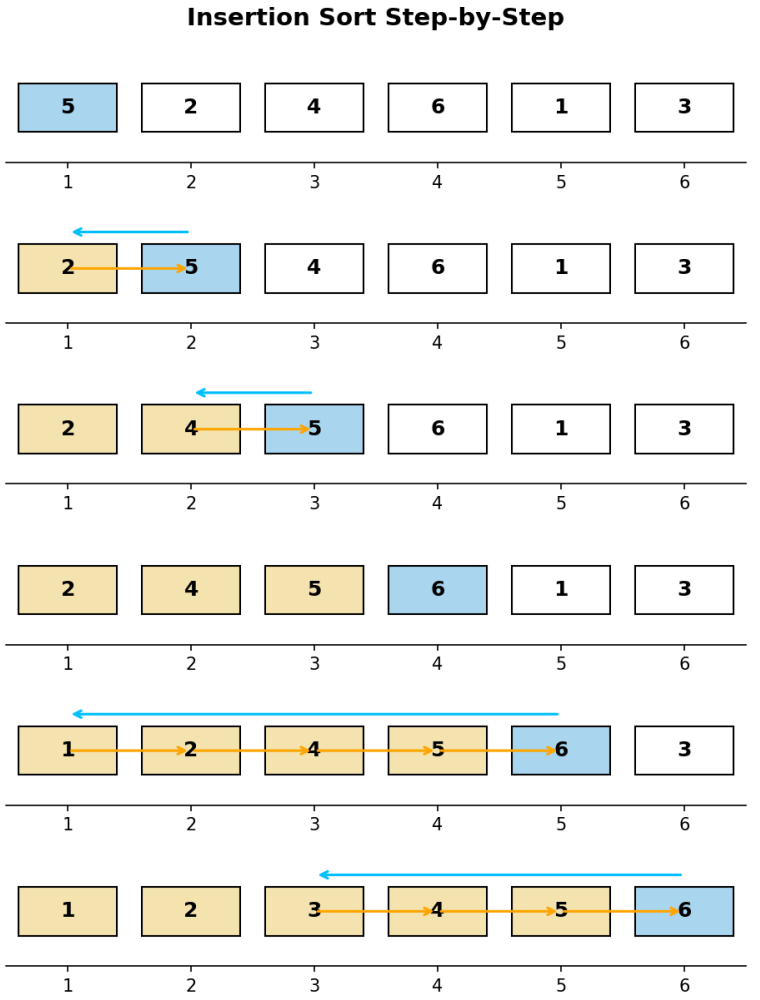
5	4	3	2	1	0

0	1	2	3	4	5

Insertion Sort (Recap.)


```
Algorithm InsertionSort(A)
Input: An array A with n elements
Output: A sorted in non-decreasing order

1. for i ← 2 to length(A) do
2.     key ← A[i]  // Current element to be inserted
3.     j ← i - 1   // Start comparing from the previous element
4.     while j > 0 and A[j] > key do
5.         A[j + 1] ← A[j]  // Shift element to the right
6.         j ← j - 1
7.     end while
8.     A[j + 1] ← key  // Place key in the correct position
9. end for
10. return A  // Sorted array
```



Insertion Sort

INSERTION-SORT(A, n)	*cost*	*times*
1 **for** $i = 2$ **to** n	c_1	n
2 $key = A[i]$	c_2	$n-1$
3 // Insert $A[i]$ into the sorted subarray $A[1:i-1]$.	0	$n-1$
4 $j = i-1$	c_4	$n-1$
5 **while** $j > 0$ and $A[j] > key$	c_5	$\sum_{i=2}^{n} t_i$
6 $A[j+1] = A[j]$	c_6	$\sum_{i=2}^{n}(t_i - 1)$
7 $j = j-1$	c_7	$\sum_{i=2}^{n}(t_i - 1)$
8 $A[j+1] = key$	c_8	$n-1$

Total time $T(n) = n(c_1 + c_2 + c_4 + c_8 - c_6 - c_7) + \sum_{i=2}^{n} t_i(c_5 + c_6 + c_7)$

