COMP250-Recurrence

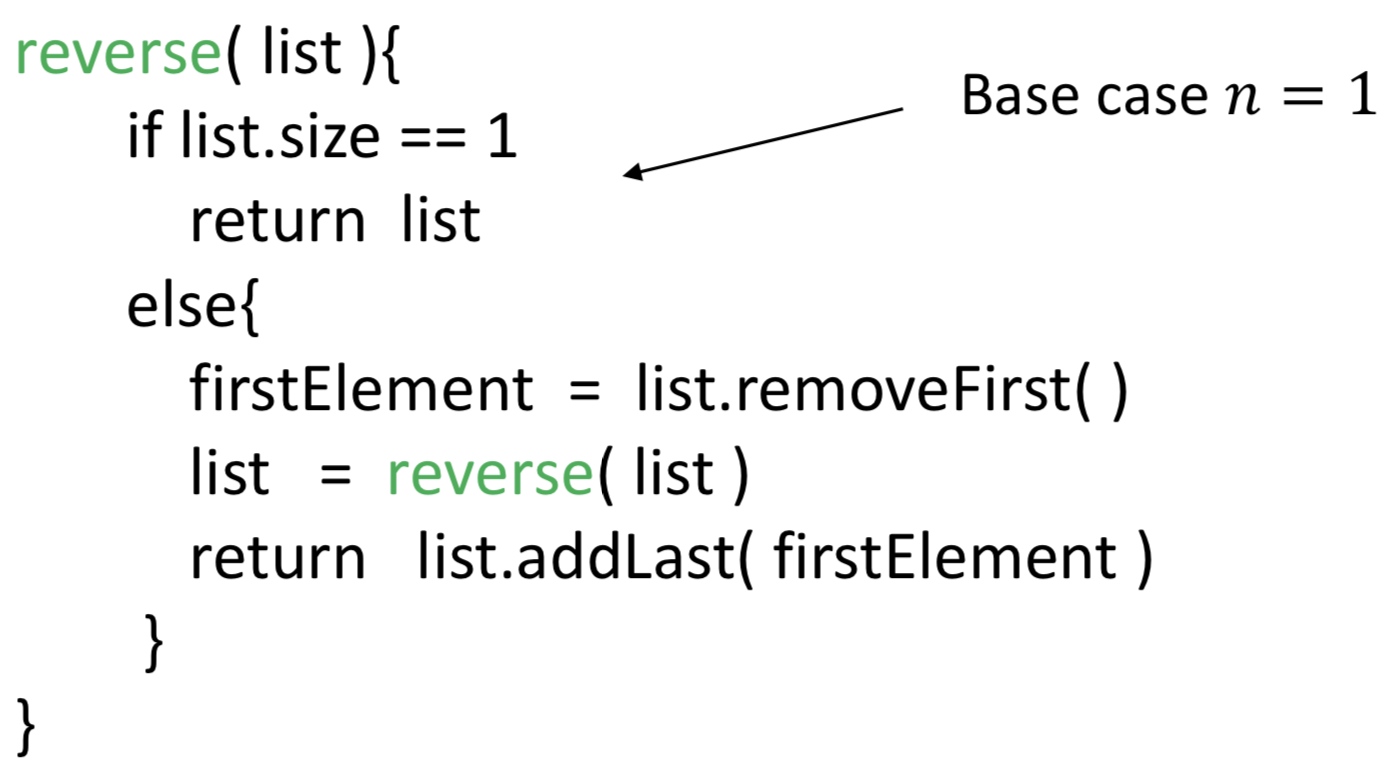
**Recurrence relation**

A *recurrence relation* is a sequence of numbers where the 𝑛-th term depends on previous terms.

e.g. Fibonacci 𝐹 𝑛 =𝐹 𝑛−1 +𝐹(𝑛−2)

We will consider recurrence relations for 𝑡 𝑛 , the time to execute a recursive algorithm, as a function of the input size 𝑛. The recurrence expresses it in terms of the smaller input size.

[Example: reversion]



Assumptions about removeFirst() and addLast()

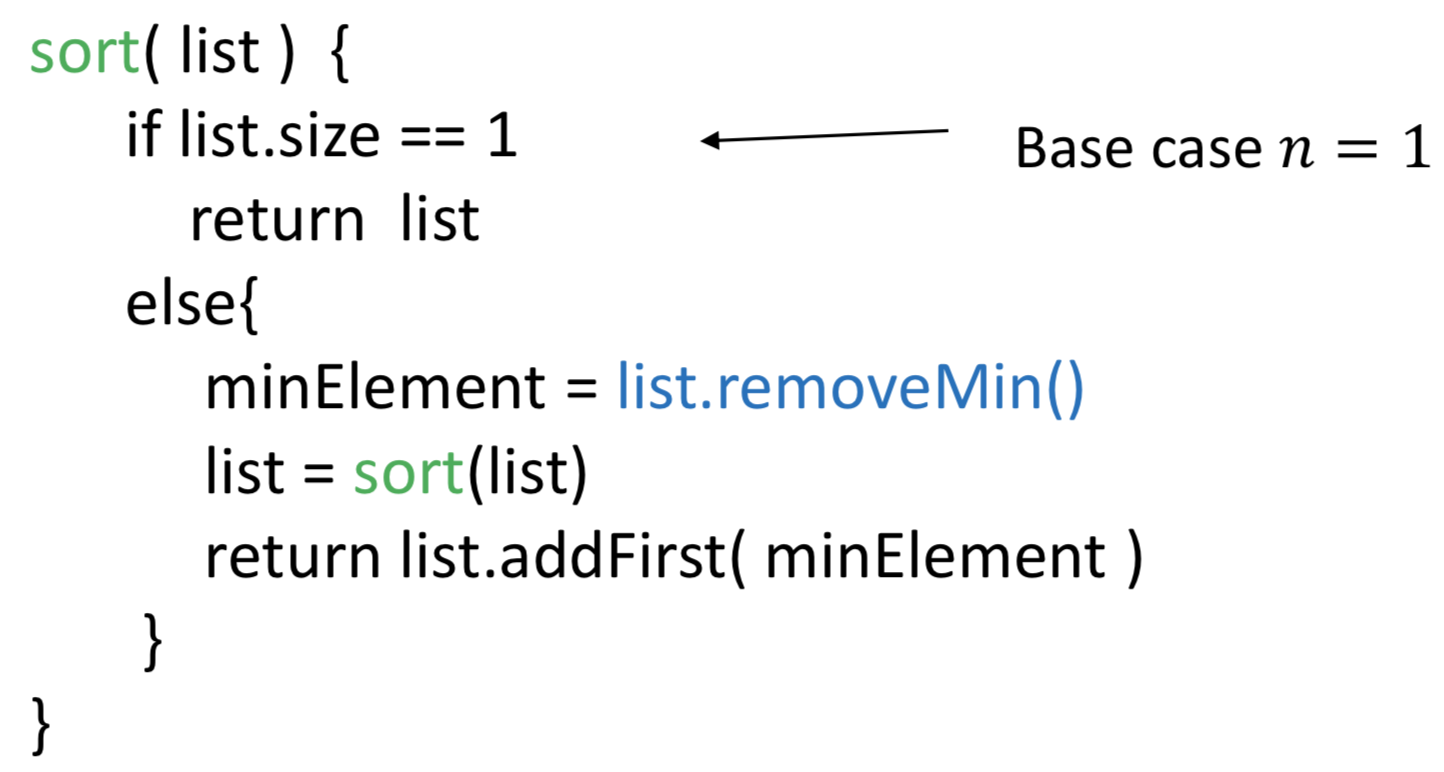
They can be done in constant time.

(The former is not true if we use an array list.)

t(n)=c+t(n-1)=c+c+t(n-2)=……=c(n-1)+t(1) if base case is 1(reversion)

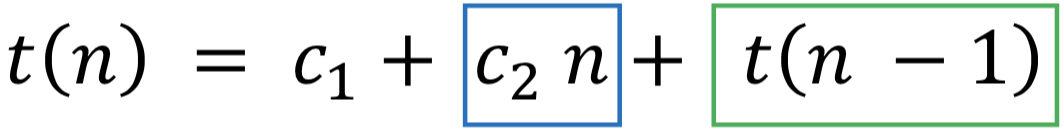
=cn+t(0) if base case is 0

[Example: sorting]

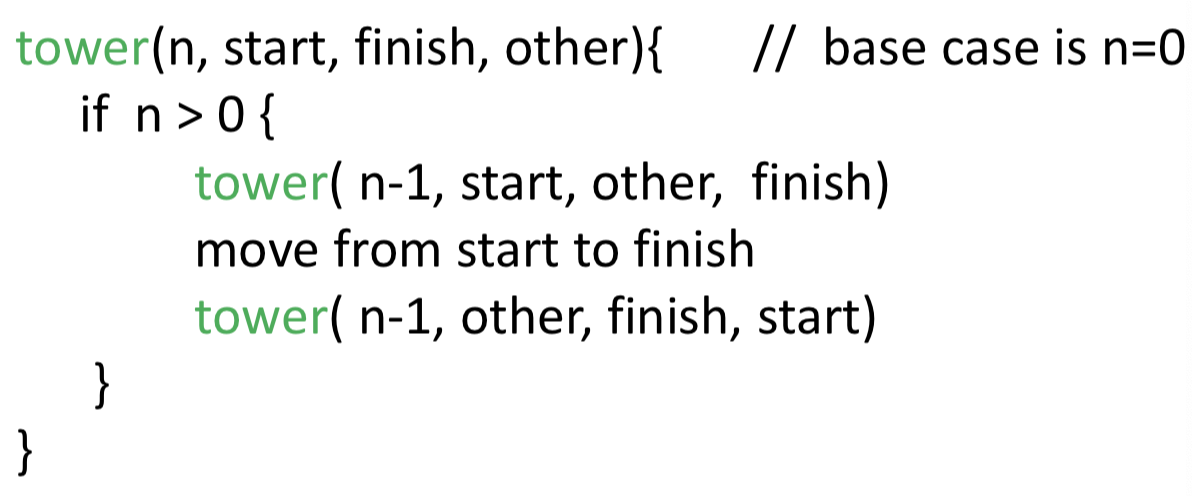


Assumptions about addFirst()

