Problem Solving Strategies and Computational Approaches SCS1304

Handout 4 : Time Complexity & Asymptotic notation - Part II

Prasad Wimalaratne PhD(Salford),SMIEEE

Time complexity

- Time complexity is a programming term that quantifies the amount of time it takes a sequence of code or an algorithm to process or execute in proportion to the size and cost of input.
- It will not look at an algorithm's overall execution time. Rather, it will provide data on the variation (increase or reduction) in execution time when the number of operations in an algorithm increases or decreases.

Asymptotic Notation

- Asymptotic notation is a way to describe the performance of algorithms, specifically how their runtime or memory usage (complexity) grows as the input size increases.
- It focuses on the dominant behavior of the algorithm as the input approaches infinity, ignoring constant factors and lower-order terms.
- Essentially, it provides a shorthand for understanding the efficiency of an algorithm in the "long run“
- Big-O notation represents the upper bound of the running time of an algorithm. Thus, it gives the worst-case complexity of an algorithm.

Asymptotic Notations
What does asymptotic mean?

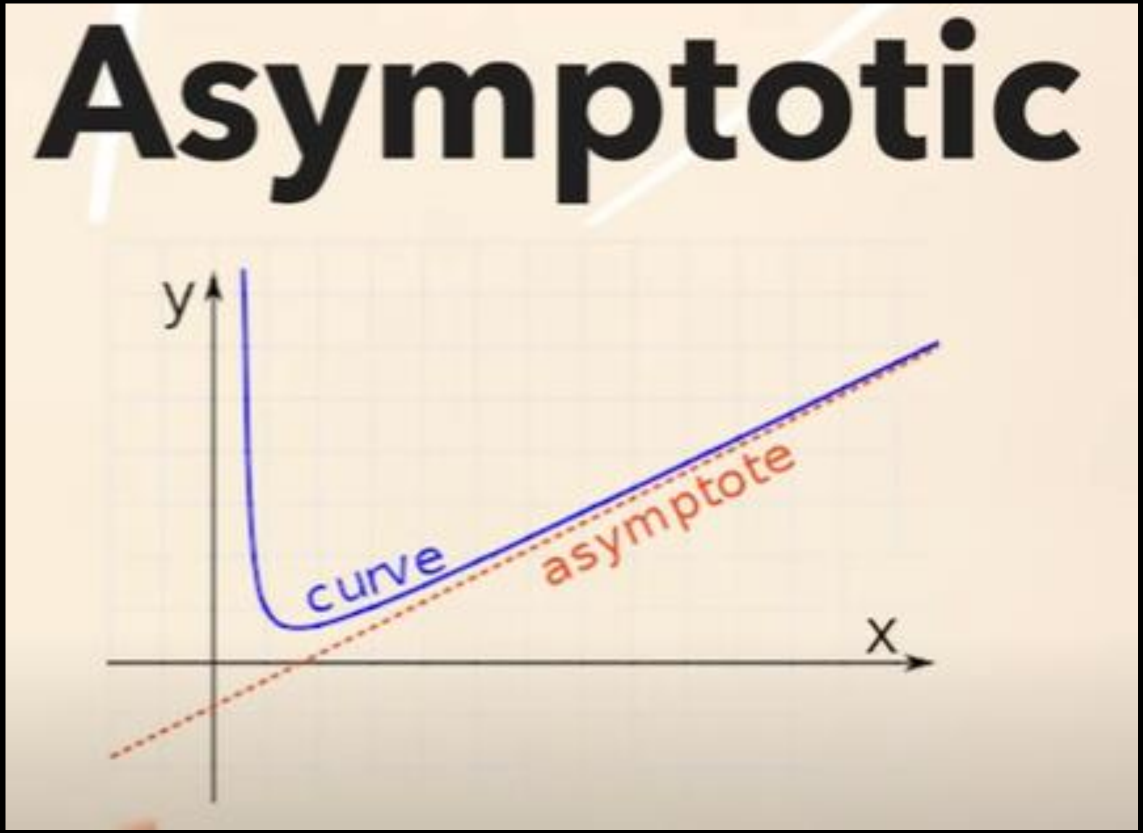
- The word asymptotic means approaching a certain value which could be considered as the limit.
  - The value can range from any finite natural number to an infinite value.

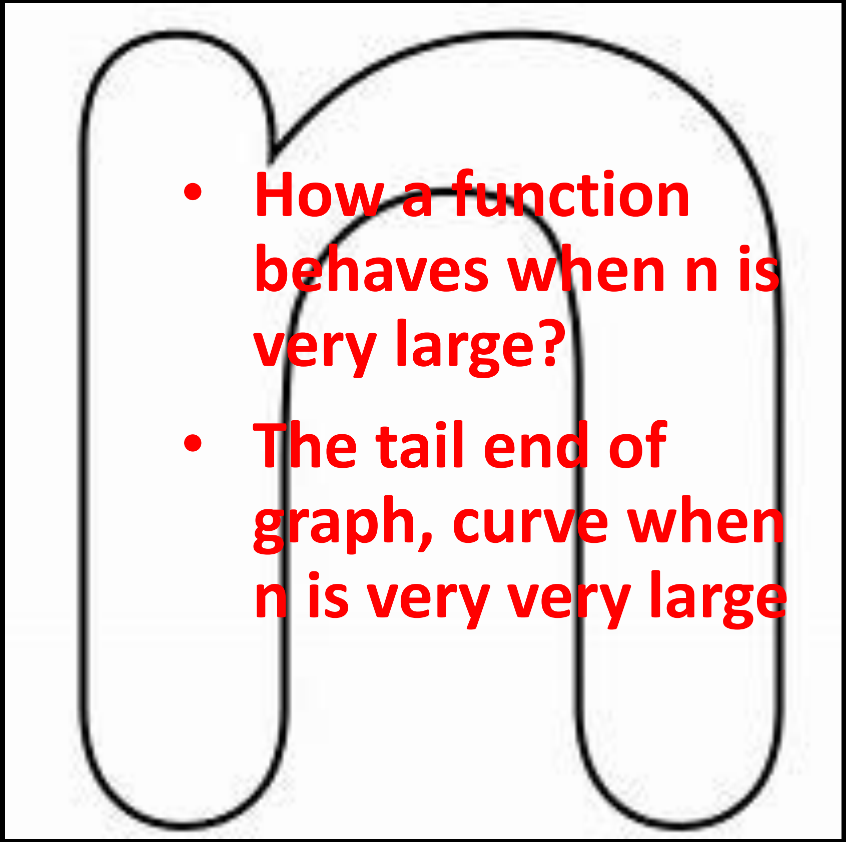
- Definition

  "Asymptotic Notation is used to decide the asymptotic running time of an algorithm, when the input size n is very large, i.e, n-> ∞

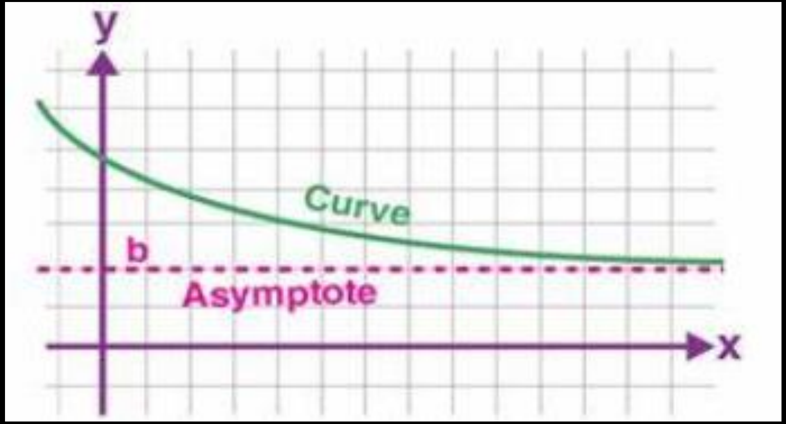
  ; 'n' belongs to the set of natural numbers. "

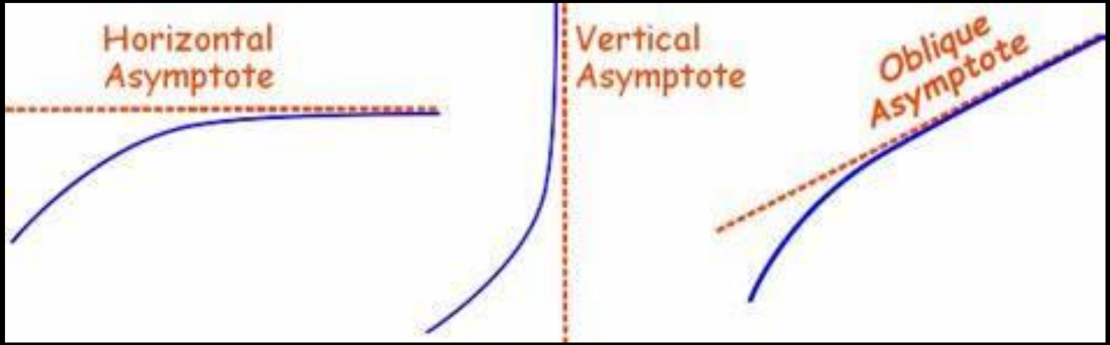
  - An asymptote of a curve is a line that the curve approaches but never actually touches or intersects.
    - It is a line that the curve gets infinitely close to as it extends towards infinity





- How a function behaves when n is very large?
- The tail end of graph, curve when n is very very large





Rigorous Definition to Order: Big O

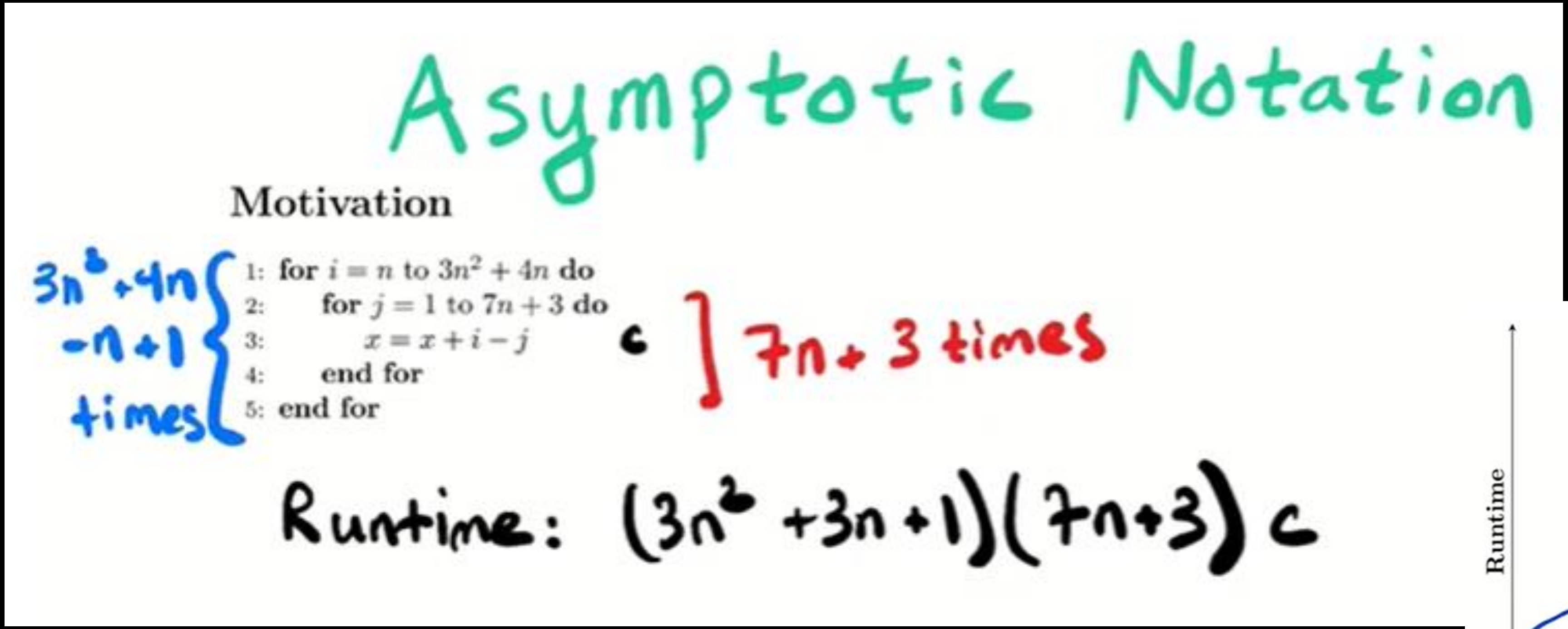
- Definition: (Asymptotic Upper Bound)
  - For a given complexity function $f(n)$, $O(f(n))$ is the set of complexity functions $g(n)$ for which there exists some positive real constant c and **some** non-negative integer N such that for all $n \geq N$,