$$-(c_2 + c_4 + c_6 - c_7 - c_8)$$

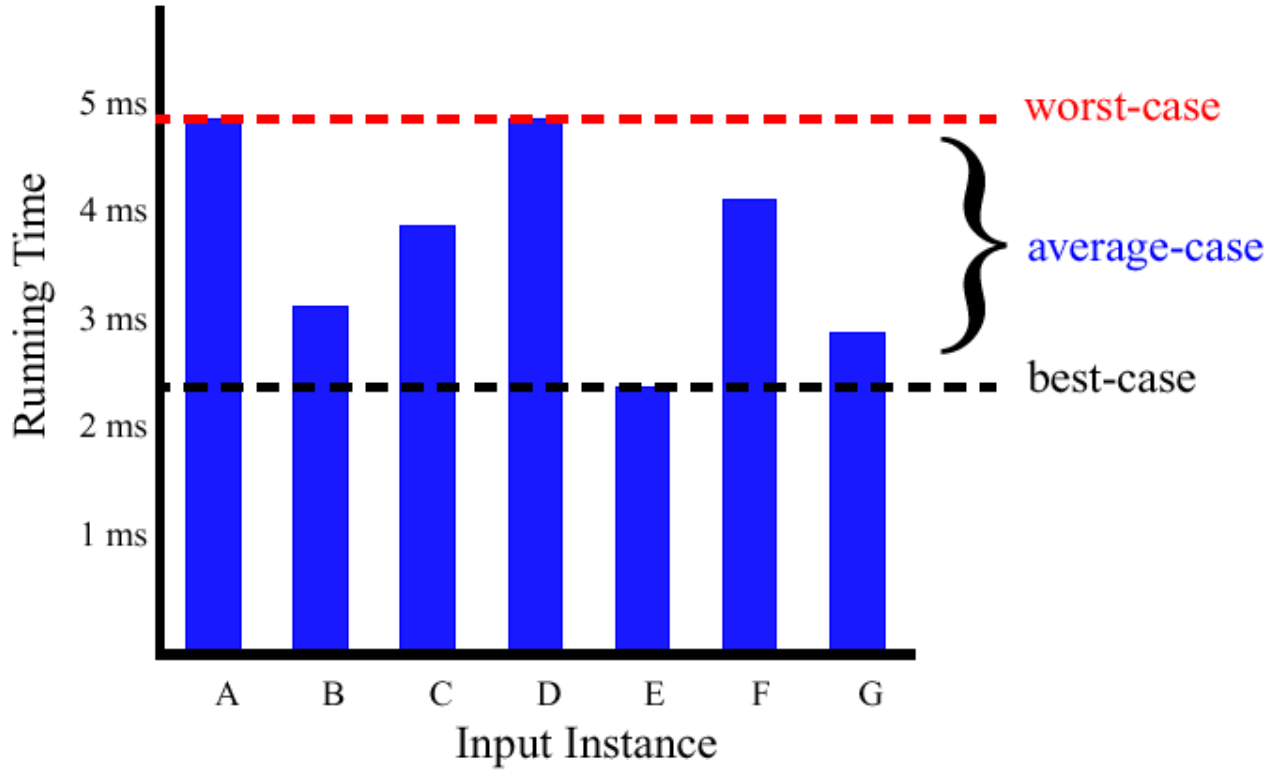
Best/Worst/Average Case

$$\text{Total time } T(n) = n(c_1 + c_2 + c_4 + c_8 - c_6 - c_7) + \sum_{i=2}^{n} t_i(c_5 + c_6 + c_7)$$
$$-(c_2 + c_4 + c_6 - c_7 - c_8)$$

- Best case:
  - elements are already sorted; Each element only does one comparison (t_i=1), running time = f(n), i.e., linear time
- Worst case:
  - elements are sorted in reverse order; each element shifts i-1 times (t_i=i-1), running time = f(n^2), i.e., quadratic time
- Average case:
  - Each element moves above half of its maximum shifts (t_i=(i-1)/2), running time = f(n^2), i.e., quadratic time