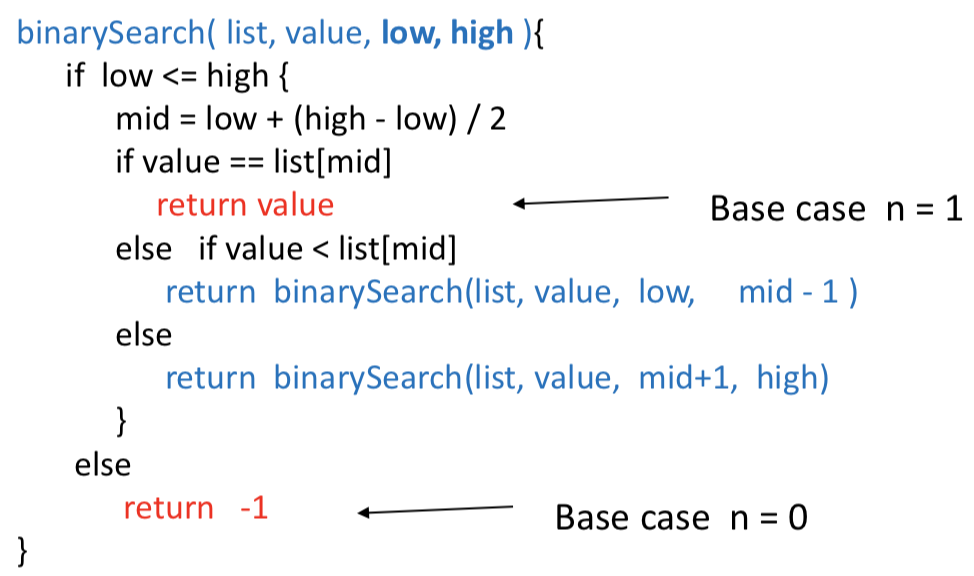
It is ok, if this step uses time proportional to n because listRemove() already has time proportional to n



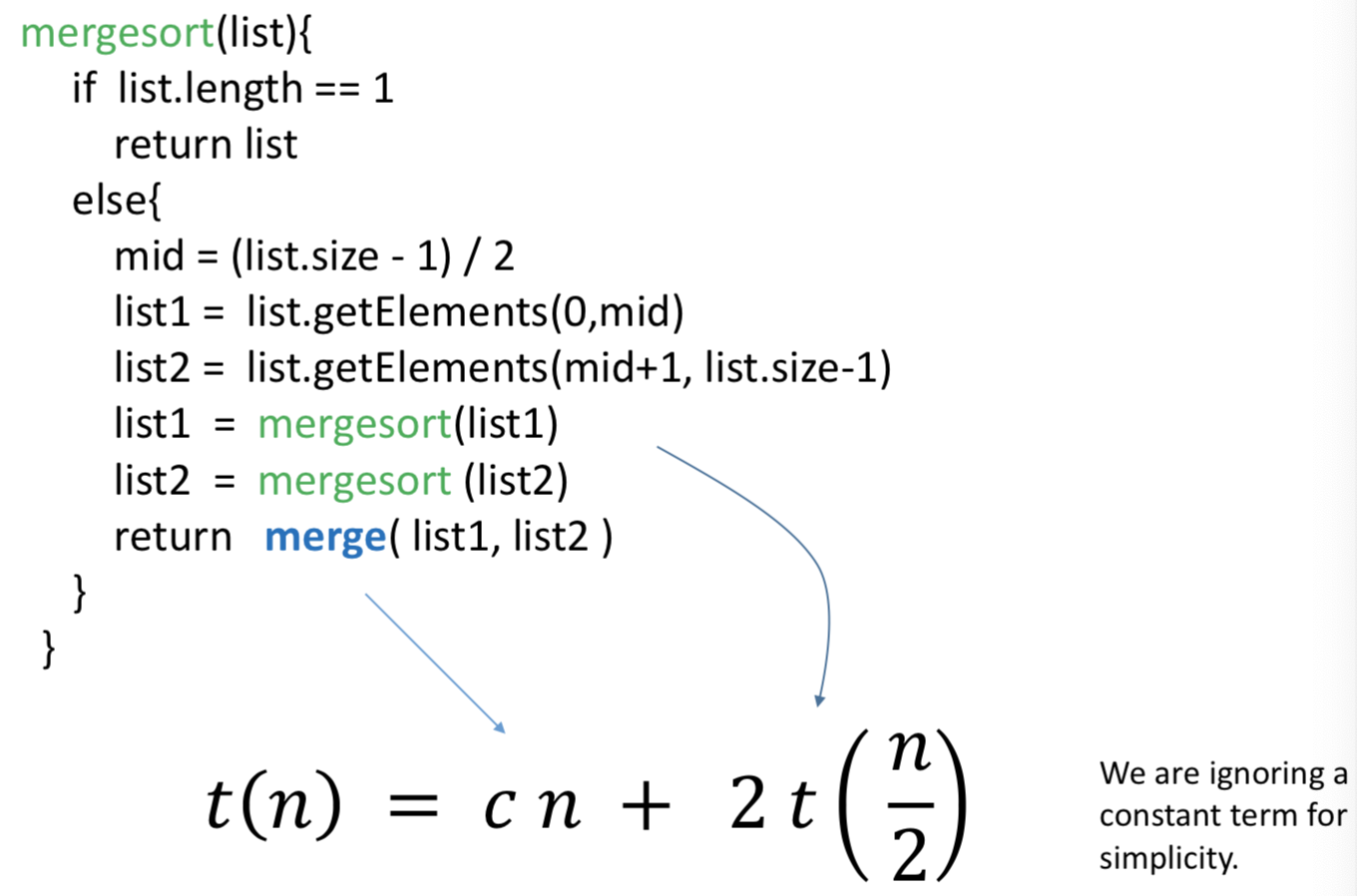
[Example: Tower of Hanoi]

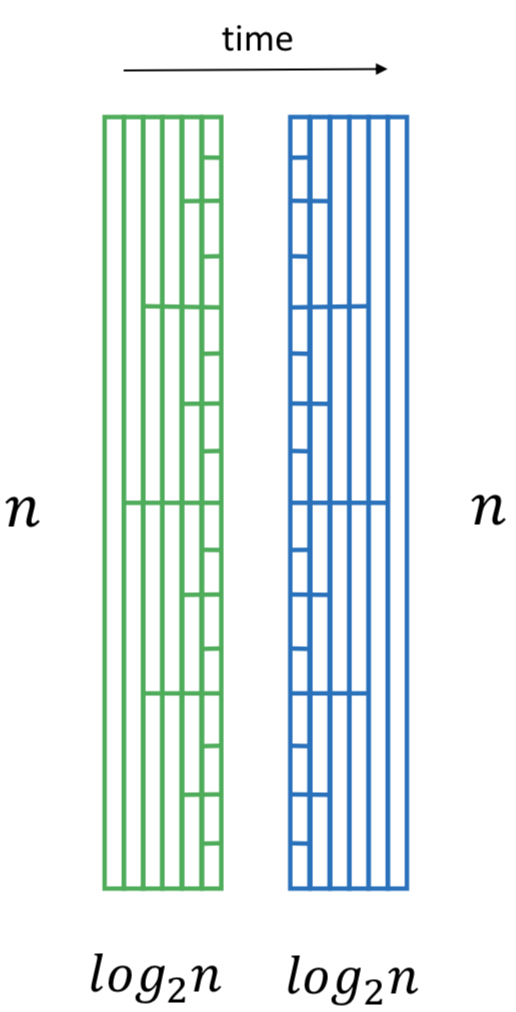
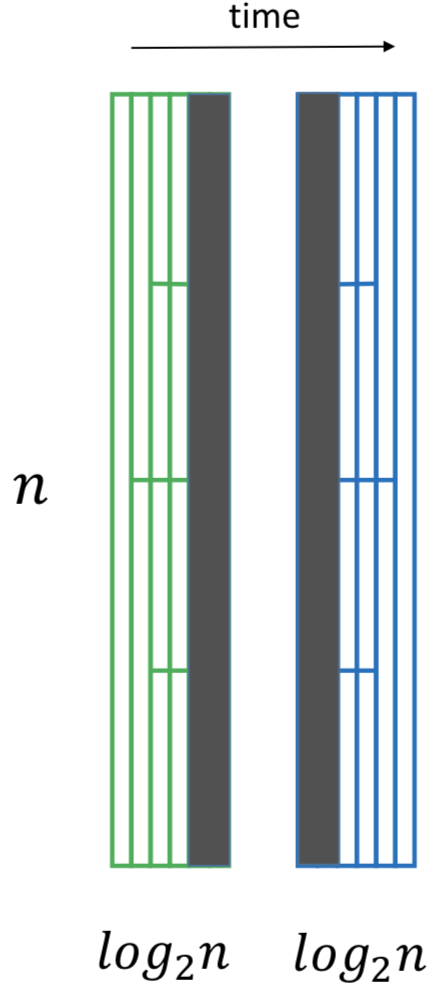


[Exampel: Binary Search]



[Example: Merge sort]



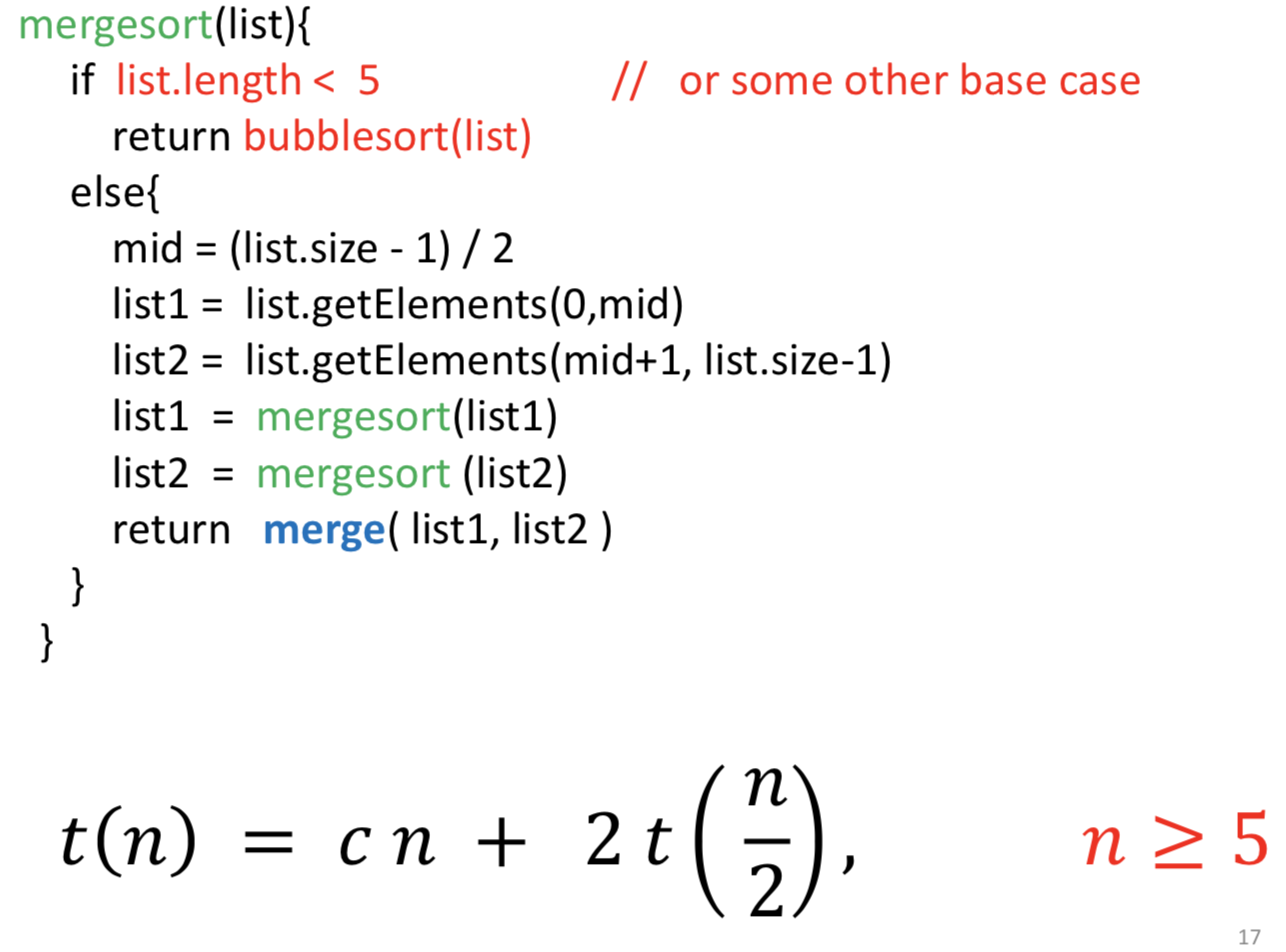


How many recursive calls are made to mergesort?