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

  - $g(n) \in O(f(n))$

**Note the word SOME in definition. i.e not unique

Refer: Asymptotic Notation (Play List)
https://www.youtube.com/playlist?list=PLQfaHkBRINswUNbAHUOwi1tTxFxi1xUII



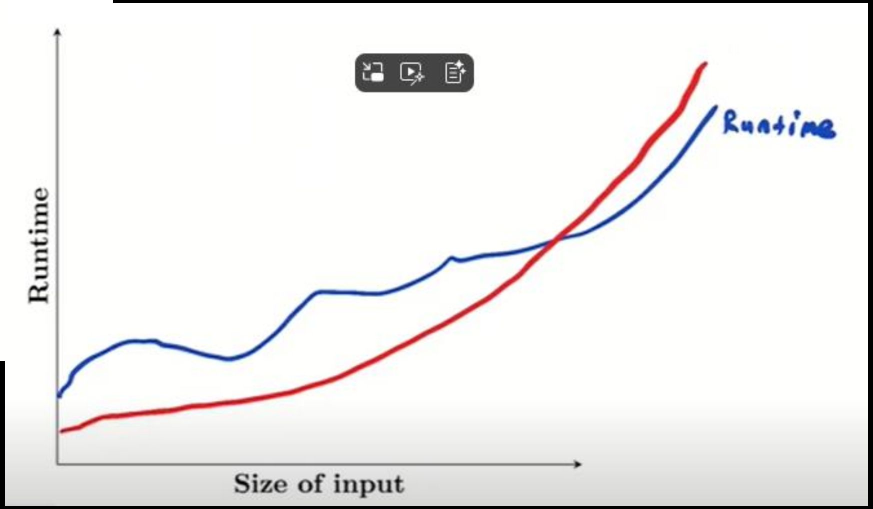
Asymptotic Notation

Motivation

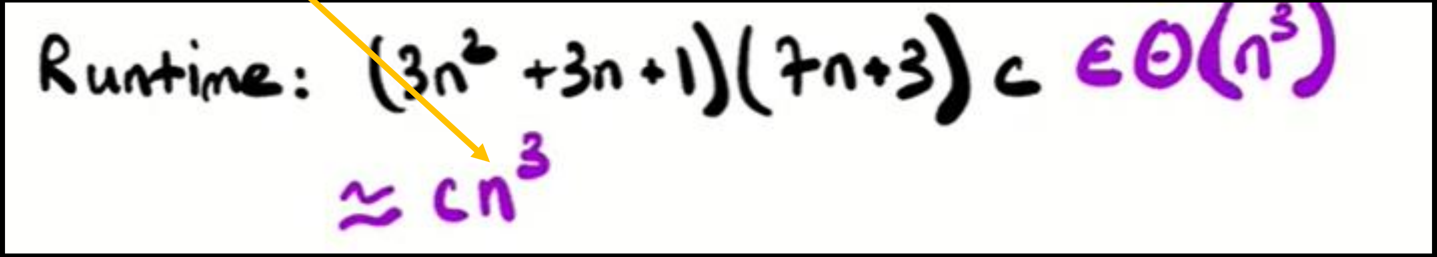
$3n^2+4n-n+1$ times

```
for $i = n$ to $3n^2 + 4n$ do
    for $j = 1$ to $7n + 3$ do
        $x = x + i - j$        c   ] 7n+3 times
    end for
end for
```

Runtime: $(3n^2+3n+1)(7n+3)c$



Fastest growing term



Runtime: $(3n^2+3n+1)(7n+3)c \in \Theta(n^3)$

$\approx cn^3$

Refer: Asymptotic Notation (Play List)
https://www.youtube.com/playlist?list=PLQfaHkBRINswUNbAHUOwi1tTxFxi1xUII

Meaning?

- Big O - what the slowest and most space an algo can use
- Omega – what the fastest and least space an algo can use
- ***   g(n) is **about the behavior** than a mathematical function.

Upper Bound O(n), Lower Bound and Average Bound of a class of functions

- Big O is written in the closest upper bound not higher though the equality holds
- Nearest class of functions is useful.

Asymptotic Notations

- The asymptotic notation is a defined pattern to measure the performance and memory usage of an algorithm

   i. Omega Notation “**Ω**”: best-case scenario where the time complexity will be as optimal as possible based on the input.

   ii. Theta Notation “**Θ**”: average-case scenario where the time complexity will be the average considering the input.

   iii. Big-O Notation “**O**”: worst-case scenario and the most used in coding interviews. It is the most important operator to learn because we can measure the worst-case scenario time complexity of an algorithm.

Illustrating "big O," Ω, and Θ.

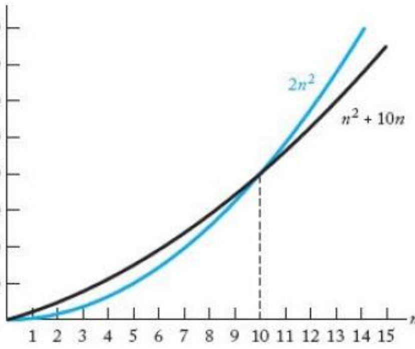
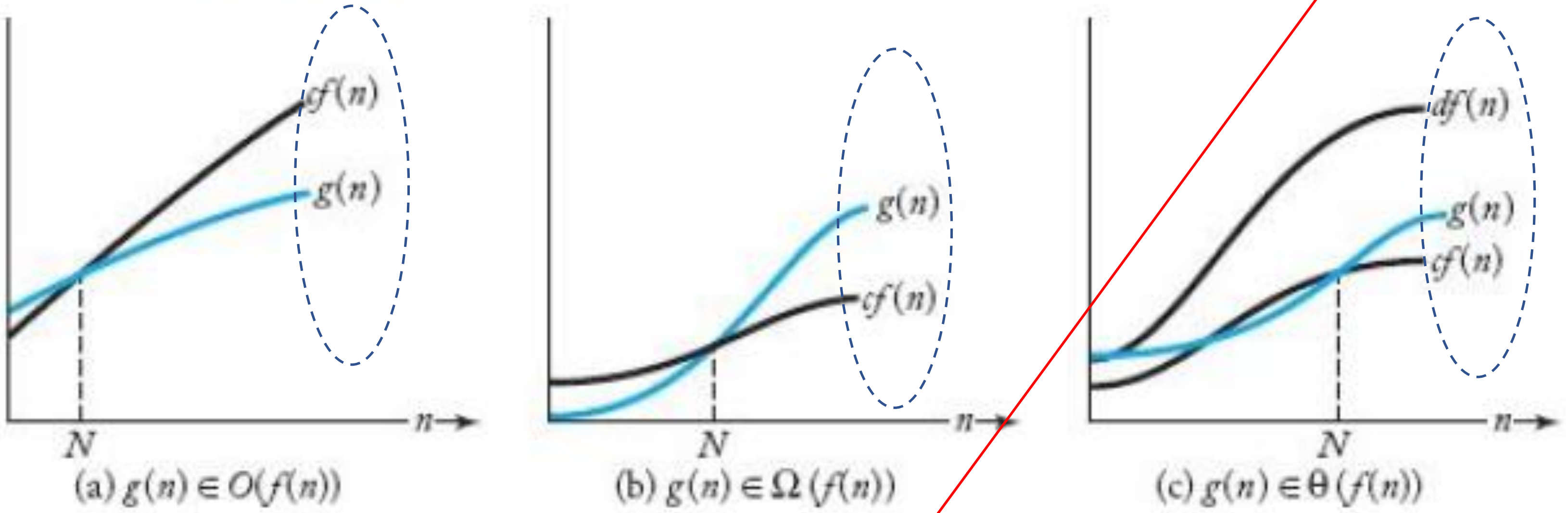


Figure 1.4 Illustrating "big O," Ω, and Θ.



$$n^2 + 10n \le 2n^2.$$

$$g(n) \le c \times f(n)$$

We can therefore take $c = 2$ and $N = 10$ in the definition of "big O" to conclude that

Substitute values for N

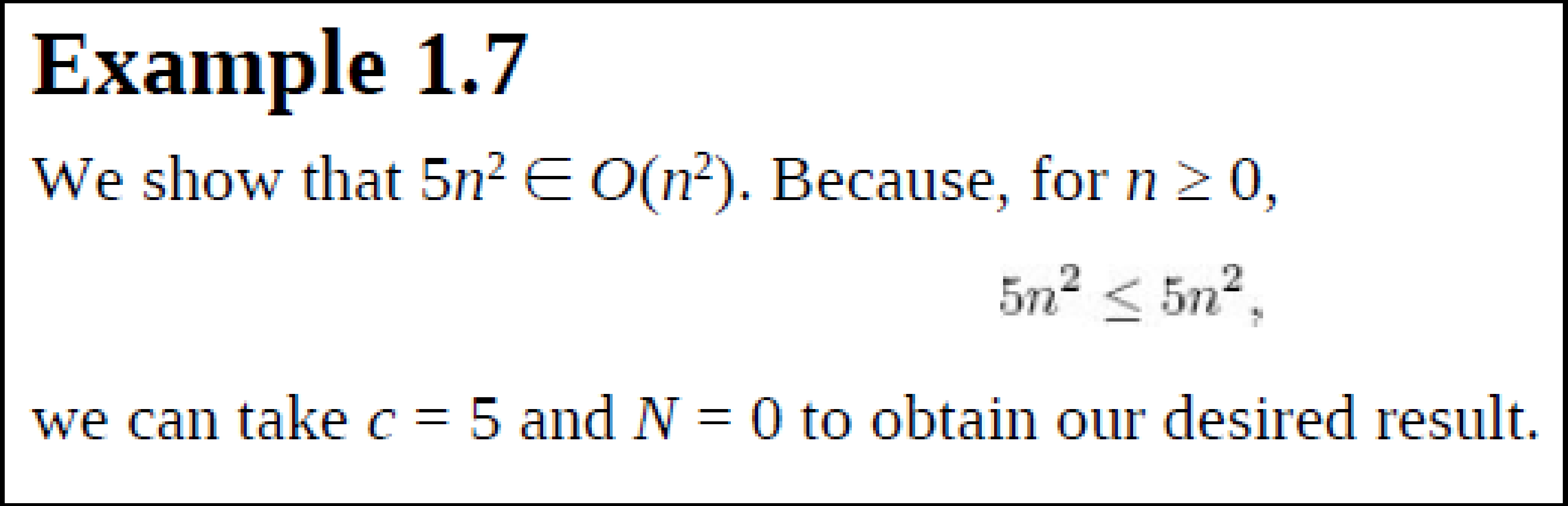
$$n^2 + 10n \in O(n^2).$$

g(n) is in O(n^2).

Meaning? This means that if g(n) is the time complexity for some algorithm, eventually the running time of the algorithm will be at least as fast as quadratic.

Illustrating “big *O*,” Ω, and Θ.

- If, for example, g(n) is in O(n^2), then eventually g(n) lies beneath some pure quadratic function **cn^2** on a graph.
  - This means that **if g(n) is the time complexity for some algorithm, eventually the running time of the algorithm will be at least as fast as quadratic.**
  - For the purposes of analysis, we can say that eventually g(n) is at least as good as a pure quadratic function.
- “Big O” (and Big Ω, and Θ) are said to describe the asymptotic behavior of a function because they are concerned only with eventual behavior.
- **We say that “big O” puts an asymptotic upper bound on a function.



Example 1.7

We show that $5n^2 \in O(n^2)$. Because, for $n \geq 0$,

$$5n^2 \leq 5n^2,$$

we can take $c = 5$ and $N = 0$ to obtain our desired result.