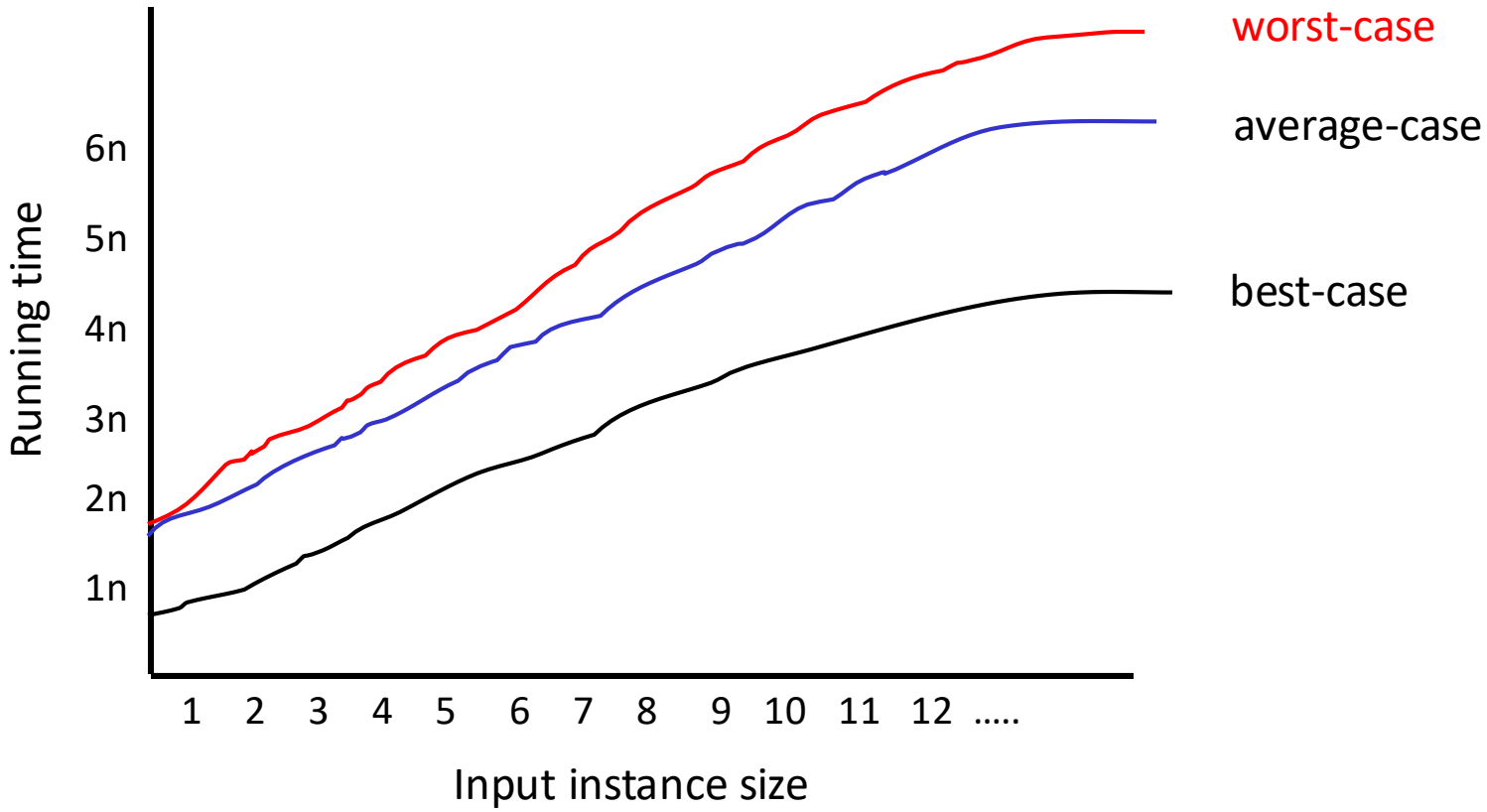
Best/Worst/Average Case

- For a specific size of input n, investigate running times for different input instances:



Best/Worst/Average Case

- For inputs of all sizes:



Best/Worst/Average Case

- Worst case is usually used
  - It is an upper-bound and in certain application domains (e.g., air traffic control, surgery) knowing the worst-case time complexity is of crucial importance
  - For some algorithms worst case occurs fairly often
  - Average case is often as bad as worst case
- Finding average case can be very difficult
- The best (fastest) case is seldom of interest

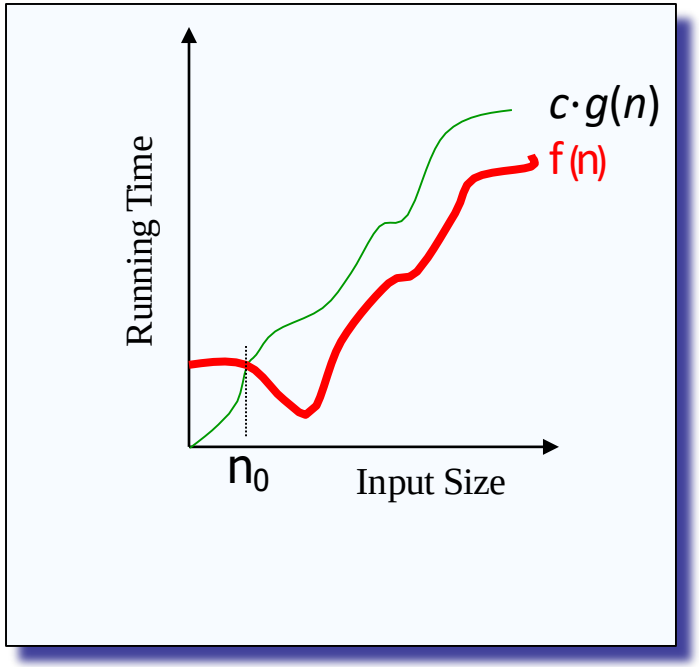
Asymptotic Analysis

Asymptotic Analysis

- Goal: to simplify analysis of running time by getting rid of "details", which may be affected by specific implementation and hardware
  - like "rounding": 1,000,001 ≈ 1,000,000
  - $3n^2 \approx n^2$
- Capturing the essence: how the running time of an algorithm increases with the size of the input in the limit
  - Asymptotically more efficient algorithms are best for all but small inputs