𝑛−1 (Count blocks)

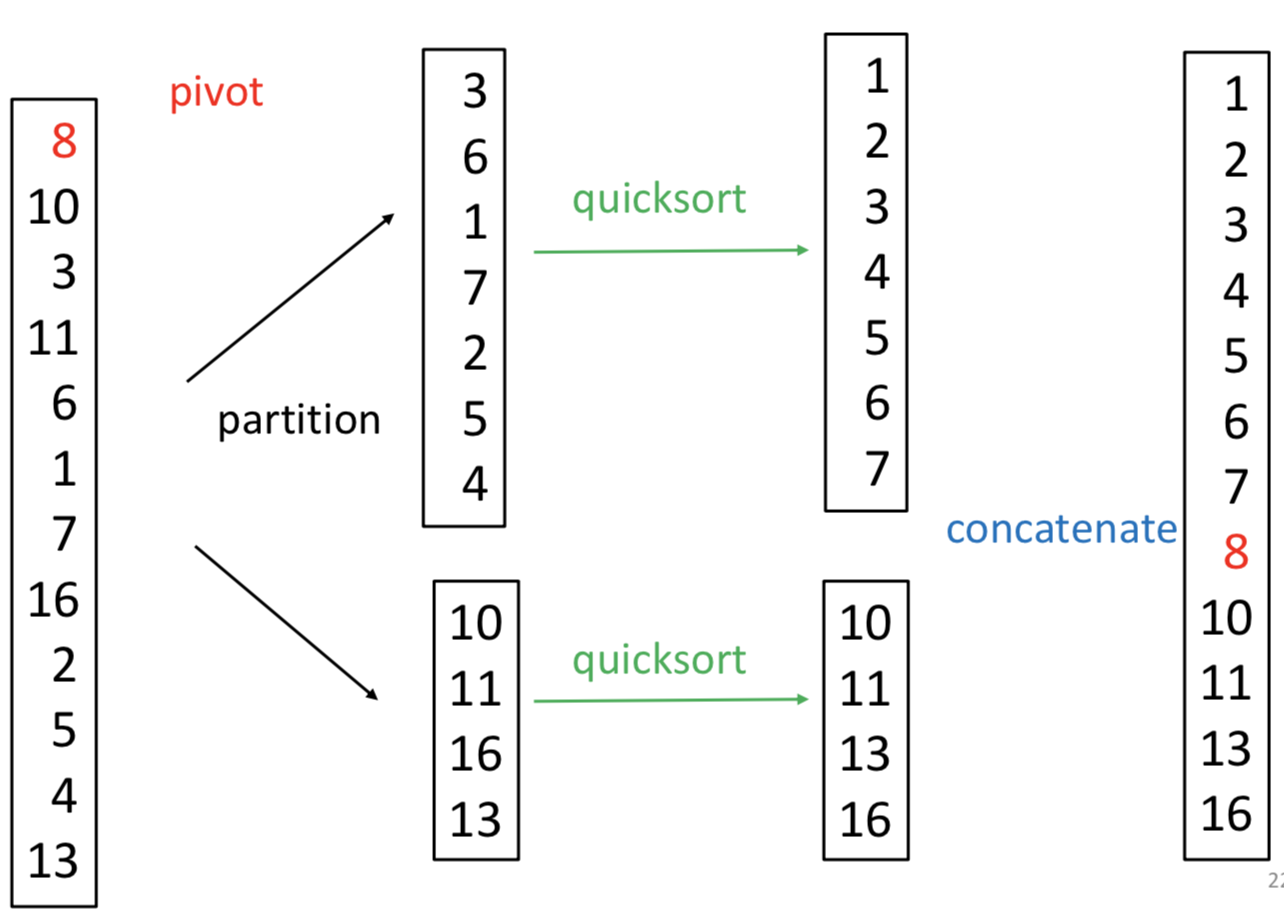
What if we change the base case and stop the recursion

at a larger listsize 𝑛0 >1 ?

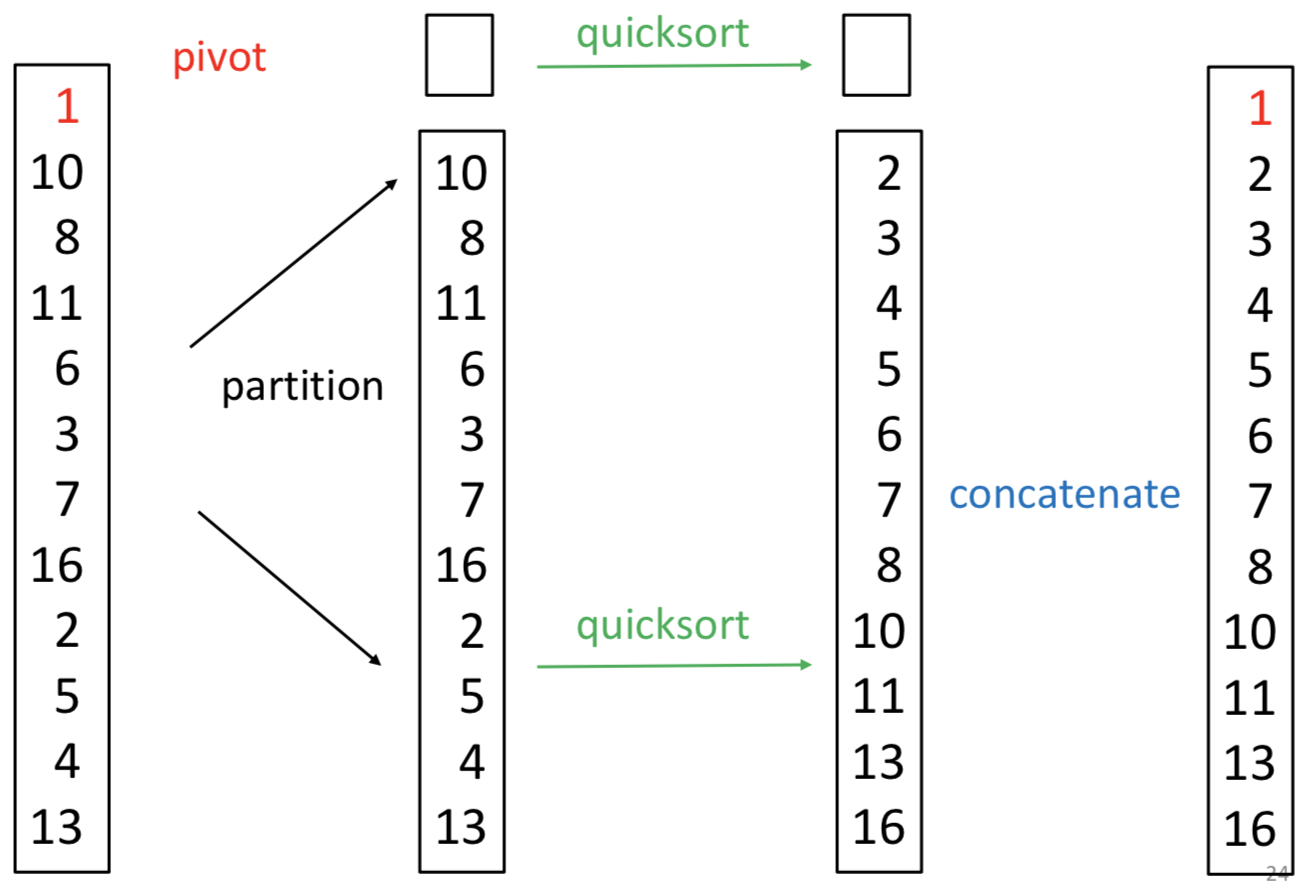


[Example: Quick sort]

best case



worst case



How to reduce the chance of an unbalanced partition at each step of Quicksort?

“Median of three” : Let the pivot be the median of first, middle, and last elements in list.

COMP250-Big O, Omega & Theta

The formal definition of O( ) itself says nothing about algorithms, and hence nothing about best and worst cases of an algorithm.

big O

Let 𝑡(𝑛) and 𝑔(𝑛) be two functions, where 𝑛 ≥ 0. We say 𝑡 𝑛 is *asymptotically bounded* above by 𝑔(𝑛) if there exists 𝑛0 such that, for all 𝑛 ≥ 𝑛0 , 𝑡(𝑛)≤𝑔(𝑛).

**[Formal definition of big O]**

Let 𝑡(𝑛) and 𝑔(𝑛) be two functions, where 𝑛 ≥ 0.

(𝑔(𝑛) will be a simple function, but this is not required in the definition.)

We say 𝑡(𝑛) is 𝑂( 𝑔(𝑛) ) if there exist two positive constants 𝑛0 and 𝑐 such that, for all 𝑛 ≥ 𝑛0 ,

𝑡(𝑛)≤𝑐𝑔(𝑛).

We say 𝑡(𝑛) is 𝑂(1), if there exist two positive constants 𝑛0 and 𝑐, s.t. for all 𝑛 ≥ 𝑛0 , 𝑡(𝑛) ≤ 𝑐.