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

Example 1.8

Recall that the time complexity of Algorithm 1.3 (Exchange Sort) is given by

$$T(n) = \frac{n(n-1)}{2}.$$

Because, for $n \geq 0$,

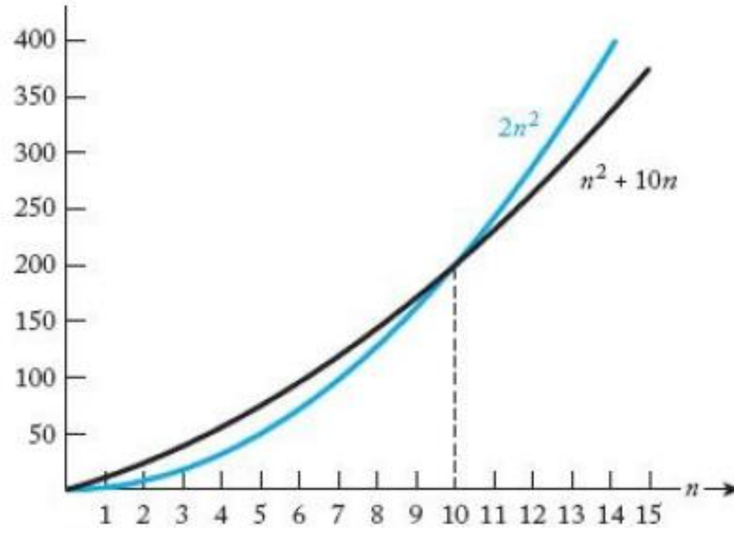
$$\frac{n(n-1)}{2} \leq \frac{n(n)}{2} = \frac{1}{2}n^2,$$

we can take $c = 1/2$ and $N = 0$ to conclude that $T(n) \in O(n^2)$.

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

A difficulty students often have with "big O" is that they erroneously think there is some unique c and unique N that must be found to show that one function is "big O" of another. This is not the case at all. Recall that Figure 1.5 illustrates that $n^2 + 10n \in O(n^2)$ using $c = 2$ and $N = 10$. Alternatively, we could show it as follows.

Figure 1.5 The function $n^2 + 10n$ eventually stays beneath the function $2n^2$.



Example 1.9

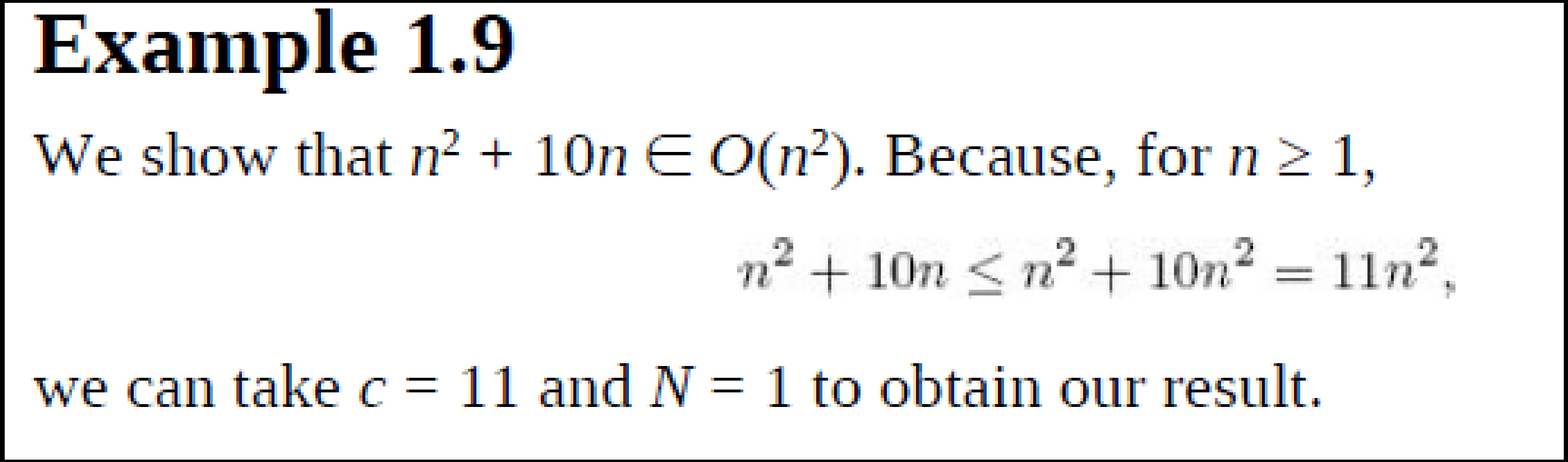
We show that $n^2 + 10n \in O(n^2)$. Because, for $n \geq 1$,

$$n^2 + 10n \leq n^2 + 10n^2 = 11n^2,$$

we can take $c = 11$ and $N = 1$ to obtain our result.

In general, one can show "big O" using **whatever manipulations seem most straightforward**

In general, one can show "big O" using whatever manipulations seem most straightforward.



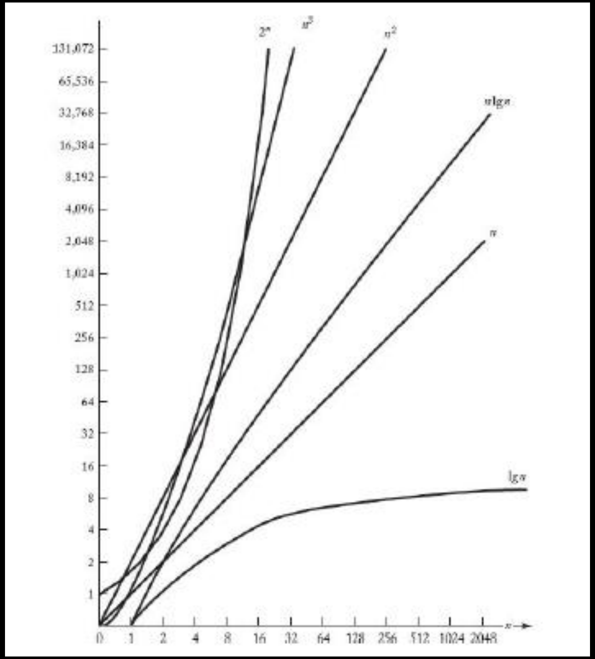
Example 1.9

We show that $n^2 + 10n \in O(n^2)$. Because, for $n \geq 1$,

$$n^2 + 10n \leq n^2 + 10n^2 = 11n^2,$$

we can take $c = 11$ and $N = 1$ to obtain our result.

- The purpose of this last example is to show that the function inside "big O" does not have to be one of the simple functions plotted in Figure 1.3.(below) It can be any complexity function.
- Ordinarily, however, we take it to be a simple function like those plotted in Figure 1.3 (below).

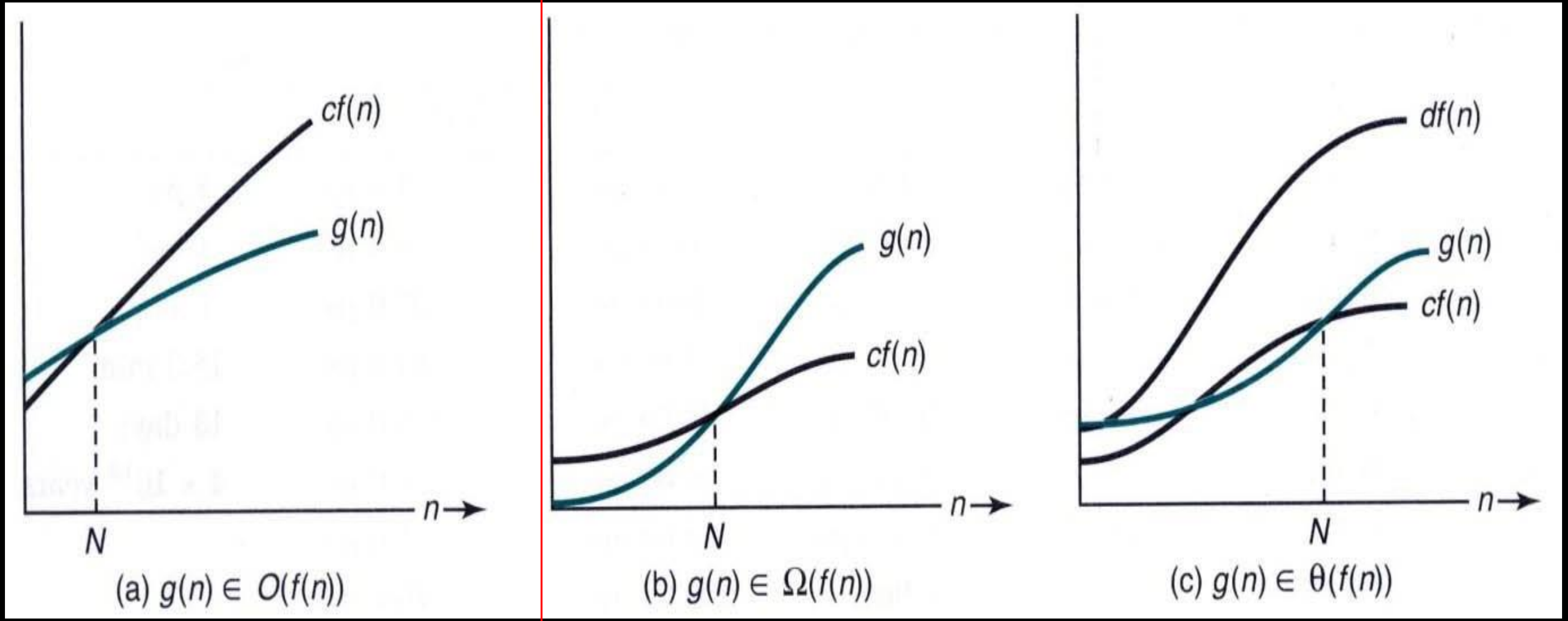


- Refer https://web.mit.edu/16.070/www/lecture/big_o.pdf

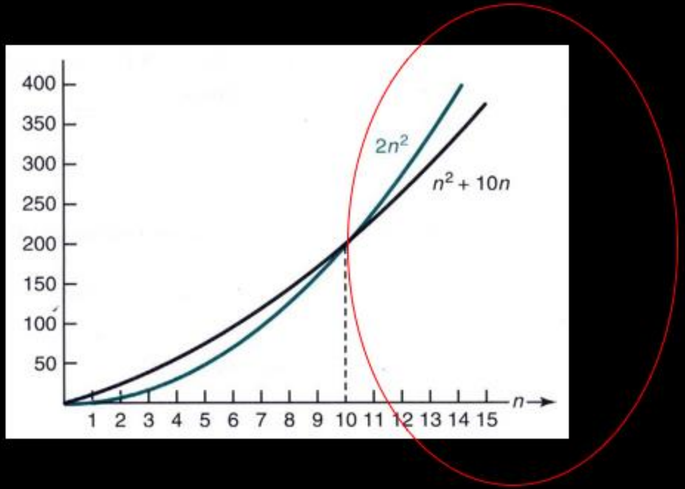
Big O Notation: Definition

Meaning of $g(n) \in O(f(n))$

- Although *g(n)* starts out above *cf(n)* in the figure, eventually it falls beneath *cf(n)* and stays there.
- If *g(n)* is the time complexity for an algorithm, eventually the running time of the algorithm will be at least as good as *f(n)*
- *f(n)* is called as an asymptotic upper bound (*of what?*) (i.e. *g(n)* cannot run slower than *f(n)*, eventually)