Asymptotic Notation

- The "Big-Oh" *O*-Notation
  - asymptotic upper bound
  - f(n) is O(g(n)), if there exists constants *c* and n_0, *s.t.* **f(n) ≤ c·g(n)** for all n ≥ n_0
    → $t(n)$ is asymptotically bounded above by $g(n)$
  - f(n) and g(n) are functions over non-negative integers
    - We usually assume both f(n) and g(n) are non-negative too
- Used for worst-case analysis

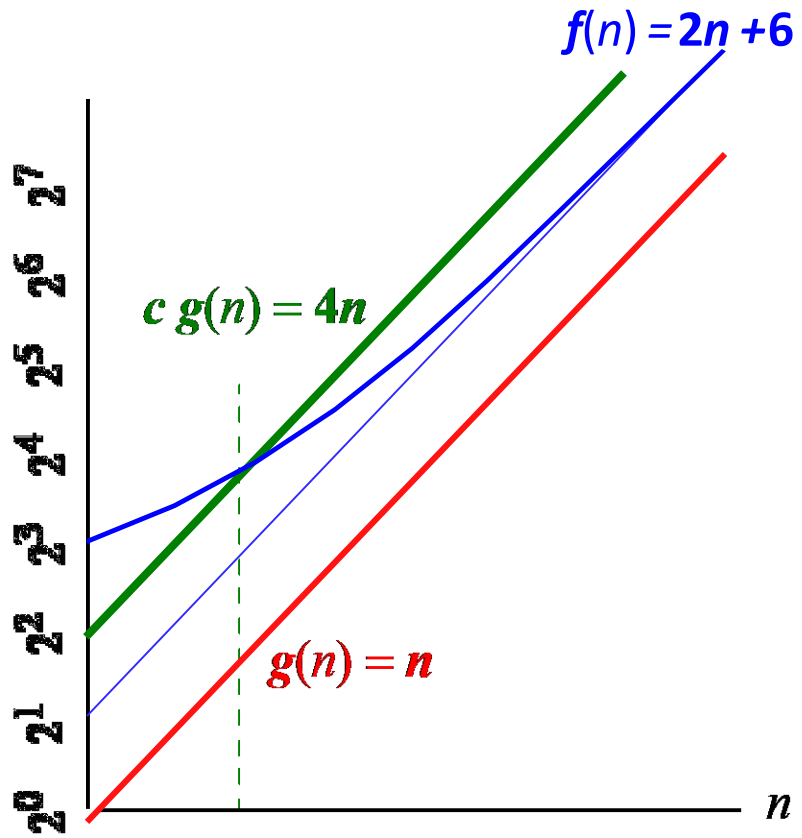


Example

For functions f(n) and g(n) there are positive constants c and n_0 such that: $f(n) \leq c\, g(n)$ for $n \geq n_0$

conclusion:

2n+6 is O(n)



Another Example

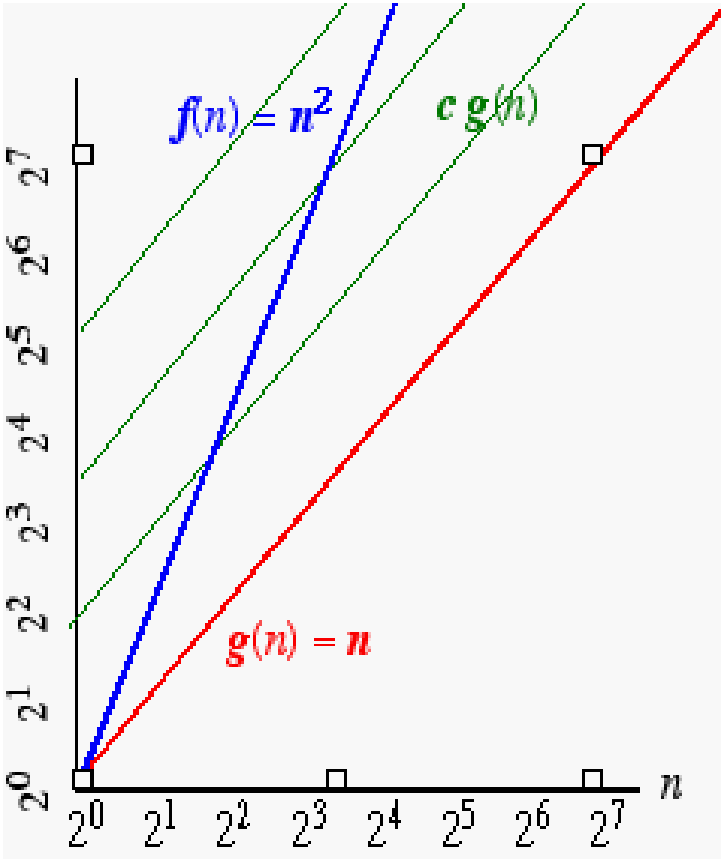
香港科技大学(广州)
THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY (GUANGZHOU)