That is, 𝑡(𝑛) is bounded: 𝑡(𝑛) has a finite number of values for 𝑛 < 𝑛0

Note:

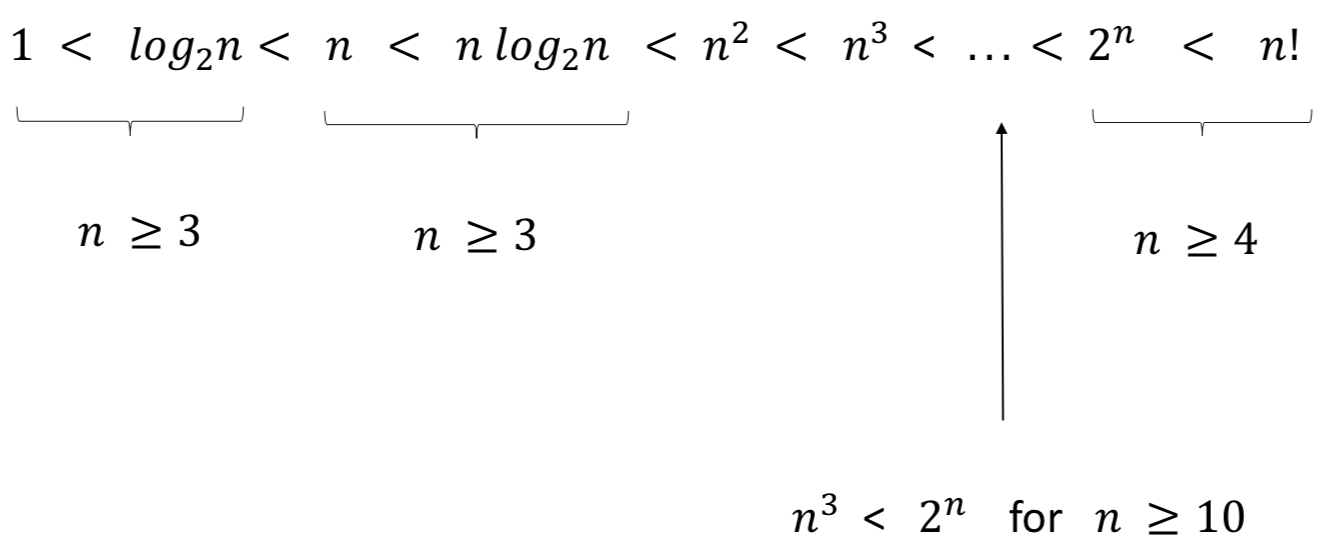
* Never write 𝑂(3𝑛) , 𝑂(5 𝑙𝑜𝑔 𝑛); Instead, write 𝑂(𝑛), 𝑂(𝑙𝑜𝑔 𝑛 ).

Why? The point of big O notation is to avoid dealing with these constant factors, though it is still technically correct to write the above.

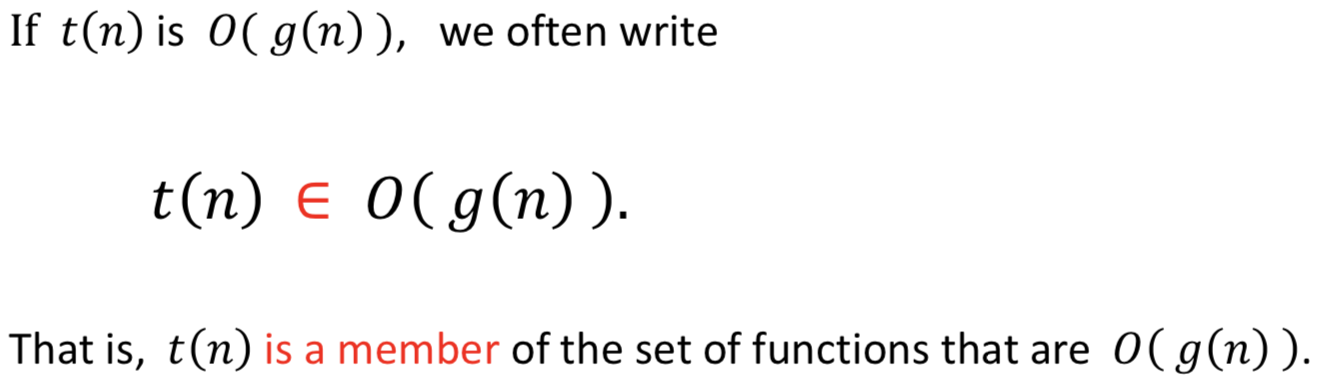
* Big O is about **upper** bounds.

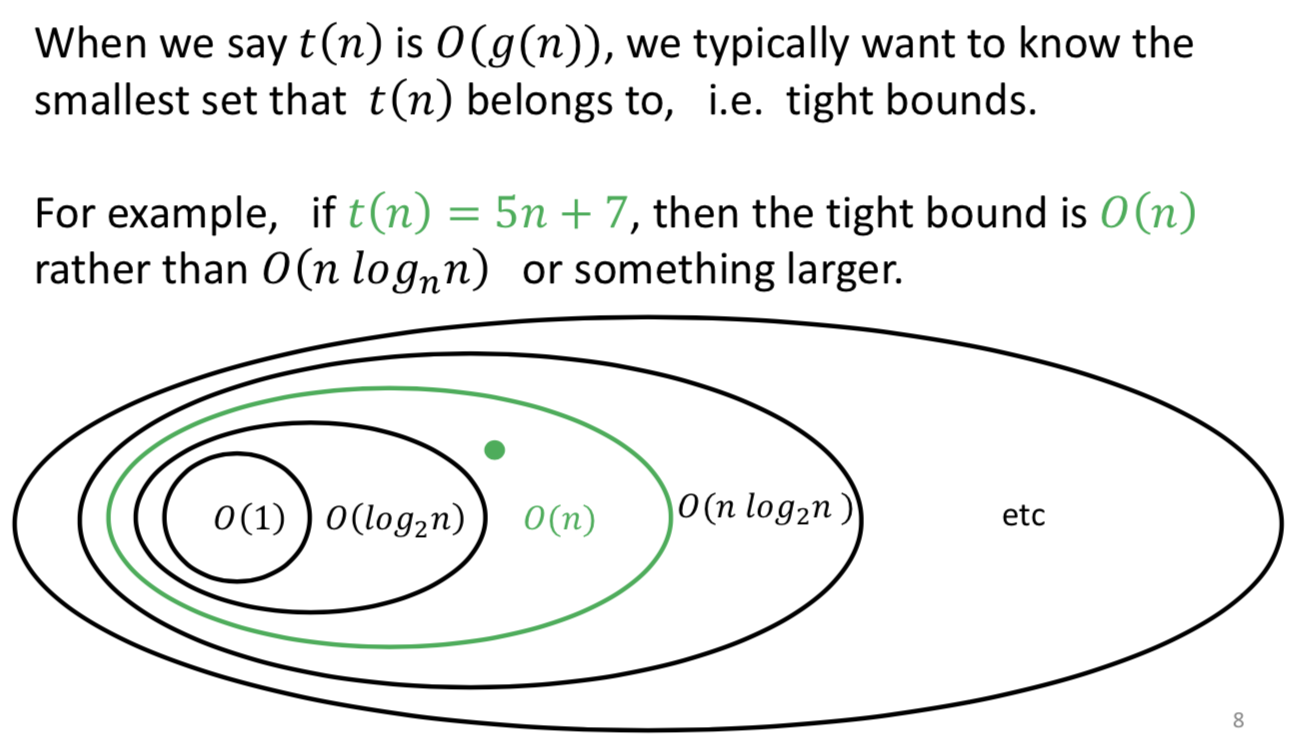
If 𝑡(𝑛) is 𝑂(𝑛), then can we say that 𝑡(𝑛) is also 𝑂(𝑛2) ? According to the formal definition, yes we can. But when we ask for “tight bounds” on 𝑡(𝑛), we want the simple function g(𝑛) with the smallest growth rate.

By convention, we will use simple functions g(n) below:



[Set of O() functions]



𝑂(1) ⊂ 𝑂(𝑙𝑜𝑔2𝑛) ⊂ 𝑂(𝑛) ⊂ 𝑂(𝑛𝑙𝑜𝑔2𝑛) ⊂ 𝑂(𝑛2)... ⊂ 𝑂(𝑛3) ⊂ ...⊂ 𝑂(2𝑛) ⊂𝑂(𝑛!)

* Constant Factor Rule

Suppose 𝑓(𝑛) is O(𝑔(𝑛)) and 𝑎 is a positive constant. Then 𝑎𝑓(𝑛) is also O(𝑔(𝑛))

* Sum Rule

Suppose 𝑓1(𝑛) is O(𝑔1(𝑛)) and 𝑓2(𝑛) is O(𝑔2(𝑛)) and 𝑔1(𝑛) is O(𝑔2(𝑛)).

Then 𝑓1(𝑛)+𝑓2(𝑛) is O(𝑔2(𝑛)).

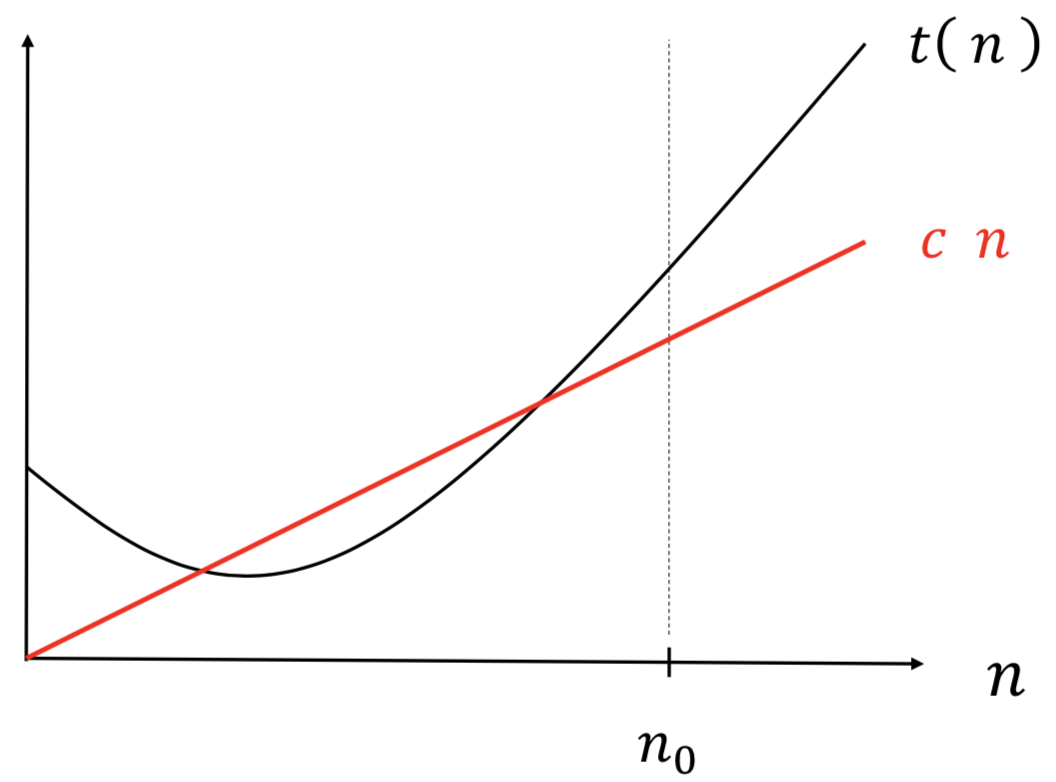
* Product Rule

Suppose 𝑓1(𝑛) is O(𝑔1(𝑛)) and 𝑓2(𝑛) is O(𝑔2(𝑛))

Then 𝑓1 (𝑛) ∗ 𝑓2 (𝑛) is O(𝑔1(𝑛))∗ O(𝑔2(𝑛))

Big Omega (Ω ): asymptotic lower bound

*Sometimes we want to say that an algorithm takes at least a certain time to run as a function of the input size 𝑛.*

* Example 1:

Let 𝑡(𝑛) be the time it takes for algorithm X to find the maximum value in an array of 𝑛 numbers.