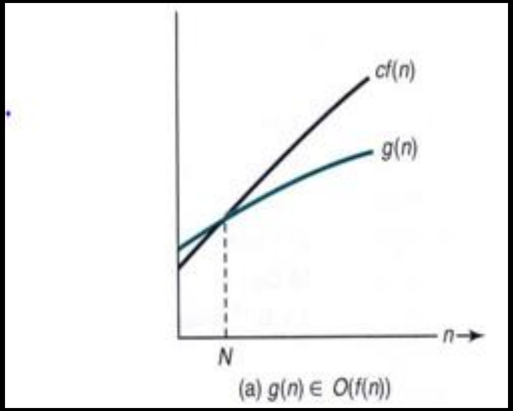


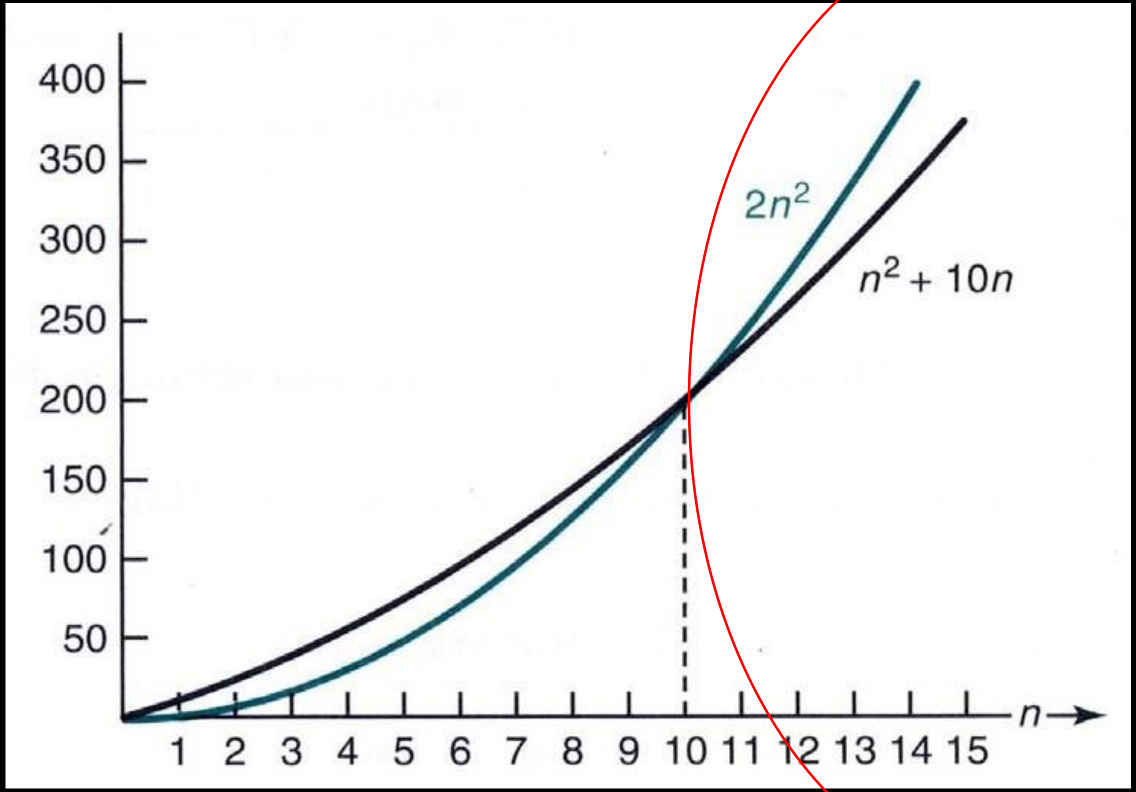
Big O Notation: Example



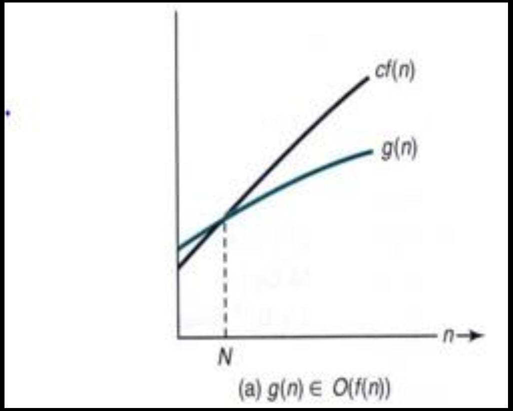
- Meaning of $n^2+10n \in O(n^2)$
  - Take *c* = *11* and *N* = *1*.
  - Take *c* = *2* and *N* = *10*.
  - If n^2+10n is the time complexity for some algorithm, eventually the running time of the algorithm will be at least as fast (good) as n^2
  - *** $11n^2$ is an asymptotic upper bound for the time complexity function of n^2+10n.

Figure 1.5 The function $n^2 + 10n$ eventually stays beneath the function $2n^2$.





Big O Notation: More Examples



(a) $g(n) \in O(f(n))$

- $5n^2 \in O(n^2)$
  - Take $c = 5$ and $N = 0$, then for all n such that $n \geq N$, $5n^2 \leq cn^2$.
- $T(n) = \frac{n(n-1)}{2}$
  - Because, for $n \geq 0$, $\frac{n(n-1)}{2} \leq \frac{n^2}{2}$
  - Therefore, we can take $c = ½$ and $N = 0$, to conclude that $T(n) \in O(n^2)$.

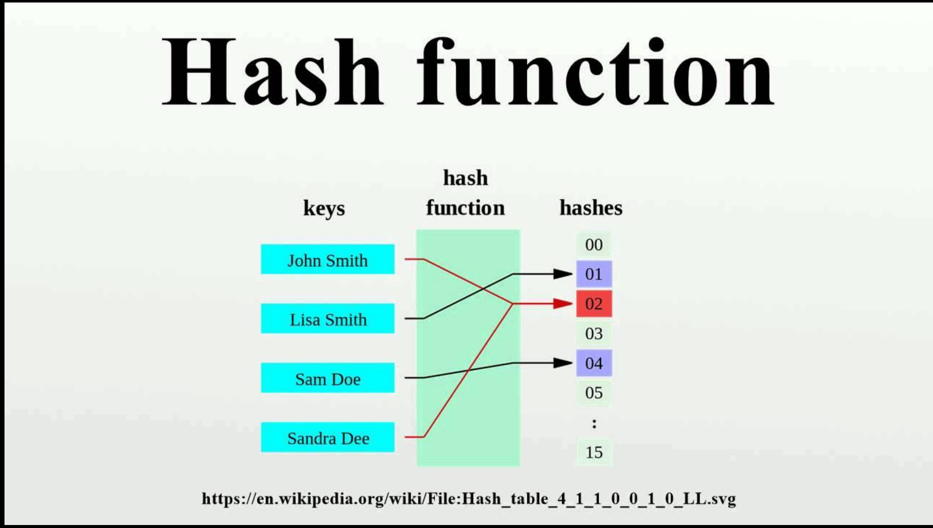
Big O Notation: More Examples (Cont'd)

- $n \in O(n^2)$
  - Take *c* = *1* and *N* = *1*, then for all n such that $n \geq N$, $n \leq 1 \times n^2$.
- $n^3 \in O(n^2)$??
  - Divide both sides by n^2
  - Then, we can obtain $n \leq c$
  - But it is impossible there exists a constant ***c*** that is large enough than a variable ***n***.
  - Therefore, n^3 does not belong to $O(n^2)$.

$$n^3 \notin O(n^2)$$

O(1) - Constant Time Examples

- In programming, a most operations are constant . Here are some examples:
  i. math operations/calculations e.g x = x+1
  ii. accessing an array via the index e.g x = S[i]
  iii. accessing a hash via the key
  iv. pushing and popping on a stack
  v. insertion and removal from a queue
  vi. returning a value from a function



Further Reading: [Hash Functions and Types of Hash functions](https://www.geeksforgeeks.org/hash-functions-and-list-types-of-hash-functions/)
https://www.geeksforgeeks.org/hash-functions-and-list-types-of-hash-functions/

O(n) → Linear Time
O(n^2) → Quadratic Time

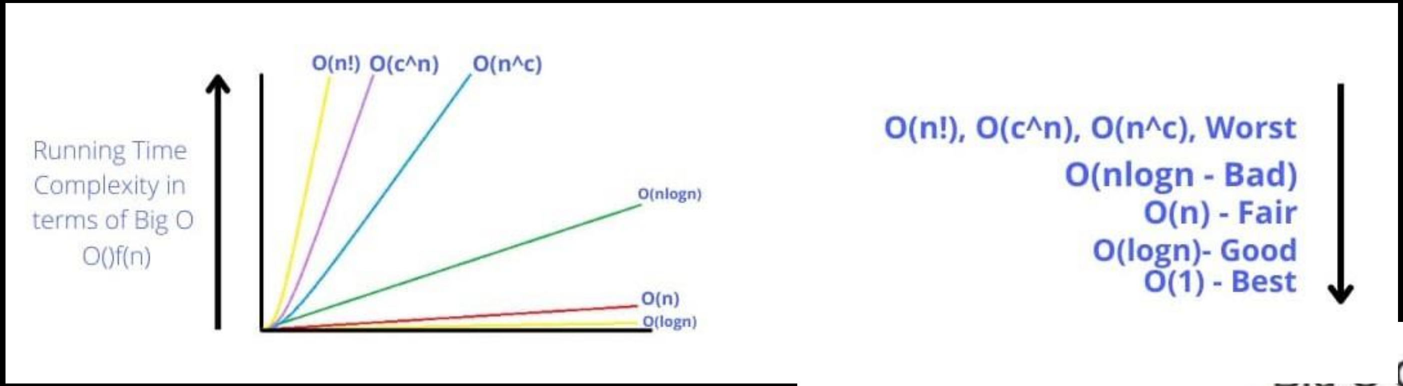
- O(**n**) means that the run-time increases at the same pace as the size of input.
- O(**n^2**) means that the calculation runs in quadratic time, which is the squared size of the input.
- In programming, many of the basic sorting algorithms have a worst-case run time of O(n^2):
  - e.g Bubble Sort, Insertion Sort, Selection Sort

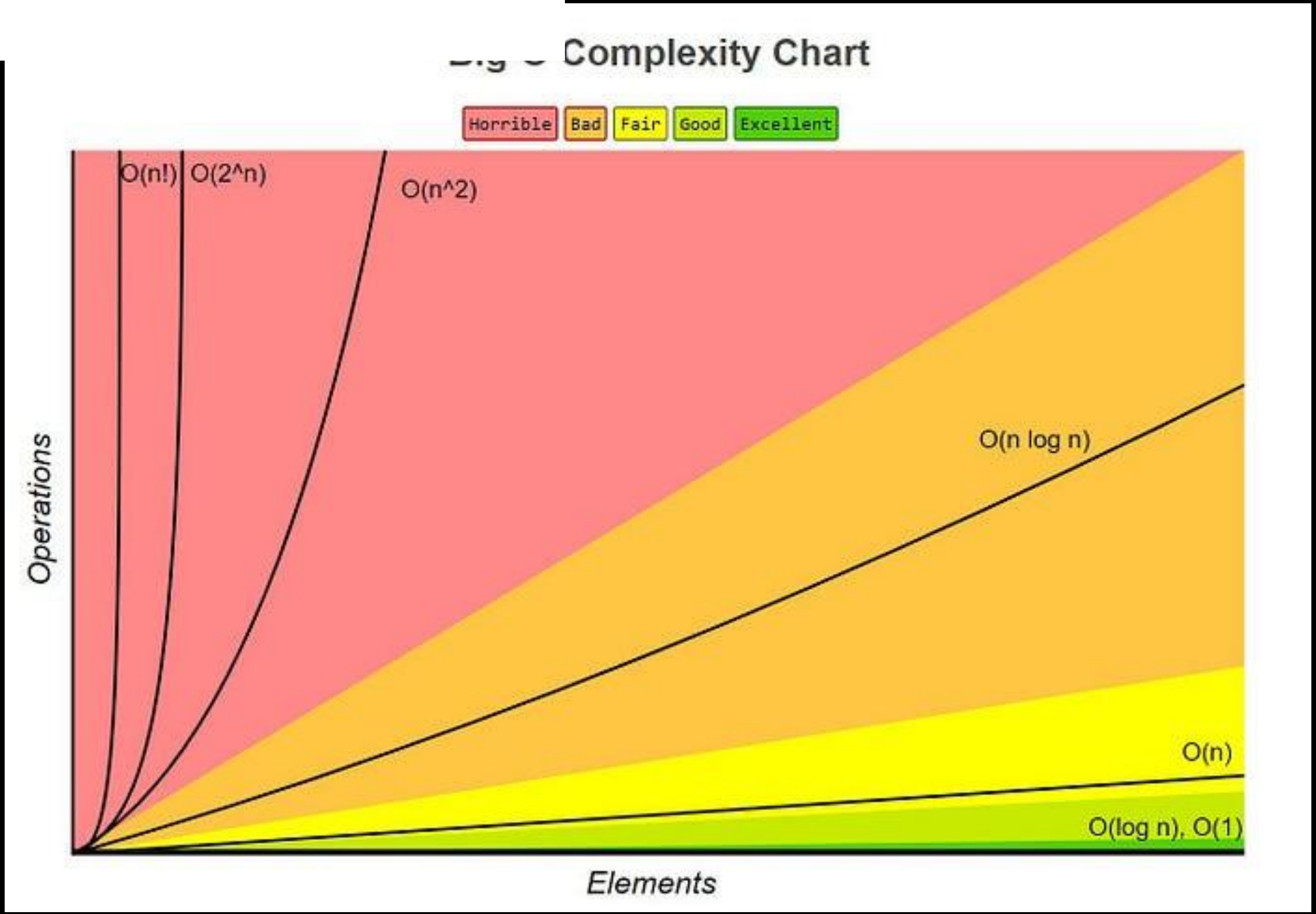
O(log n) → Logarithmic Time

- O(log n) means that the running time grows in proportion to the logarithm of the input size. this means that the run time barely increases as you exponentially increase the input.
- Note: O(n log n), which is often confused with O(log n), means that the running time of an algorithm is linearithmic, which is a combination of linear and logarithmic complexity.
- Sorting algorithms that utilize a divide and conquer strategy are linearithmic, such as the following:
  - e.g merge sort, timsort, heapsort
- When looking at time complexity, O(n log n) lands between O(n^2) and O(n).