On the other hand …

n^2 is not O(n) because there is no **c** and n_0 such that:

$$n^2 \leq cn \text{ for } n \geq n_0$$

The figure to the right illustrates that no matter how large a c is chosen there is an n big enough that $n^2 > cn$



Asymptotic Notation

- Simple Rule: Drop lower order terms and constant factors
  - 50 n log n is O( )
  - $7n$ - 3 is O( )
  - $8n^2 \log n + 5n^2 + n$ is O( )

Asymptotic Analysis of Running Time

- Use *O*-notation to express number of primitive operations executed as function of input size
- Comparing asymptotic running times
  - an algorithm that runs in O(n) time is better than one that runs in O(n2) time
  - similarly, O(log n) is better than O(n)
  - hierarchy of functions: log n < n < n2 < n3 < 2n
- Caution! Beware of very large constant factors. An algorithm running in time 1,000,000 n is still O(n) but might be less efficient than one running in time $2n^2$, which is $O(n^2)$

Example of Asymptotic Analysis


```
Algorithm prefixAverages1(X)
```

Input: An n-element array X of numbers

Output: An n-element array A of numbers such that A[i] is the average of elements X[1], …, X[i]

```
for i ← 1 to n do
   a ← 0
   for j ← 1 to i do
     a ← a+X[j]
   A[i] ← a/i
return array A
```

⟵1 step

i iterations with i=1,2,…,n

n iterations

Analysis: running time is $O(n^2)$

A Better Algorithm

Algorithm prefixAverages2(X)

Input: An *n*-element array X of numbers

Output*:* An *n*-element array A of numbers such that A[*i*] is the average of elements X[1], … , X[*i*]

```
s ← 0
for i ← 1 to n do
   s ← s + X[i]
   A[i] ← s/i
return array A
```

Analysis: Running time is …

Asymptotic Notation (Terminology)

香港科技大学(广州)
THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY (GUANGZHOU)

- Special classes of algorithms:
  - Logarithmic: O(log n)
  - Linear: O(n)
  - Quadratic: $O(n^2)$
  - Polynomial: $O(n^k)$, $k \geq 1$
  - Exponential: $O(a^n)$, $a > 1$
- "Relatives" of the Big-Oh
  - Ω (f(n)): Big Omega-asymptotic lower bound
  - Θ (f(n)): Big Theta-asymptotic tight bound

What does O(1) mean?

We say $t(n)$ is $O(1)$, if there exist two positive constants n_0 and c such that, for all $n \geq n_0$.

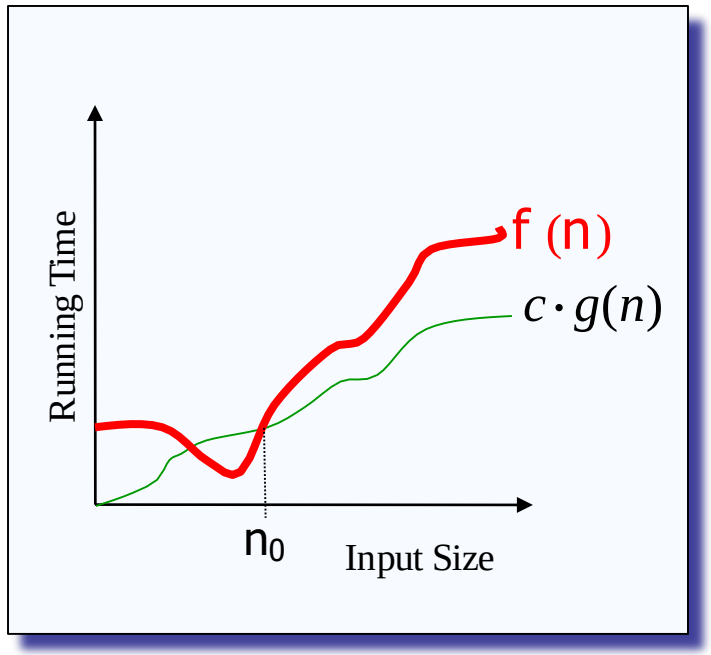
$$t(n) \leq c$$

So, it means that **$t(n)$ is bounded**.