𝑡(𝑛) = Ω(𝑛)

* Example 2: (Comparison based sorting)

Let 𝑡(𝑛) be the number of element comparisons used by algorithm X to sort an array of 𝑛 numbers.

𝑡(𝑛) = Ω(𝑛𝑙𝑜𝑔2𝑛) COMP 251

That is, there is no faster algorithm possible than those we have seen (e.g. merge/heap/quicksort).

**[Formal Definition of Big Omega** Ω**]**

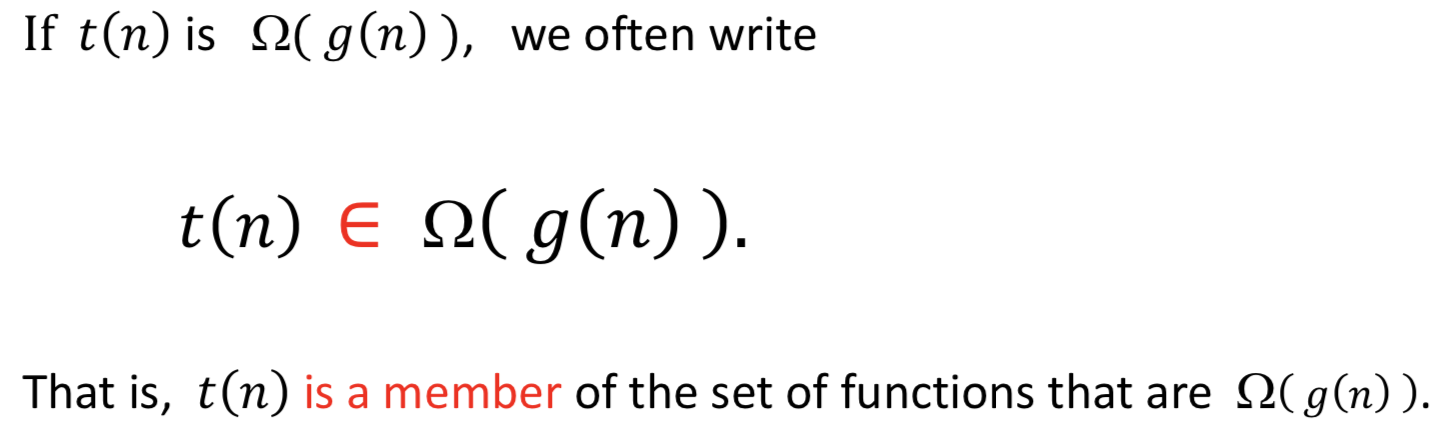
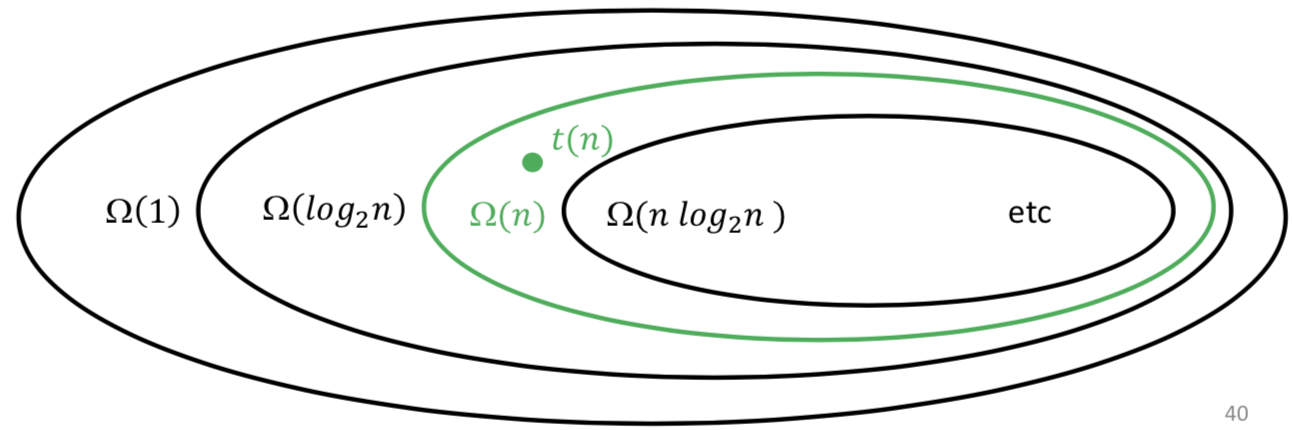
Let 𝑡(𝑛) and 𝑔(𝑛) be two functions of 𝑛 ≥ 0.

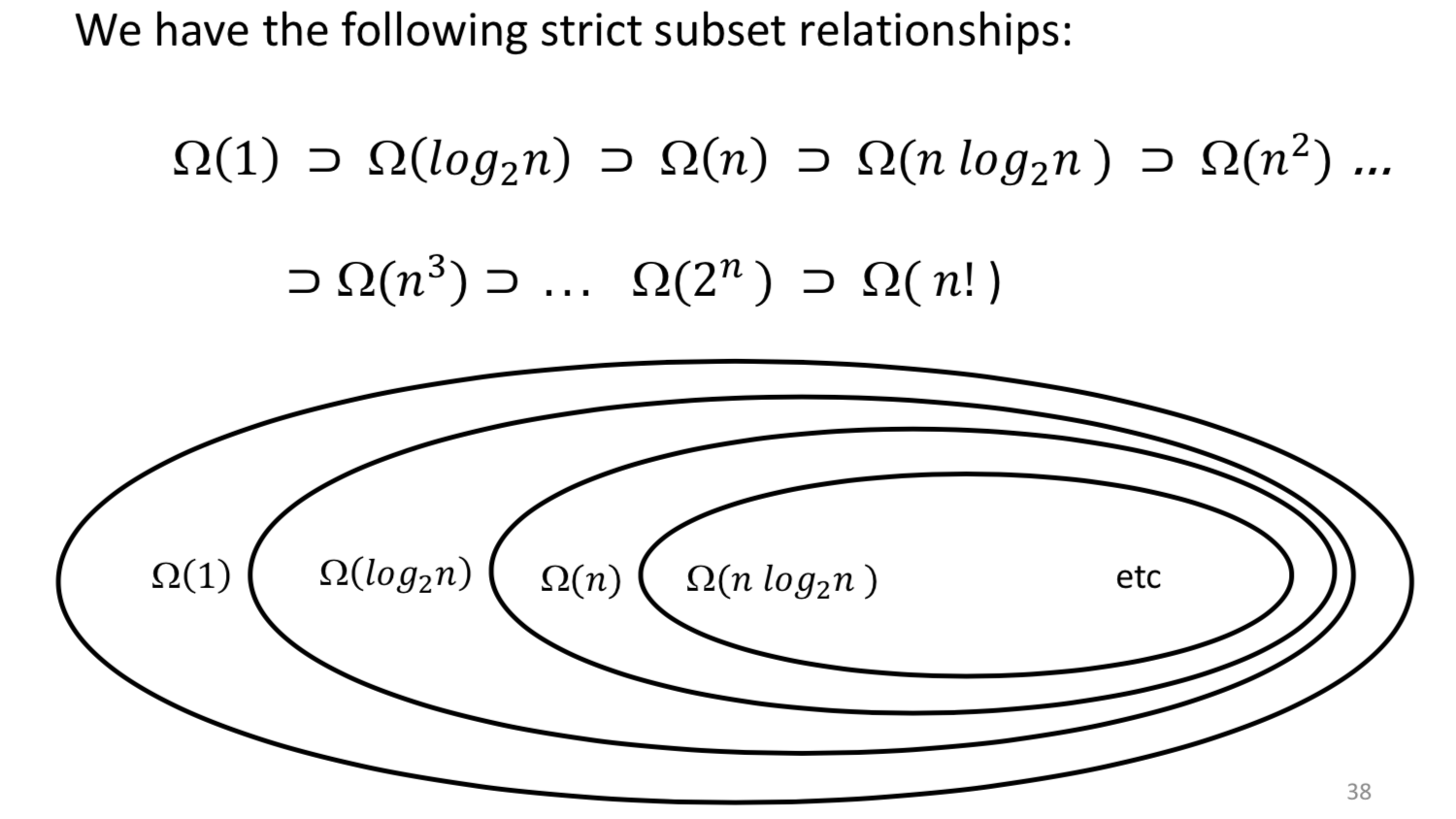
We say 𝑡(𝑛) is Ω(𝑔(𝑛)), if there exist two positive constants 𝑛0 and 𝑐 such that, for all 𝑛 ≥ 𝑛0 ,

𝑡(𝑛)≥𝑐𝑔(𝑛).

Let 𝑓(𝑛) and 𝑔(𝑛) be two functions of 𝑛 ≥ 0. The following are equivalent statements:

f (n) is O(g(n)) g(n) is Ω(𝑔(𝑛))