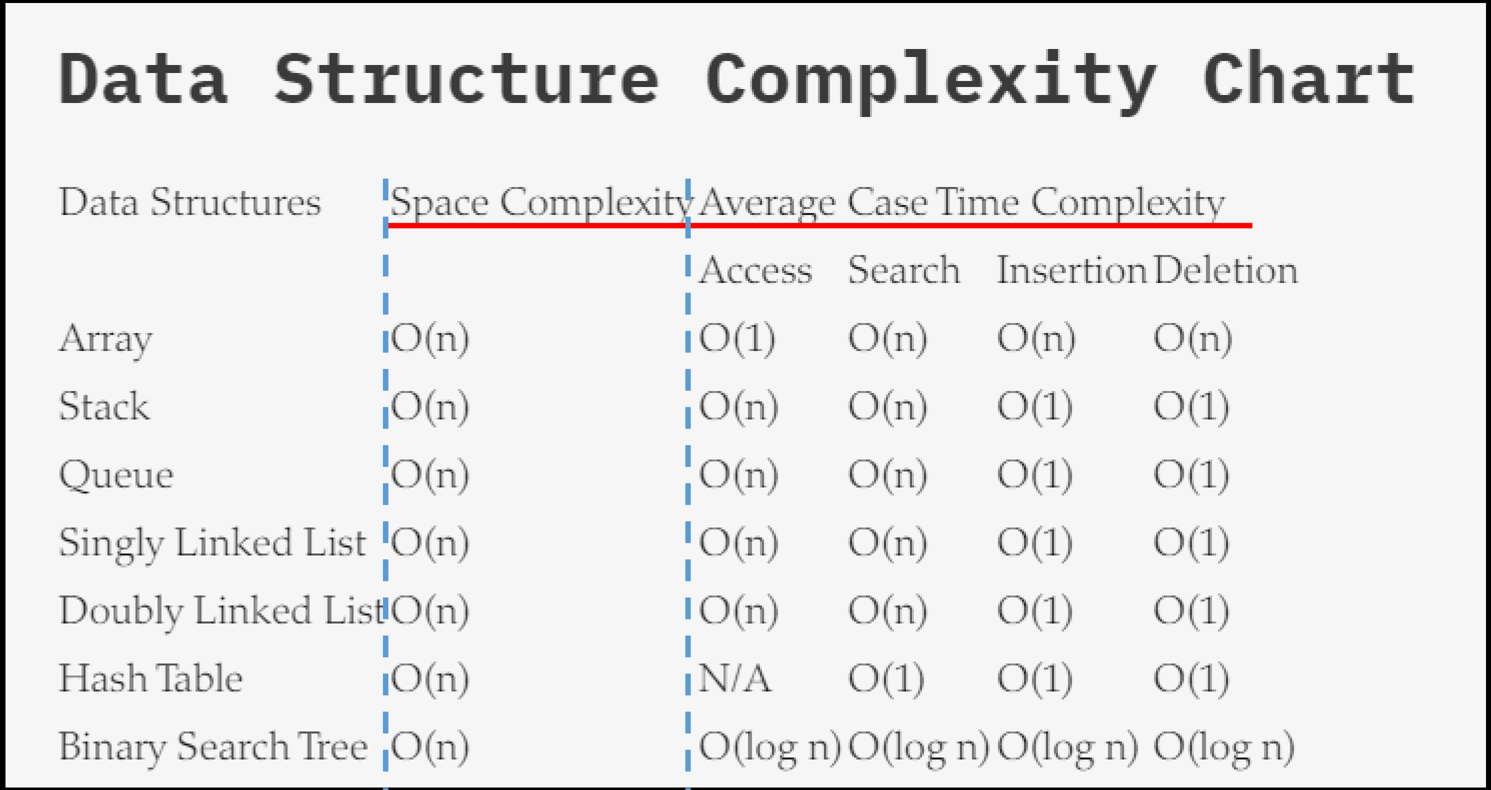
Factorial O(n!)

- The concept of factorial is simple. Suppose we have the factorial of 5! then this will equal 1 * 2 * 3 * 4 * 5 which results in 120.
  - A good example of an algorithm that has factorial time complexity is the array permutation. In this algorithm, we need to check how many permutations are possible given the array elements. For example, if we have 3 elements A, B, and C, there will be 6 permutations. Let us see how it works:
    - ABC, BAC, CAB, BCA, CAB, CBA – Notice that we have 6 possible permutations, the same as 3!.
    - If we have 4 elements, we will have 4! which is the same as 24 permutations, if 5! 120 permutations, and so on.



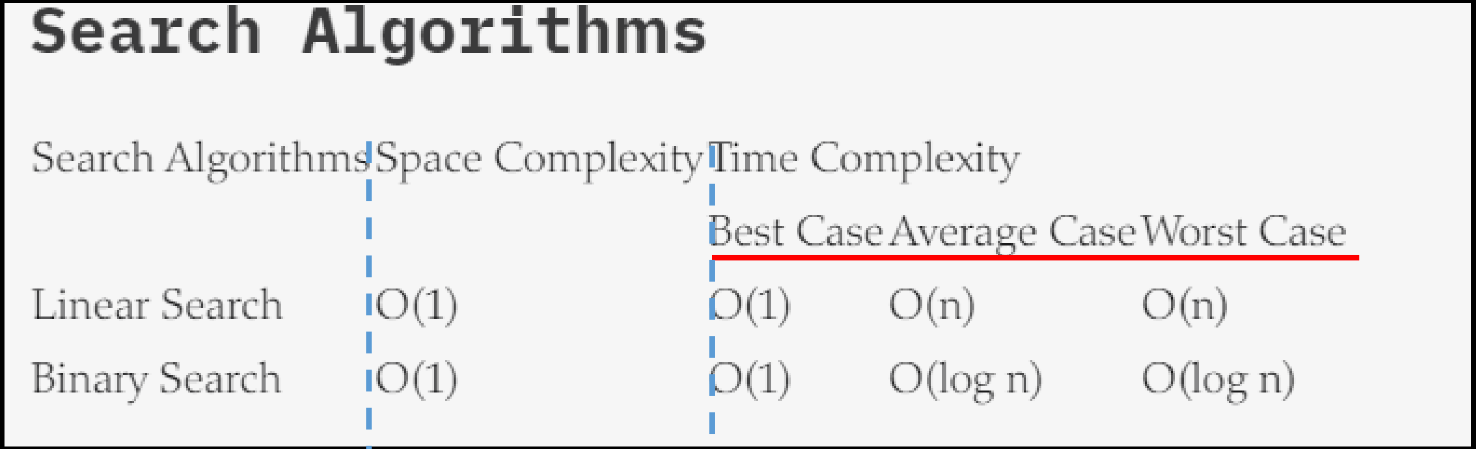


https://betterprogramming.pub/big-o-notation-a-simple-explanation-with-examples-a56347d1daca



Data Structure Complexity Chart

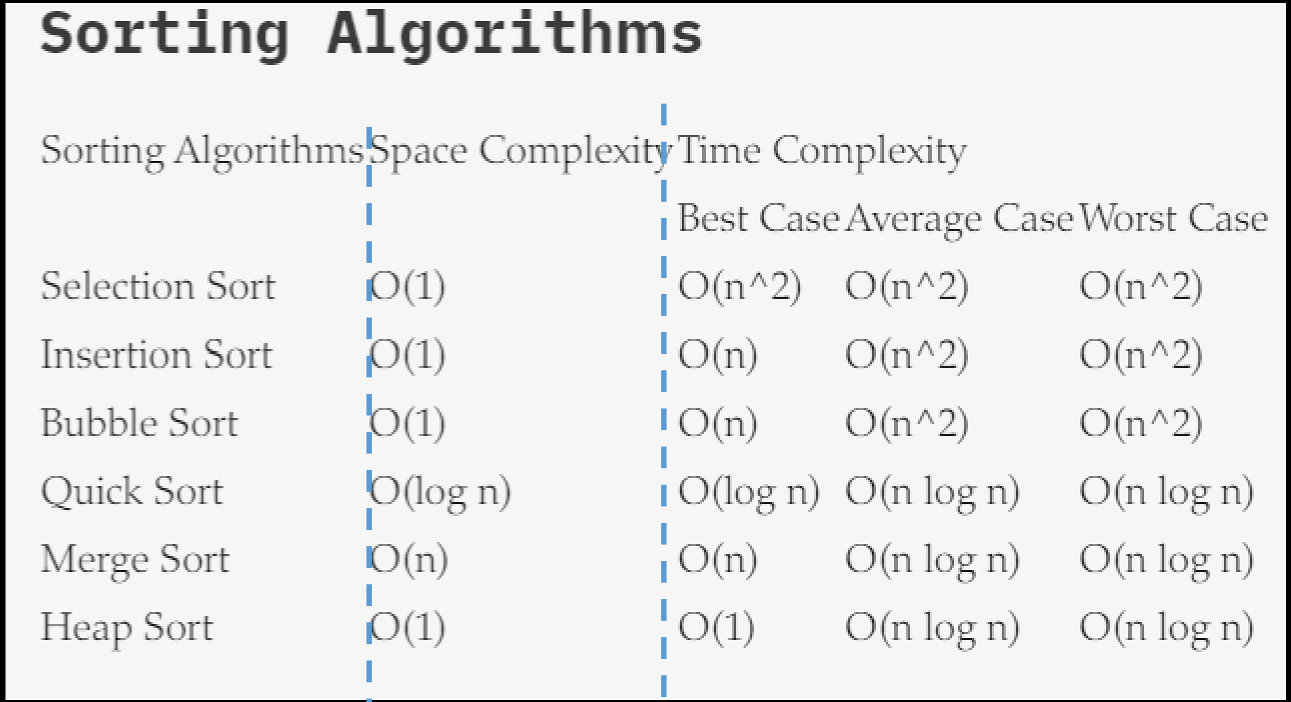
Data Structures	Space Complexity	Average Case Time Complexity			
		Access	Search	Insertion	Deletion
Array	O(n)	O(1)	O(n)	O(n)	O(n)
Stack	O(n)	O(n)	O(n)	O(1)	O(1)
Queue	O(n)	O(n)	O(n)	O(1)	O(1)
Singly Linked List	O(n)	O(n)	O(n)	O(1)	O(1)
Doubly Linked List	O(n)	O(n)	O(n)	O(1)	O(1)
Hash Table	O(n)	N/A	O(1)	O(1)	O(1)
Binary Search Tree	O(n)	O(log n)	O(log n)	O(log n)	O(log n)