Asymptotic Notation

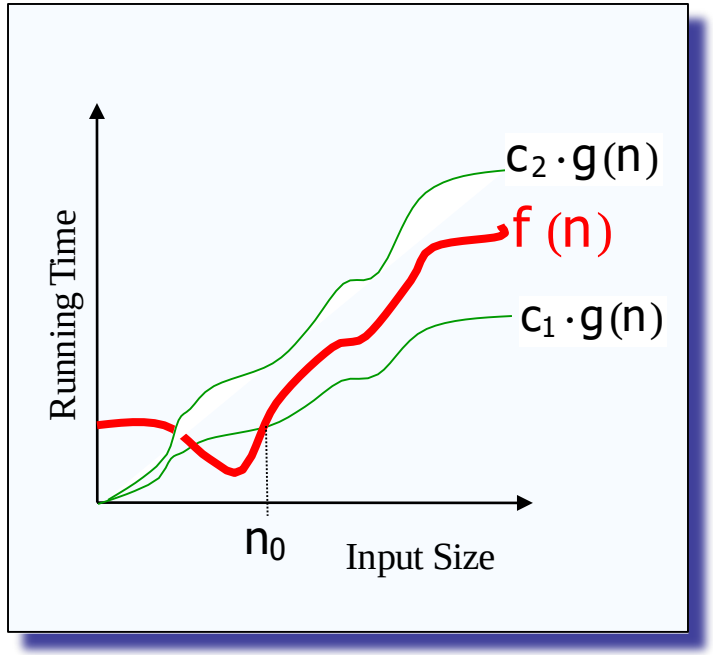
- The "Big-Omega" Ω–Notation
  - asymptotic lower bound
  - *f(n)* is Ω*(g(n))* if there exists constants *c* and n_0, *s.t.* **c g(n) ≤ f(n)** for $n \geq n_0$

- Used to describe *best-case* running times or lower bounds for algorithmic problems
  - E.g., lower-bound for searching in an unsorted array is Ω*(n)*



Asymptotic Notation

- The "Big-Theta" Θ–Notation
  - asymptotically tight bound
  - f(n) is Θ(g(n)) if there exists constants c_1, c_2, and n_0, s.t. $\mathbf{c_1\, g(n) \le f(n) \le c_2\, g(n)}$ for $n \ge n_0$
- f(n) is Θ(g(n)) if and only if f(n) is O(g(n)) and f(n) is Ω(g(n))
- O(f(n)) is often misused instead of Θ(f(n))



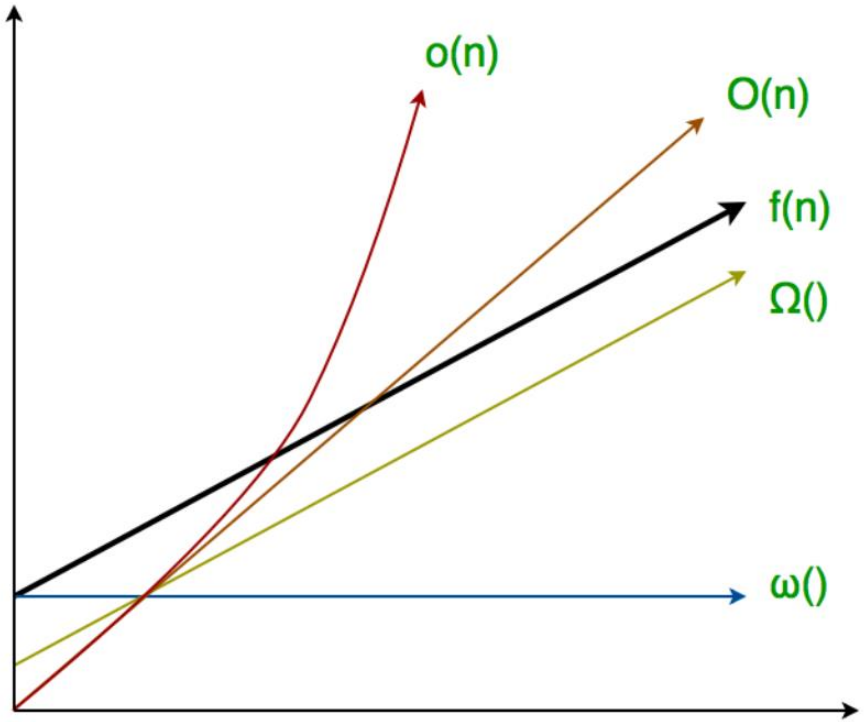
Asymptotic Notation

Two more asymptotic notations

- "Little-oh" notation *f(n)* is *o(g(n))*
  non-tight analogue of Big-Oh
  - For every $c > 0$, there should exist n_0, s.t.
    f(n) ≤ c g(n) for $n \geq n_0$
  - Used for **comparisons** of running times
  - If *f(n) is o(g(n))*, it is said that *g(n) dominates f(n)*
- "Little-omega" notation *f(n)* is *ω(g(n))*
  non-tight analogue of Big-Omega

Asymptotic Notation

- Analogy with real numbers
  - $f(n)$ is $O(g(n))$ ≅ $f \leq g$
  - $f(n)$ is $\Omega(g(n))$ ≅ $f \geq g$
  - $f(n)$ is $\Theta(g(n))$ ≅ $f = g$
  - $f(n)$ is $o(g(n))$ ≅ $f < g$
  - $f(n)$ is $\omega(g(n))$ ≅ $f > g$