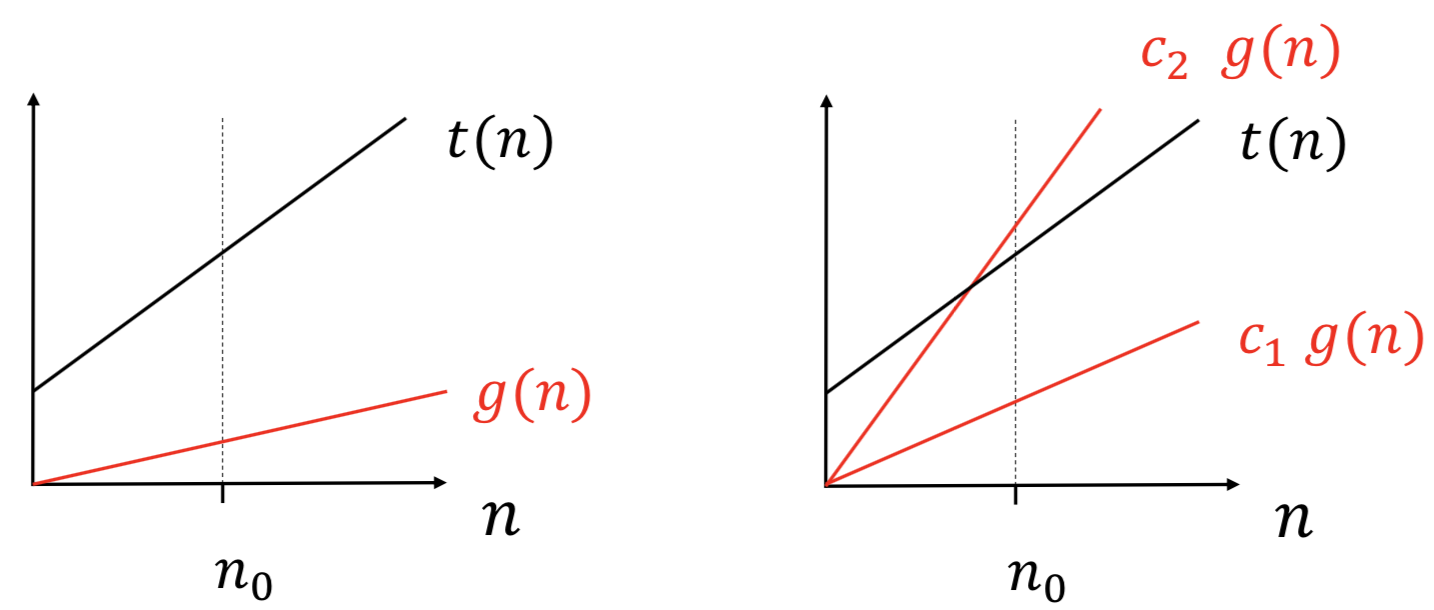


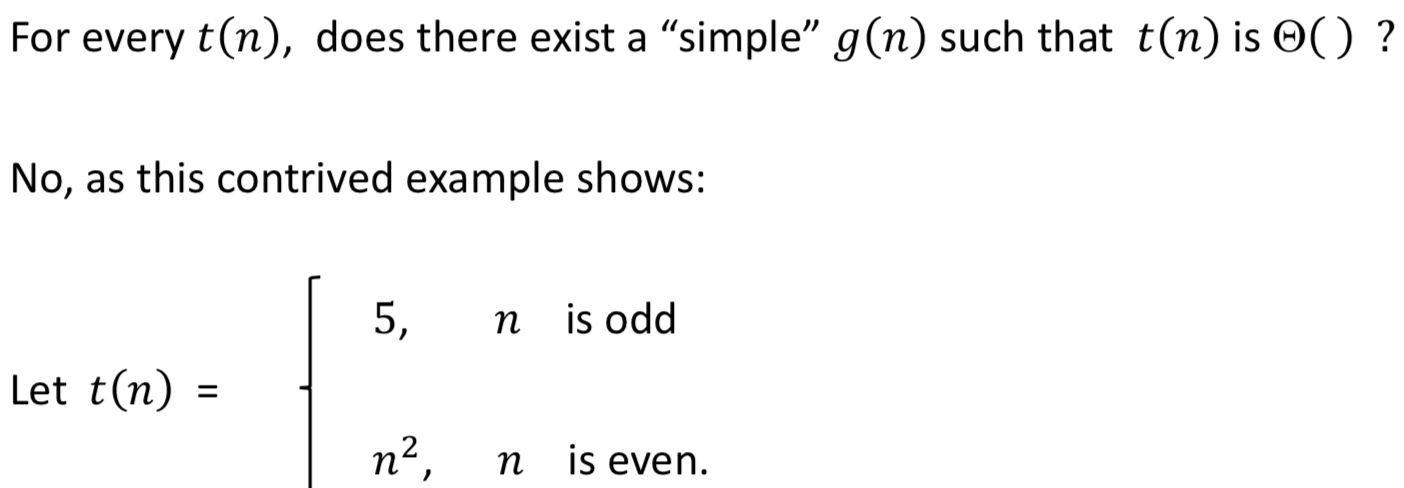
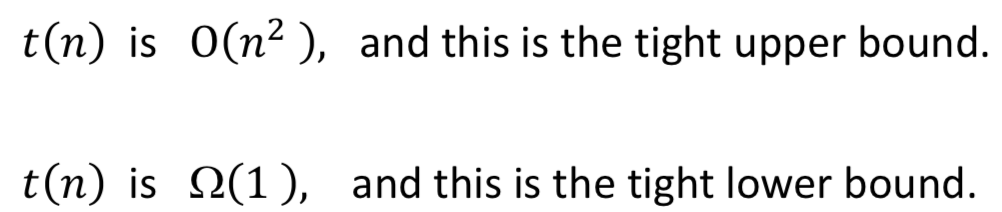


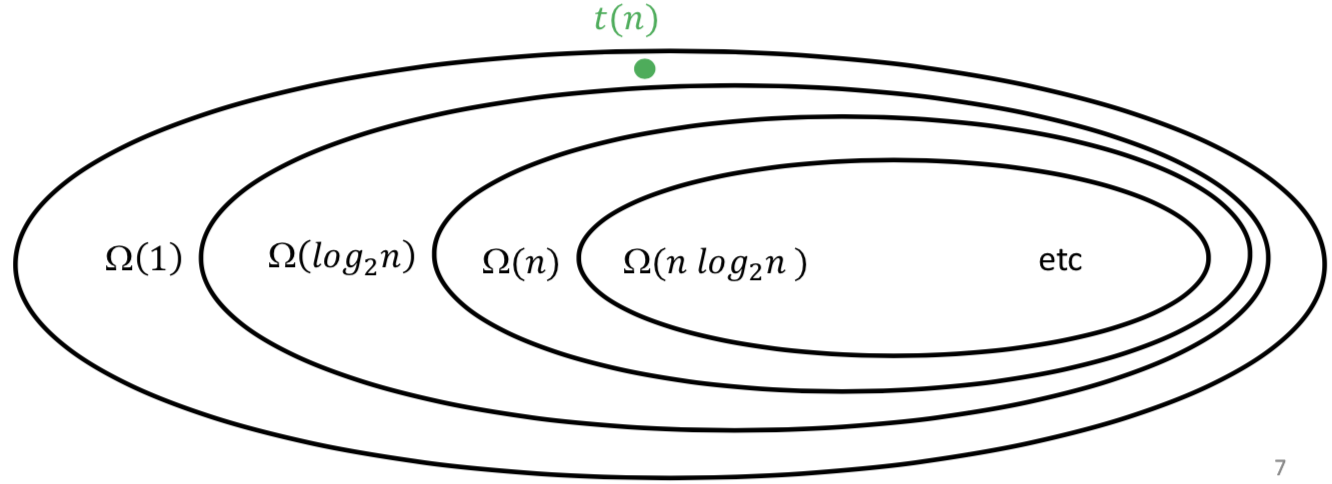
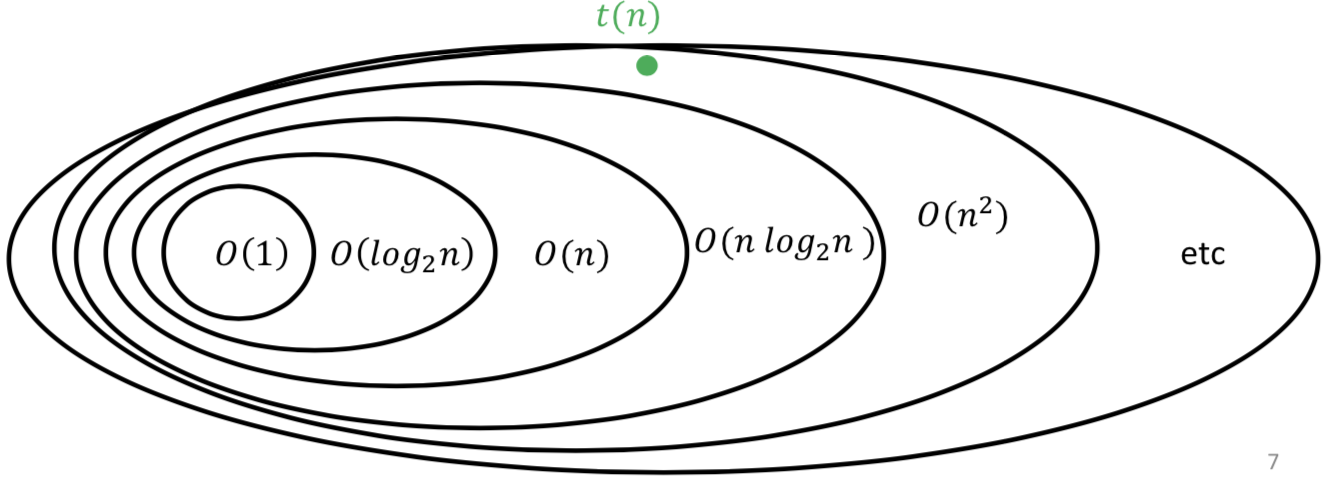
Big Theta

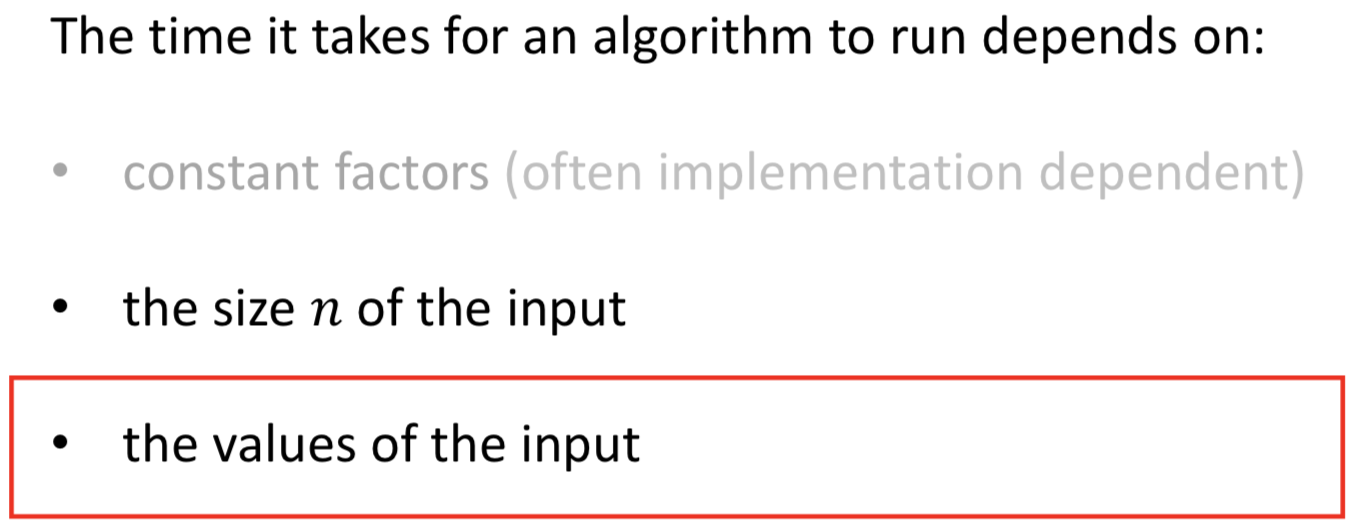
**[Formal Definition of Big Theta]**

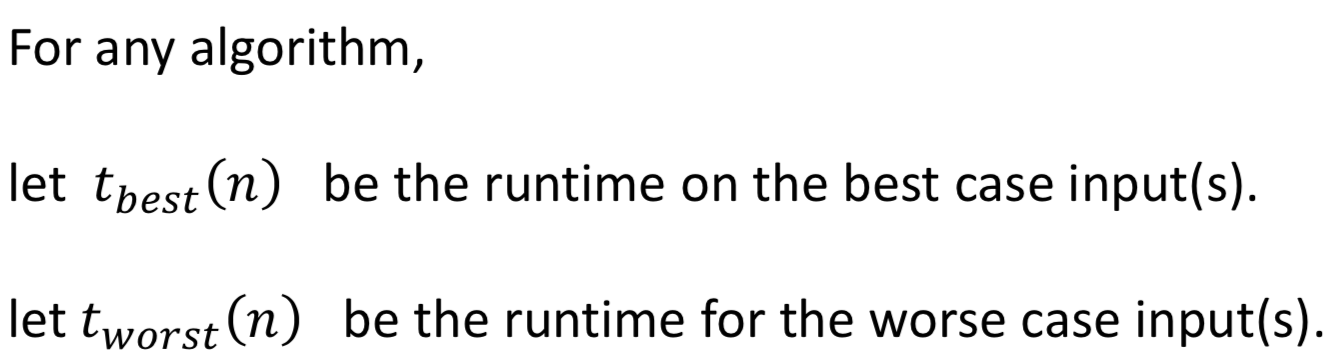
Let 𝑡(𝑛) and 𝑔(𝑛) be two functions of 𝑛 ≥ 0.