Search Algorithms

Search Algorithms	Space Complexity	Time Complexity		
		Best Case	Average Case	Worst Case
Linear Search	O(1)	O(1)	O(n)	O(n)
Binary Search	O(1)	O(1)	O(log n)	O(log n)

https://flexiple.com/algorithms/big-o-notation-cheat-sheet

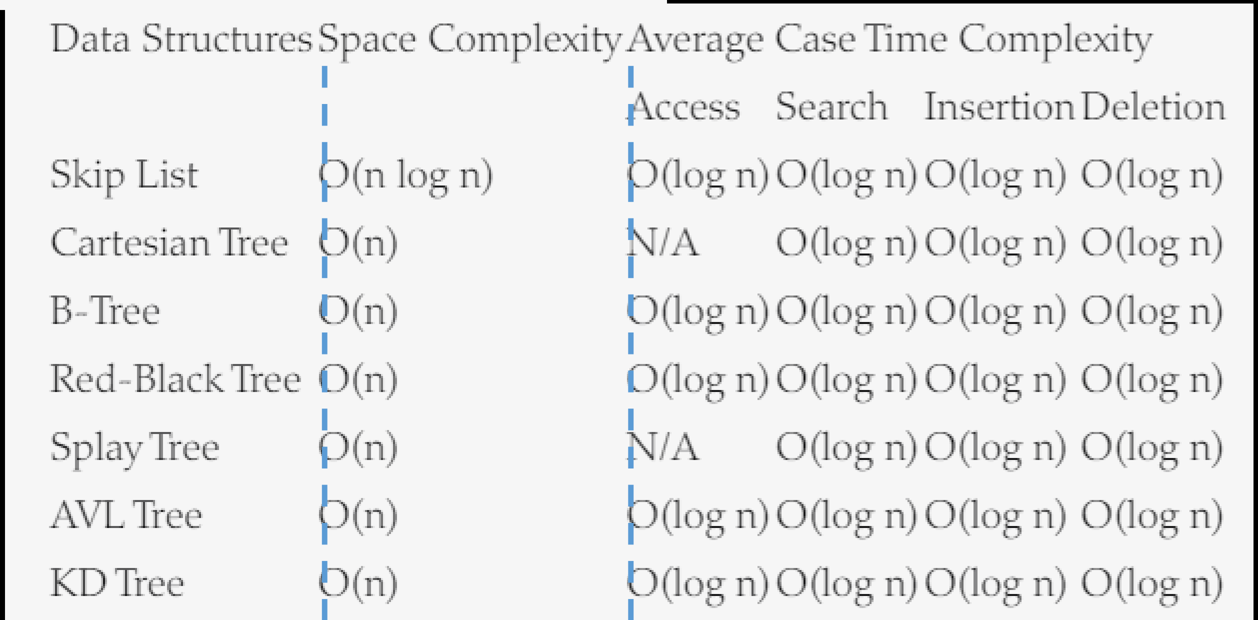


Sorting Algorithms

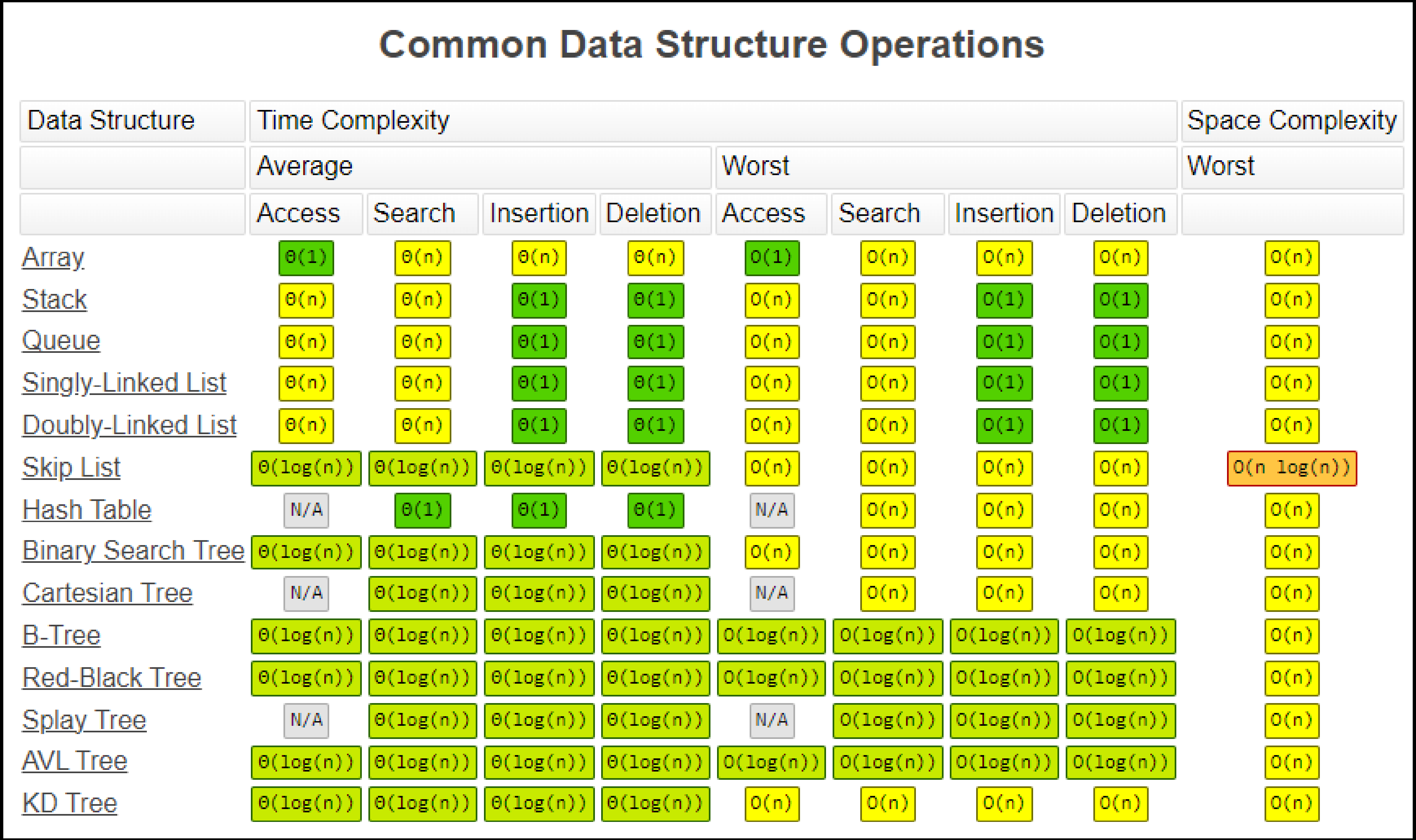
Sorting Algorithms	Space Complexity	Time Complexity		
		Best Case	Average Case	Worst Case
Selection Sort	O(1)	O(n^2)	O(n^2)	O(n^2)
Insertion Sort	O(1)	O(n)	O(n^2)	O(n^2)
Bubble Sort	O(1)	O(n)	O(n^2)	O(n^2)
Quick Sort	O(log n)	O(log n)	O(n log n)	O(n log n)
Merge Sort	O(n)	O(n)	O(n log n)	O(n log n)
Heap Sort	O(1)	O(1)	O(n log n)	O(n log n)

https://flexiple.com/algorithms/big-o-notation-cheat-sheet

Advanced Data Structures



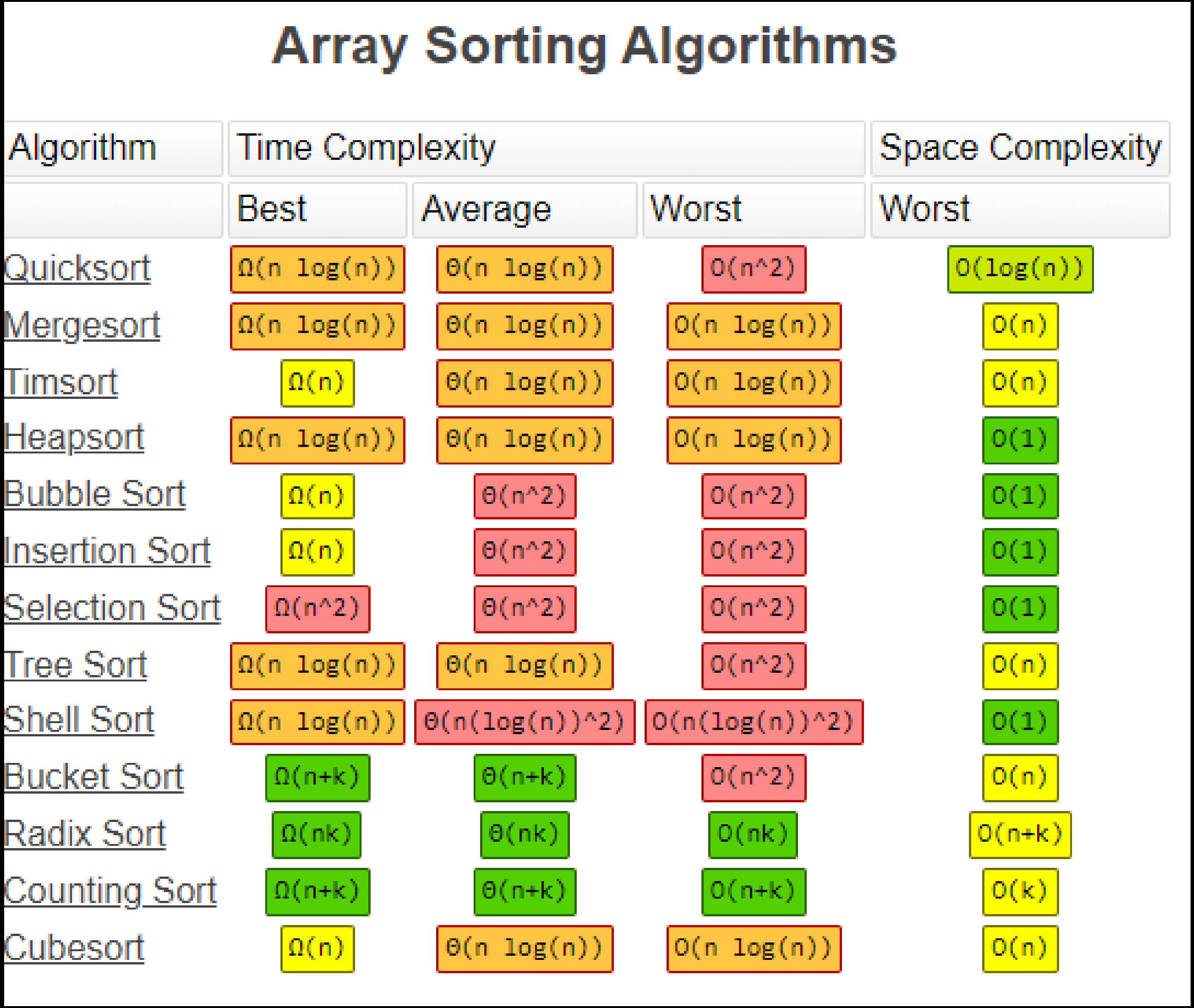
Data Structures	Space Complexity	Average Case Time Complexity			
		Access	Search	Insertion	Deletion
Skip List	O(n log n)	O(log n)	O(log n)	O(log n)	O(log n)
Cartesian Tree	O(n)	N/A	O(log n)	O(log n)	O(log n)
B-Tree	O(n)	O(log n)	O(log n)	O(log n)	O(log n)
Red-Black Tree	O(n)	O(log n)	O(log n)	O(log n)	O(log n)
Splay Tree	O(n)	N/A	O(log n)	O(log n)	O(log n)
AVL Tree	O(n)	O(log n)	O(log n)	O(log n)	O(log n)
KD Tree	O(n)	O(log n)	O(log n)	O(log n)	O(log n)