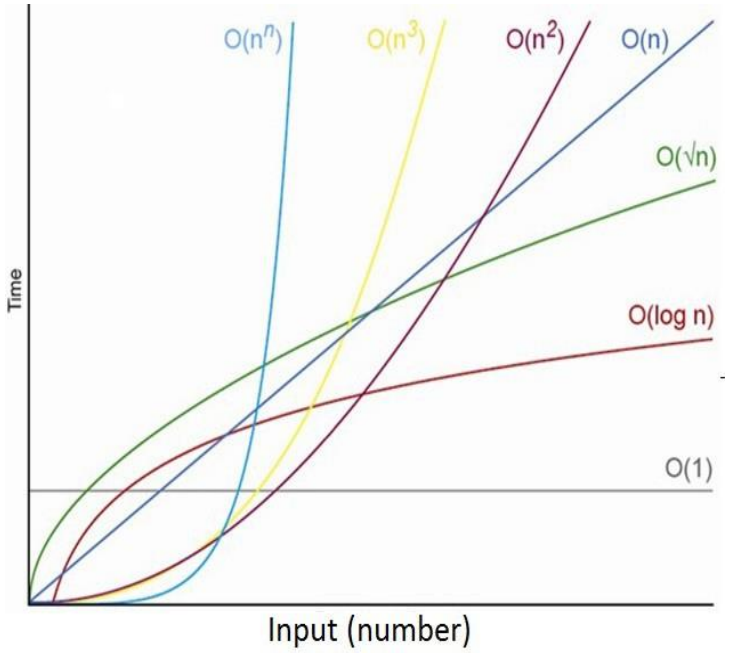


- Abuse of notation: $f(n) = O(g(n))$ actually means $f(n) \in O(g(n))$

The big O (resp. big Ω) denotes a tight upper (resp. lower) bounds, while the little o (resp. little ω) denotes a loose upper (resp. lower) bounds.

Practical meaning of big O…

	constant	*logarithmic*	*linear*	*N-log-N*	*quadratic*	*cubic*	*exponential*
n	**O(1)**	**O(log *n*)**	**O(*n*)**	**O(*n* log *n*)**	**O(n^2)**	**O(n^3)**	**O(2^n)**
1	1	1	1	1	1	1	2
2	1	1	2	2	4	8	4
4	1	2	4	8	16	64	16
8	1	3	8	24	64	512	256
16	1	4	16	64	256	4,096	65536
32	1	5	32	160	1,024	32,768	4,294,967,296
64	1	6	64	384	4,069	262,144	1.84×10^{19}



If the unit is in seconds, this would make ~10^{11} years…

Constant Factor Rule

Suppose $\mathrm{f(n)}$ is $\mathrm{O(g(n))}$ and a is a positive constant.
Then, $a \cdot f(n)$ is also $O(g(n))$

Proof: By definition, if $\mathrm{f(n)}$ is $\mathrm{O(g(n))}$ then there exists two positive constants n_0 and c such that for all $n \geq n_0$,

$$f(n) \leq c \cdot g(n)$$

Thus, $$\mathrm{a} \cdot f(n) \leq \mathrm{a} \cdot c \cdot g(n)$$

We use the constant $a \cdot c$ to show that $a \cdot f(n)$ is $O(g(n))$.

Multiplying a function by a constant does not change its Big O upper bound since these upper bounds ignore constants

Sum Rule

Suppose $f_1(n)$ *is* $O(g(n))$ and $f_2(n)$ *is* $O(g(n))$.

Then, $f_1(n) + f_2(n)$ *is* $O(g(n))$.

Proof: Let n_1, c_1 and n_2, c_2 be constants such that

$$f_1(n) \leq c_1 g(n), \text{ for all } n \geq n_1$$

$$f_2(n) \leq c_2 g(n), \text{ for all } n \geq n_2$$

So, $f_1(n) + f_2(n) \leq (c_1+c_2)g(n), \text{ for all } n \geq \max(n_1, n_2)$.

We can use the constants $c_1 + c_2$ and $\max(n_1, n_2)$ to satisfy the definition.

A sum of two functions is inferior to a sum of two greater functions

Product Rule

Suppose $f_1(n)$ *is* $O(g(n))$ and $f_2(n)$ *is* $O(g(n))$.

Then, $f_1(n) \cdot f_2(n)$ *is* $O(g_1(n) \cdot g_2(n))$.

Proof: Let n_1, c_1 and n_2, c_2 be constants such that

$$f_1(n) \leq c_1 g_1(n), \text{ for all } n \geq n_1$$

$$f_2(n) \leq c_2 g_2(n), \text{ for all } n \geq n_2$$

So, $f_1(n) \cdot f_2(n) \leq (c_1 \cdot c_2) \cdot (g_1(n) \cdot g_2(n))$, for all $n \geq \max(n_1, n_2)$.