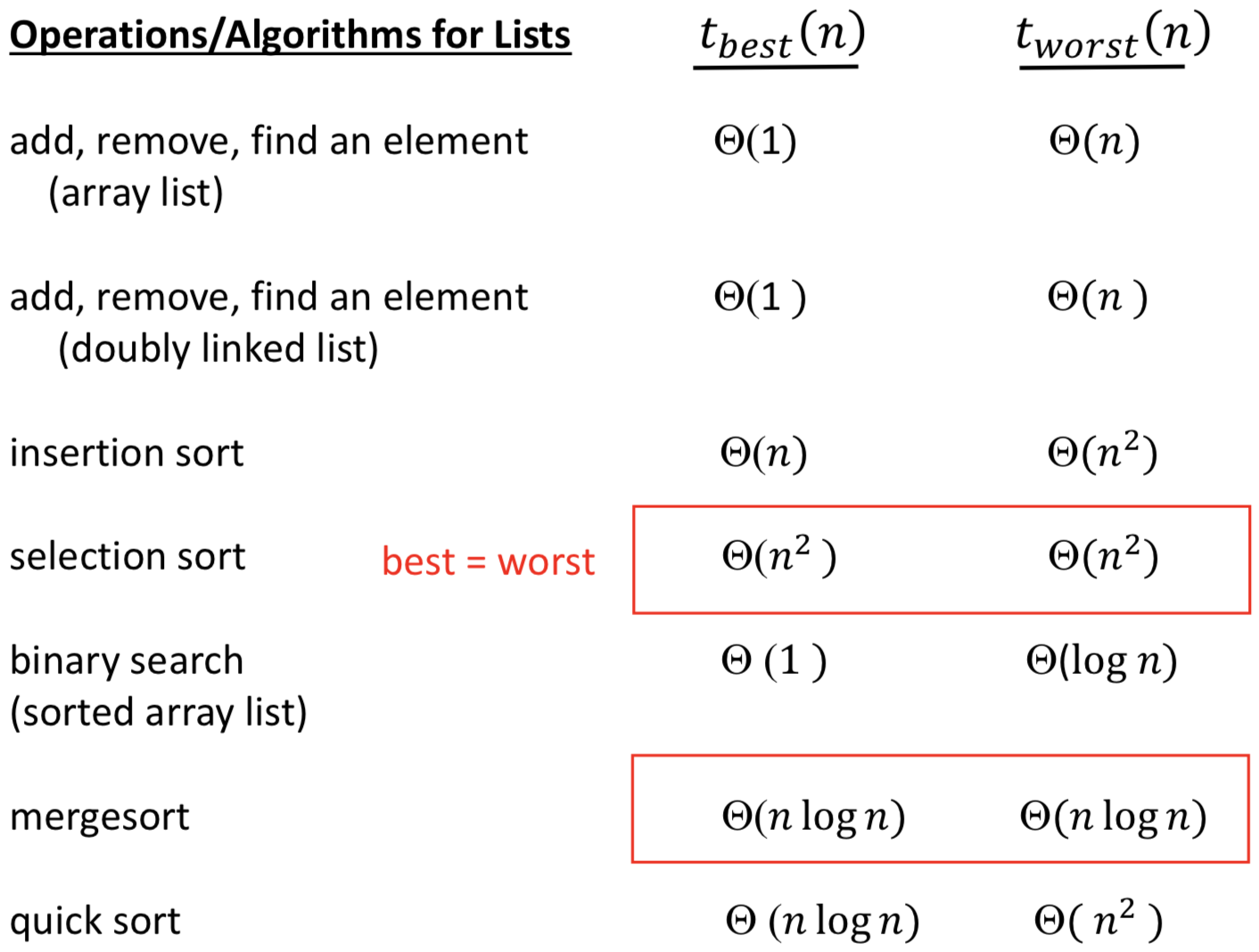
We say 𝑡(𝑛) is Q(𝑔(𝑛)) if 𝑡(𝑛) is both O(𝑔(𝑛)) and Ω (𝑔(𝑛)).