Common Data Structure Operations

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

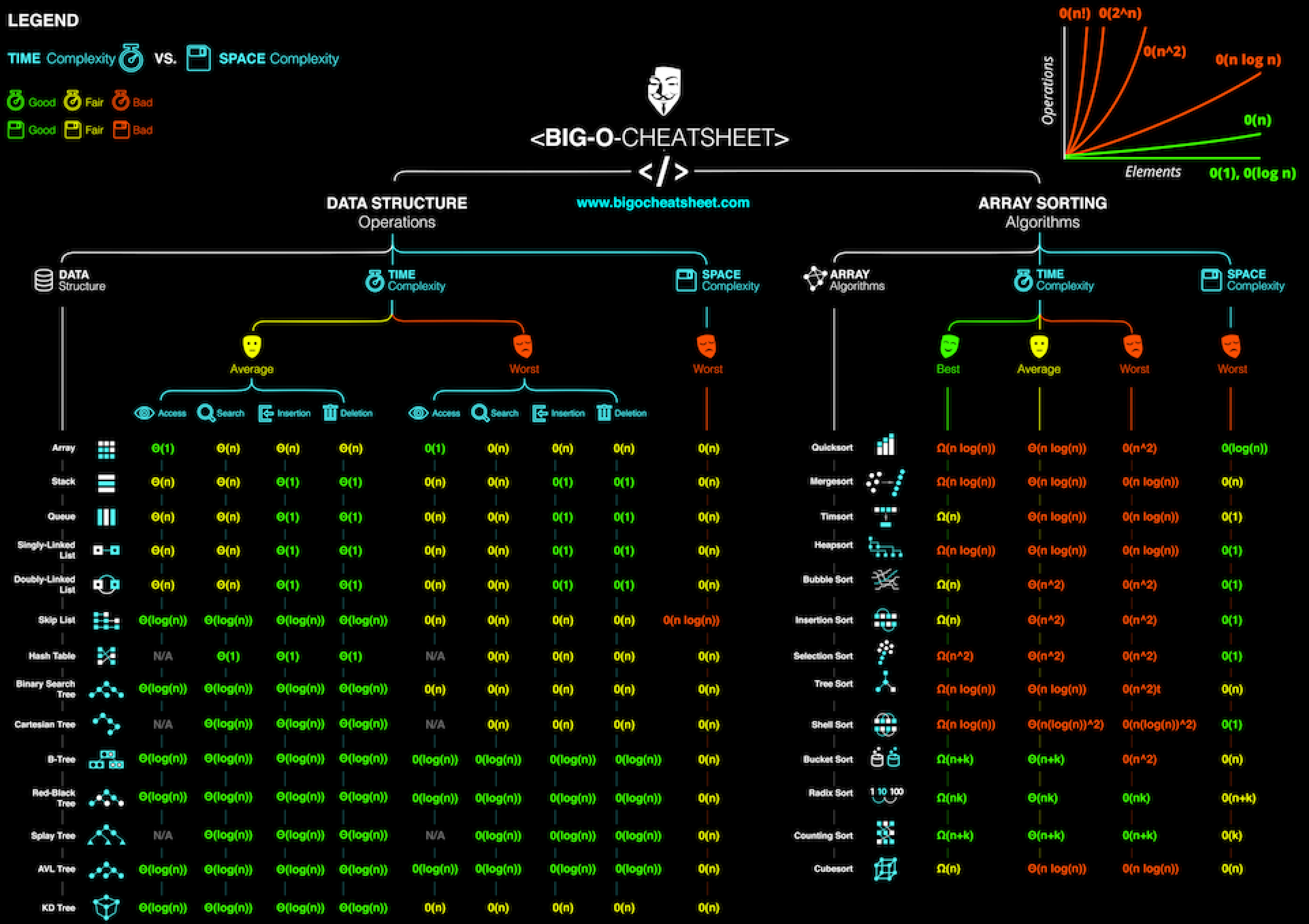
https://www.bigocheatsheet.com/



Array Sorting Algorithms

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	Θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	Θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	Θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	Θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)

https://www.bigocheatsheet.com/


O(n!) O(2^n)
O(n^2)
O(n log n)
O(n)
O(1), O(log n)
Operations
Elements

LEGEND

TIME Complexity VS. SPACE Complexity

🟩 Good 🟨 Fair 🟥 Bad

<BIG-O-CHEATSHEET>

</>

www.bigocheatsheet.com

DATA STRUCTURE Operations

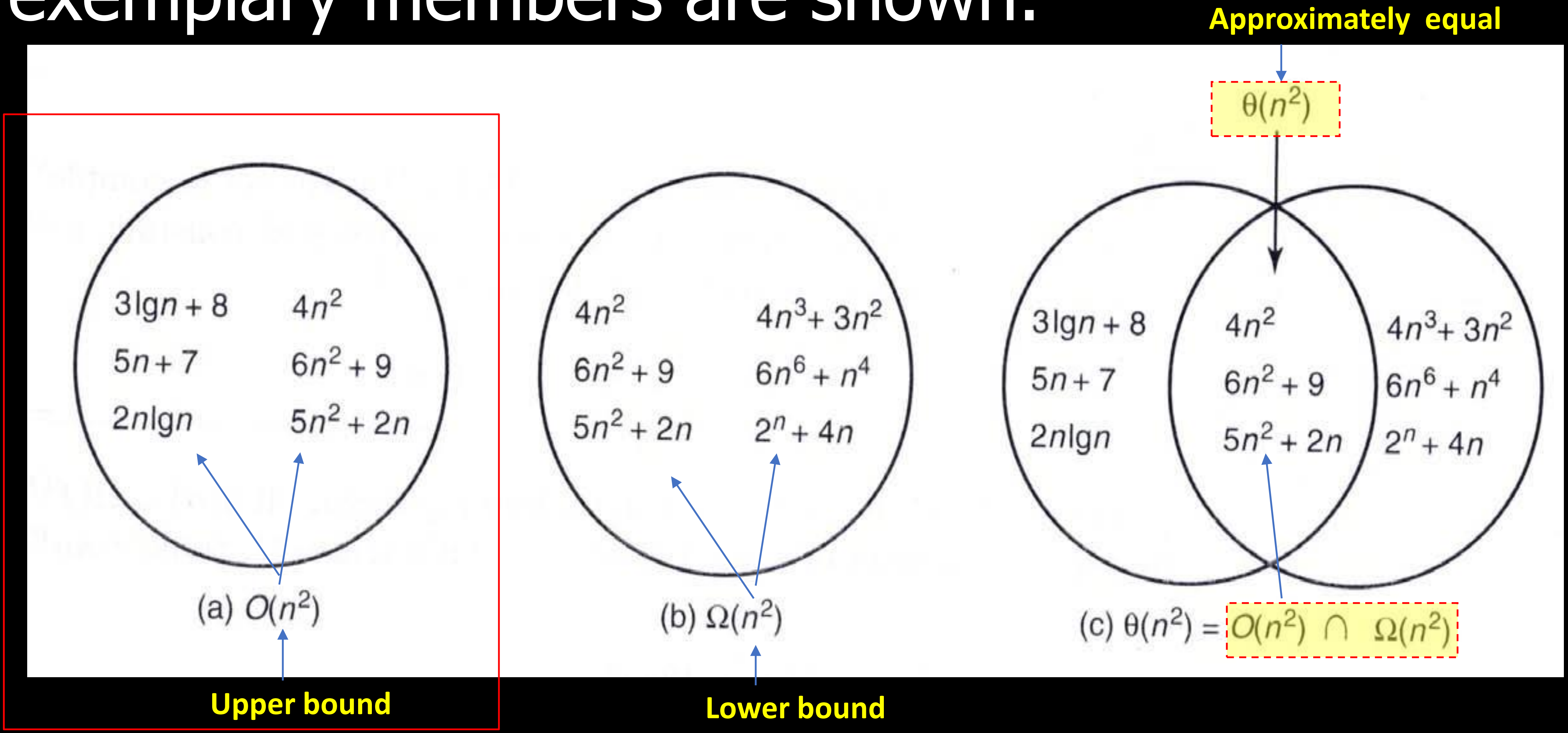
DATA Structure	TIME Complexity: Average				TIME Complexity: Worst				SPACE Complexity: Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

ARRAY SORTING Algorithms

ARRAY Algorithms	TIME Complexity: Best	TIME Complexity: Average	TIME Complexity: Worst	SPACE Complexity: Worst
Quicksort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	Θ(n log(n))	O(n log(n))	O(1)
Heapsort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	Θ(n log(n))	O(n^2)t	O(n)
Shell Sort	Ω(n log(n))	Θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	Θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	Θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	Θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)

https://www.bigocheatsheet.com/

Figure 1.6 The sets $O(n^2)$, $\Omega(n^2)$, $\Theta(n^2)$. Some exemplary members are shown.



Rigorous Definition to Order: Ω (Best Case)

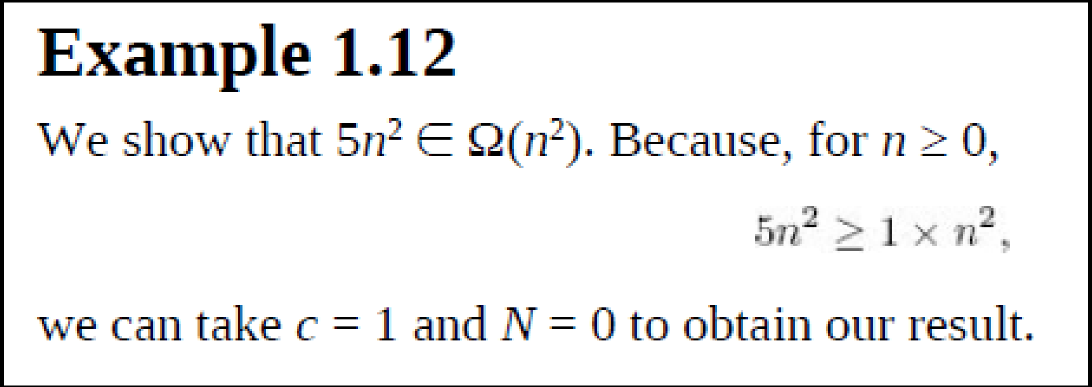
- Definition: (Asymptotic Lower Bound)
  - For a given complexity function $f(n)$, $\Omega(f(n))$ is the set of complexity functions $g(n)$ for which there exists some positive real constant c and some non-negative integer N such that for all $n \geq N$,

$$g(n) \geq c \times f(n)$$

  - $g(n) \in \Omega(f(n))$

Ω (Best Case)

- The symbol Ω is the Greek capital letter "omega." If g(n) ∈ Ω(f(n)), we say that g(n) is omega of f(n). Figure 1.4(b) illustrates Ω.

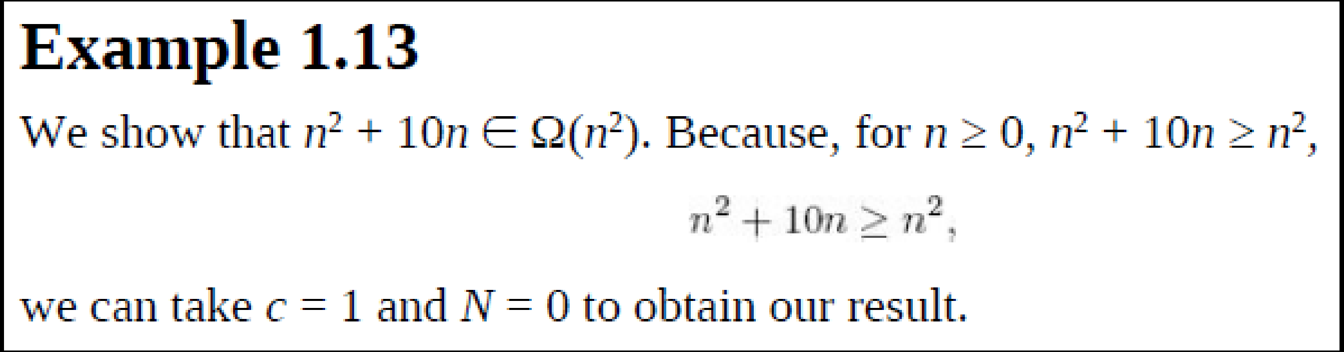


Example 1.12

We show that $5n^2 \in \Omega(n^2)$. Because, for $n \geq 0$,

$$5n^2 \geq 1 \times n^2,$$

we can take $c = 1$ and $N = 0$ to obtain our result.