We can use the constants $c_1 \cdot c_2$ and $\max(n_1, n_2)$ to satisfy the definition.

A product of two functions is less than a product of two greater functions

Transitivity Rule

Suppose $f(n)$ *is* $O(g(n))$ and $g(n)$ *is* $O(h(n))$.

Then, $f(n)$*is* $O(h(n))$.

Proof: Let n_1, c_1 and n_2, c_2 be constants such that

$$f(n) \leq c_1 g(n), \text{ for all } n \geq n_1$$

$$g(n) \leq c_2 h(n), \text{ for all } n \geq n_2$$

So, $f(n) \leq (c_1 \cdot c_2) h(n)$, for all $n \geq \max(n_1, n_2)$.

We can use the constants $c_1 \cdot c_2$ and $\max(n_1, n_2)$ to satisfy the definition.

If a function A is greater than function B, and function B is greater than function C, then function A is greater than function C

Analyzing insertion sort

$O(n^2)$ sorting algorithm: Insertion Sort

INSERTION-SORT(A, n)	*cost*	*times*
1 **for** $i = 2$ **to** n	c_1	n
2 $key = A[i]$	c_2	$n-1$
3 // Insert $A[i]$ into the sorted subarray $A[1:i-1]$.	0	$n-1$
4 $j = i-1$	c_4	$n-1$
5 **while** $j > 0$ and $A[j] > key$	c_5	$\sum_{i=2}^{n} t_i$
6 $A[j+1] = A[j]$	c_6	$\sum_{i=2}^{n}(t_i - 1)$
7 $j = j-1$	c_7	$\sum_{i=2}^{n}(t_i - 1)$
8 $A[j+1] = key$	c_8	$n-1$

Best case: A is sorted, while loop does not execute.

$$\mathrm{T}(n) = n(c_1 + c_2 + c_4 + c_8 - c_6 - c_7) + (n-1)c_5$$

$$-(c_2 + c_4 + \mathrm{c}_6 - c_7 - c_8) = O(n)$$

$O(n^2)$ sorting algorithm: Insertion Sort

INSERTION-SORT(A, n)	*cost*	*times*
1 **for** $i = 2$ **to** n	c_1	n
2 $key = A[i]$	c_2	$n-1$
3 // Insert $A[i]$ into the sorted subarray $A[1:i-1]$.	0	$n-1$
4 $j = i-1$	c_4	$n-1$
5 **while** $j > 0$ and $A[j] > key$	c_5	$\sum_{i=2}^{n} t_i$
6 $A[j+1] = A[j]$	c_6	$\sum_{i=2}^{n}(t_i - 1)$
7 $j = j-1$	c_7	$\sum_{i=2}^{n}(t_i - 1)$
8 $A[j+1] = key$	c_8	$n-1$

Worse case: A is reverse-ordered. The while loop execute i-1 times for each i

$$\mathrm{T}(n) = n(c_1 + c_2 + c_4 + c_8 - c_6 - c_7) + \sum_{\mathrm{i}=2}^{n}(i-1)(c_5 + c_6 + c_7) - (c_2 + c_4 + \mathrm{c}_6 - c_7 - c_8)$$

$$= n(c_1 + c_2 + c_4 + c_8 - c_6 - c_7) + \sum_{\mathrm{i}=2}^{n}(i-1)(c_5 + c_6 + c_7) = an + \frac{bn(n-1)}{2} = O(n^2)$$

$O(n^2)$ sorting algorithm: Insertion Sort

INSERTION-SORT(A, n)	*cost*	*times*
1 **for** $i = 2$ **to** n	c_1	n
2 $key = A[i]$	c_2	$n-1$
3 // Insert $A[i]$ into the sorted subarray $A[1:i-1]$.	0	$n-1$
4 $j = i-1$	c_4	$n-1$
5 **while** $j > 0$ and $A[j] > key$	c_5	$\sum_{i=2}^{n} t_i$
6 $A[j+1] = A[j]$	c_6	$\sum_{i=2}^{n}(t_i - 1)$
7 $j = j-1$	c_7	$\sum_{i=2}^{n}(t_i - 1)$
8 $A[j+1] = key$	c_8	$n-1$

Average case: The while loop is expected to execute (i-1)/2 times for each i

$$\mathrm{T}(n) = n(c_1 + c_2 + c_4 + c_8 - c_6 - c_7) + \sum_{\mathrm{i}=2}^{n} \frac{i-1}{2}(c_5 + c_6 + c_7) \; -(c_2 + c_4 + \mathrm{c}_6 - c_7 - c_8)$$

$$= n(c_1 + c_2 + c_4 + c_8 - c_6 - c_7) + \sum_{\mathrm{i}=2}^{n} \frac{i-1}{2}(c_5 + c_6 + c_7) = an + \frac{bn(n-1)}{4} = O(n^2)$$