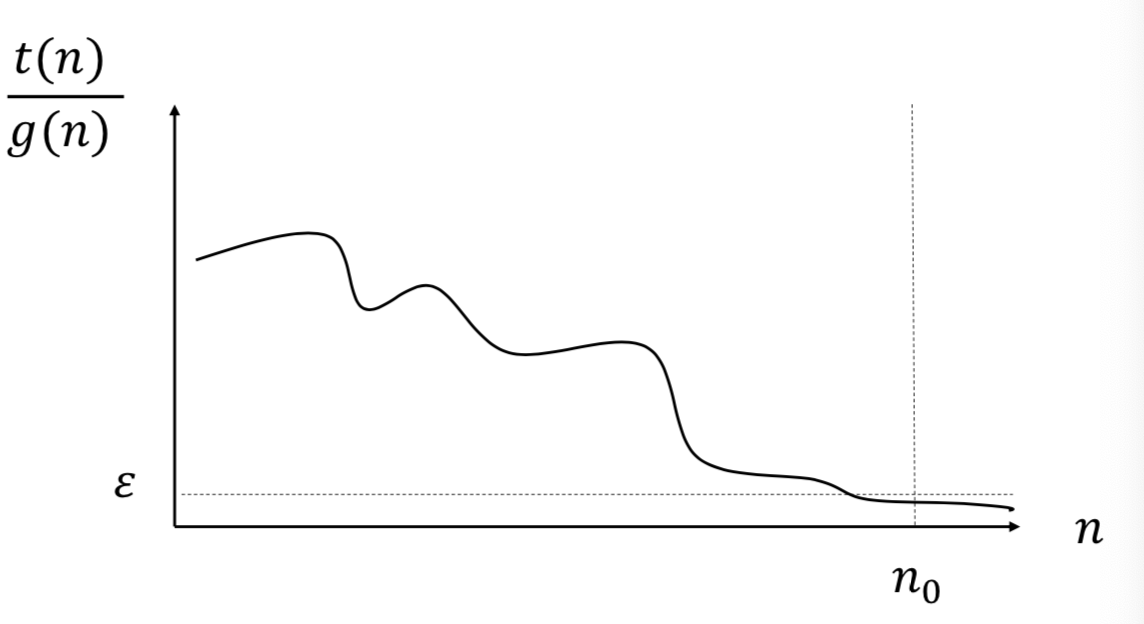




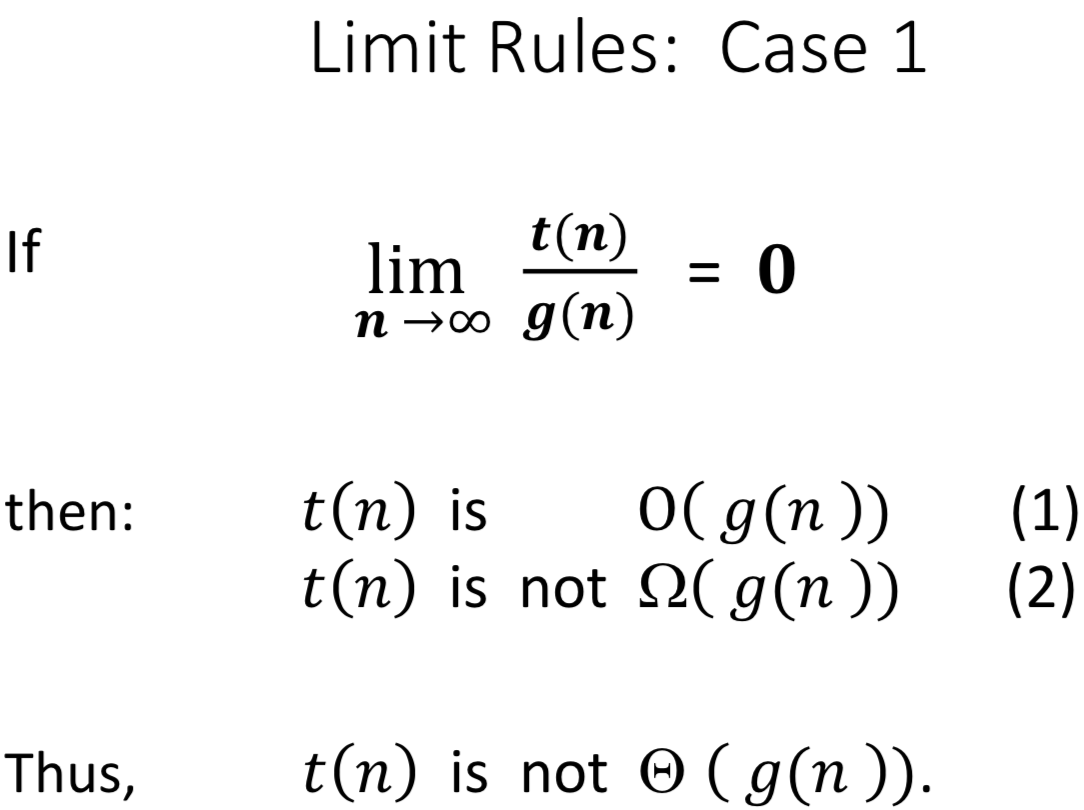




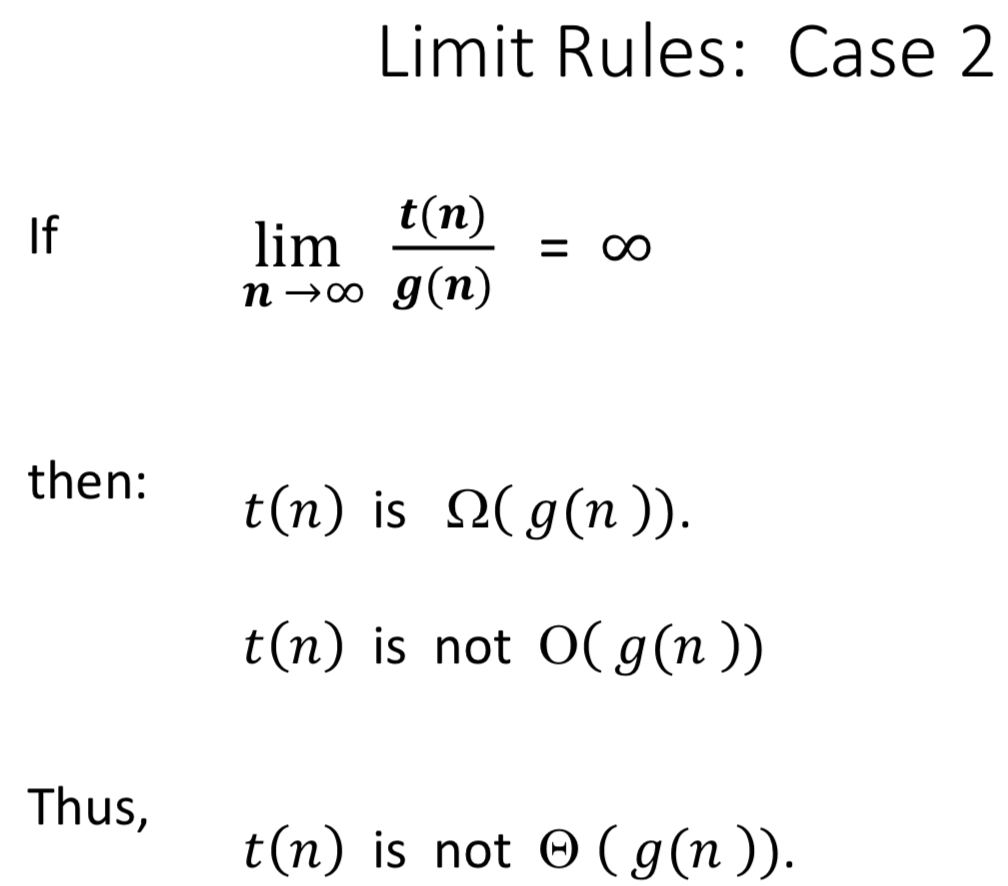
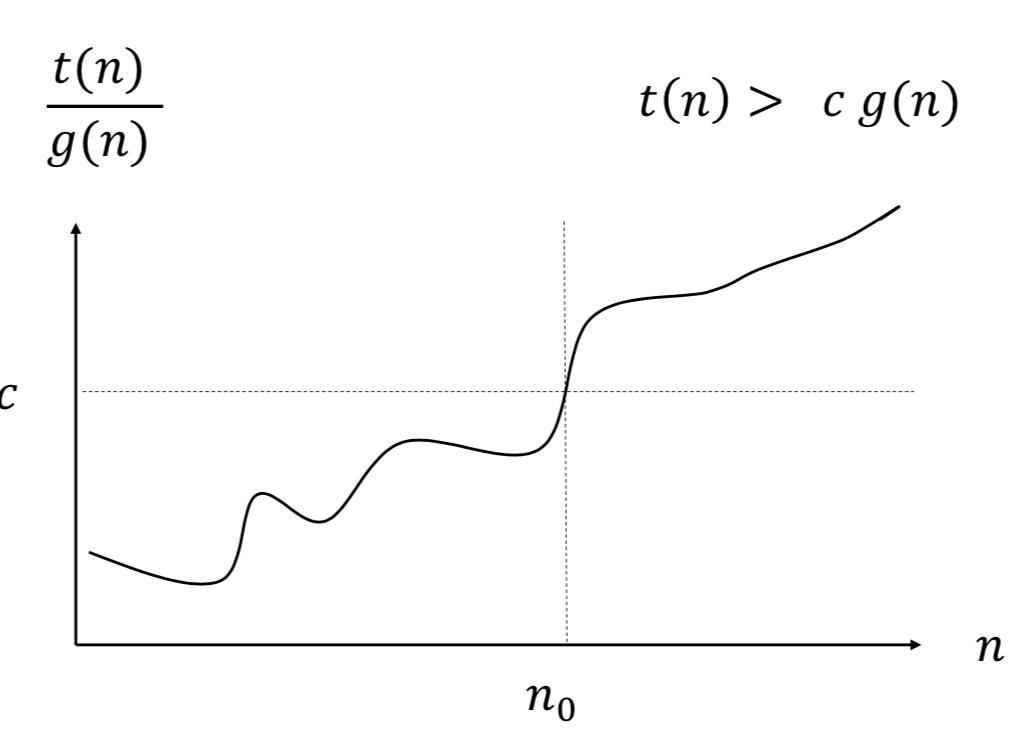
Limits



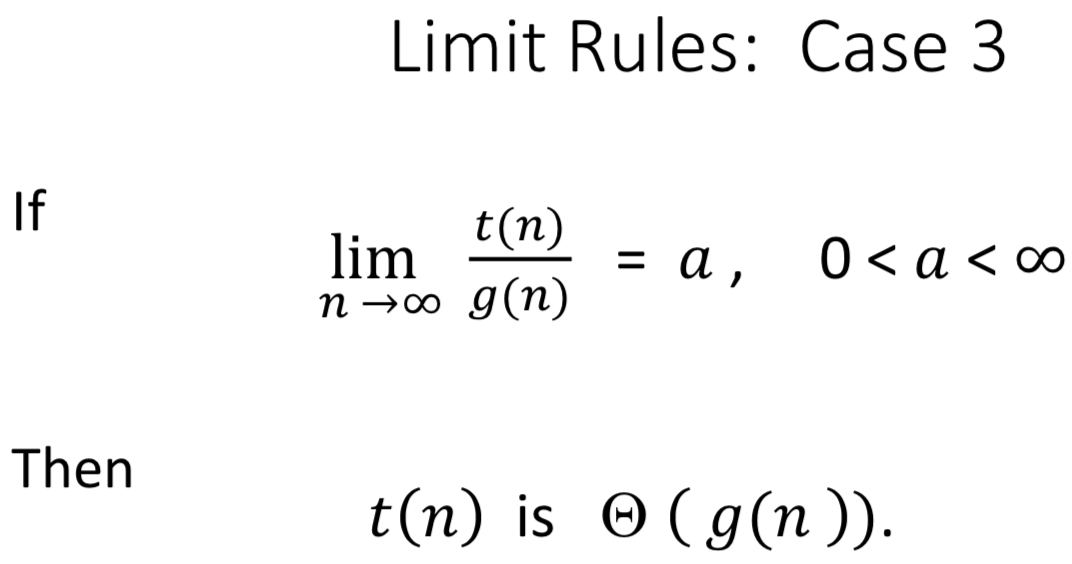
**Limit Rule #1**

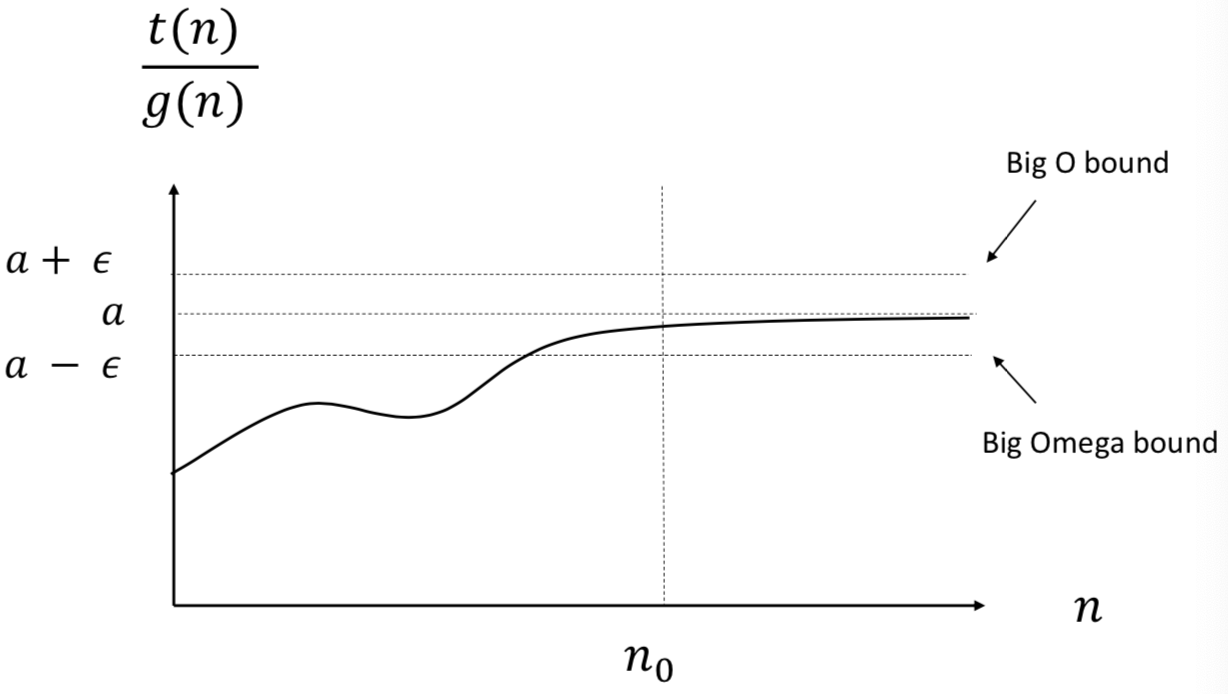


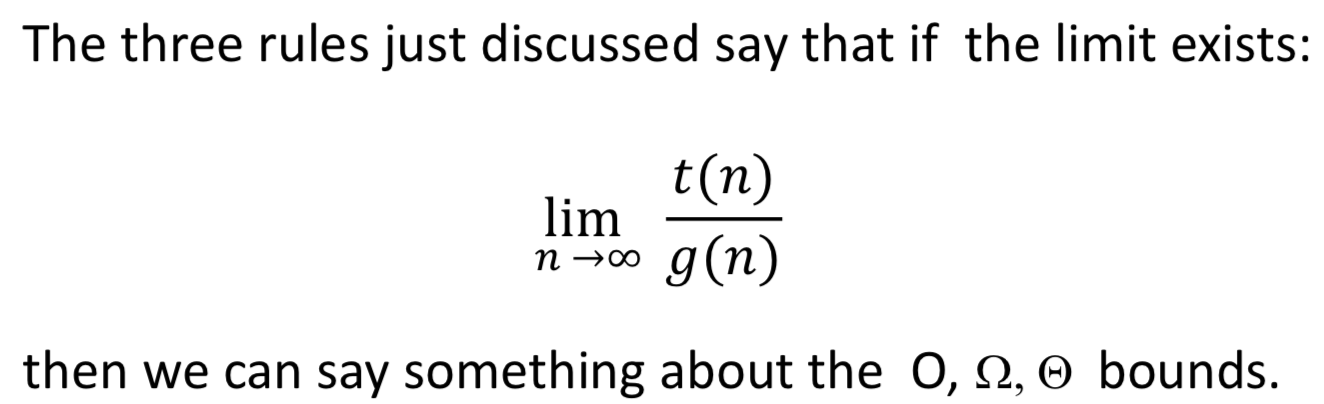
**Limit Rule #2**

**Limit Rule #3**







one direction

(the reverse is NOT true)