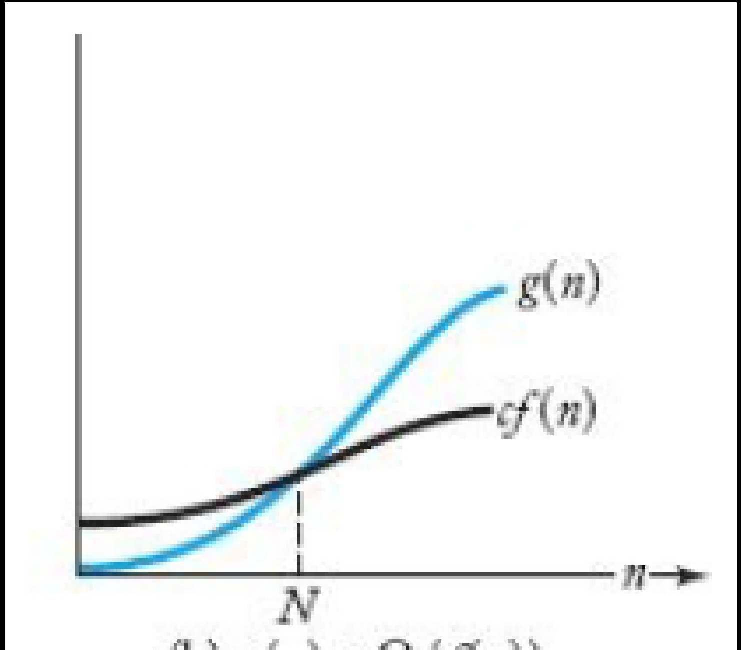


Example 1.13

We show that $n^2 + 10n \in \Omega(n^2)$. Because, for $n \geq 0$, $n^2 + 10n \geq n^2$,

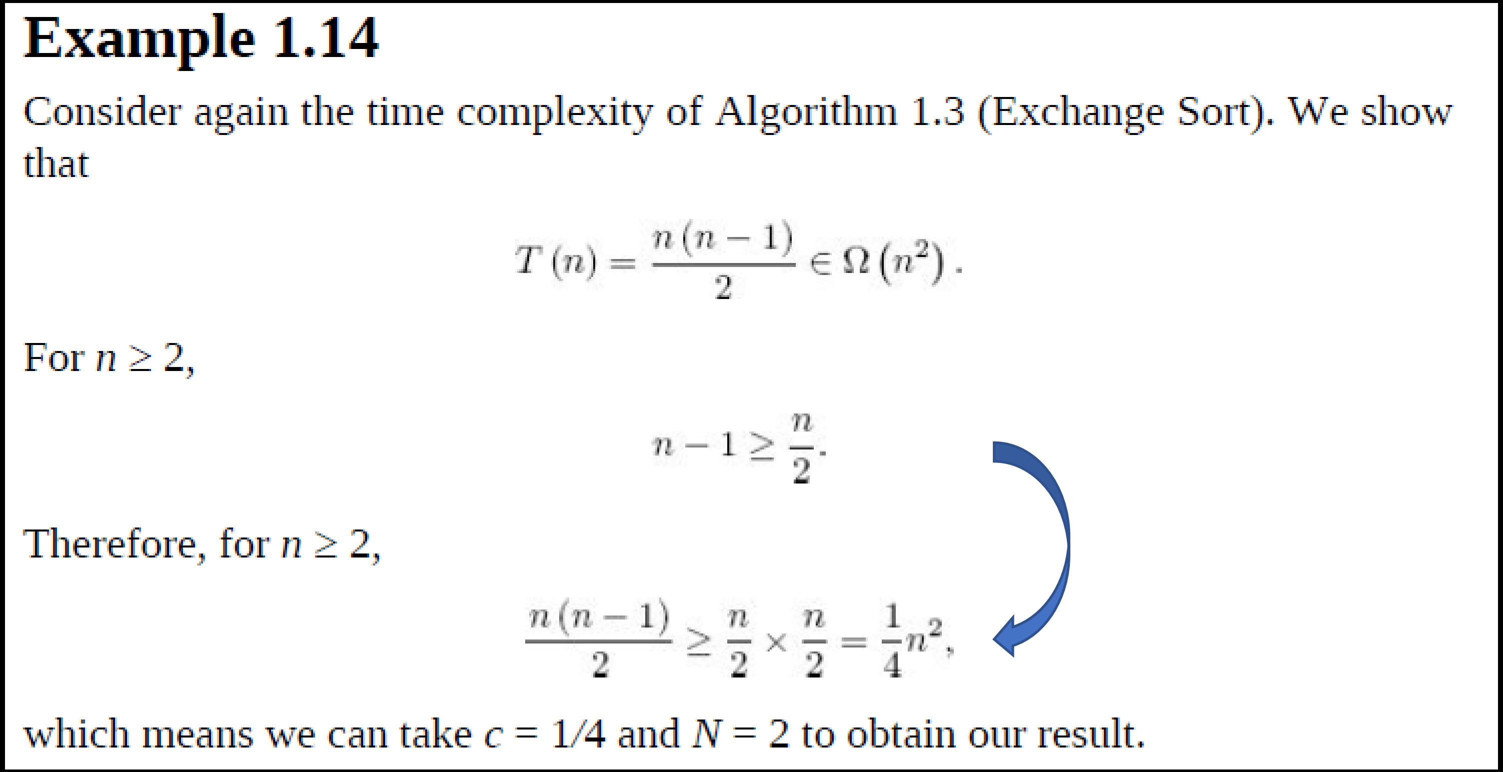
$$n^2 + 10n \geq n^2,$$

we can take $c = 1$ and $N = 0$ to obtain our result.



(b) $g(n) \in \Omega(f(n))$

Ω (Best Case)



Example 1.14

Consider again the time complexity of Algorithm 1.3 (Exchange Sort). We show that

$$T(n) = \frac{n(n-1)}{2} \in \Omega(n^2).$$

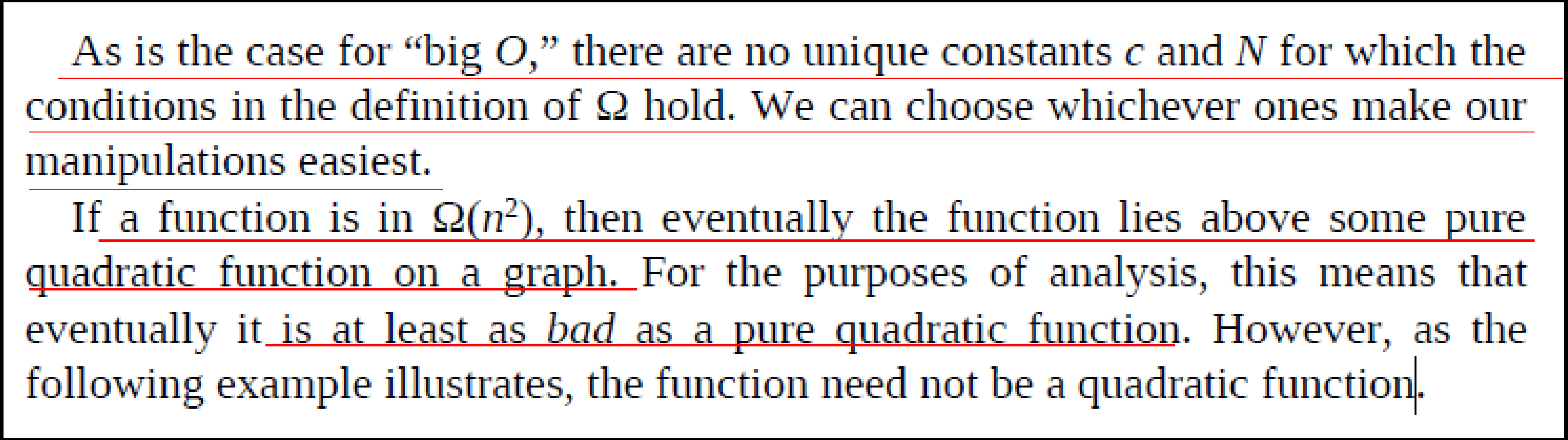
For $n \geq 2$,

$$n - 1 \geq \frac{n}{2}.$$

Therefore, for $n \geq 2$,

$$\frac{n(n-1)}{2} \geq \frac{n}{2} \times \frac{n}{2} = \frac{1}{4}n^2,$$

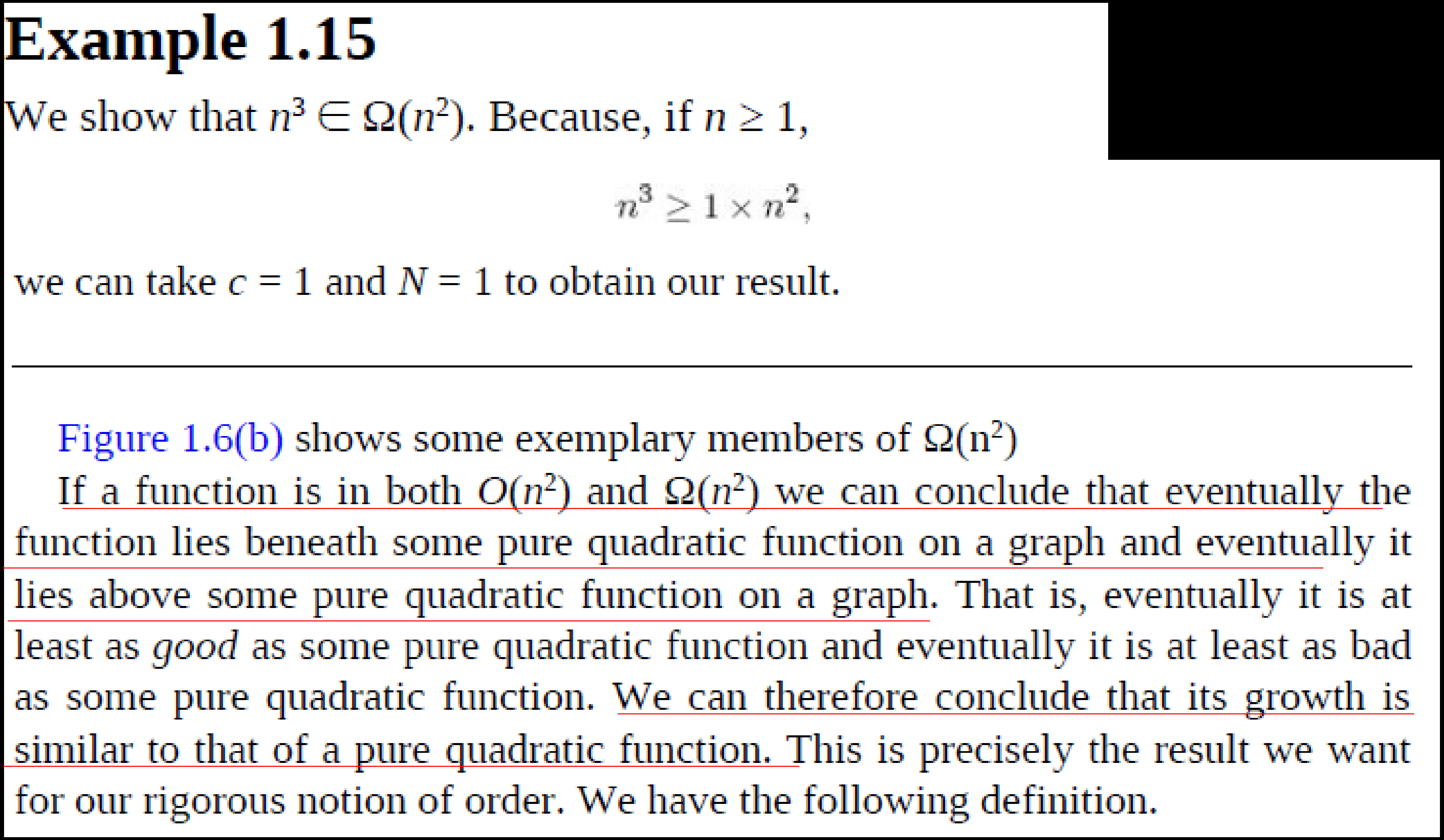
which means we can take $c = 1/4$ and $N = 2$ to obtain our result.



As is the case for "big O," there are no unique constants c and N for which the conditions in the definition of Ω hold. We can choose whichever ones make our manipulations easiest.

If a function is in $\Omega(n^2)$, then eventually the function lies above some pure quadratic function on a graph. For the purposes of analysis, this means that eventually it is at least as *bad* as a pure quadratic function. However, as the following example illustrates, the function need not be a quadratic function.

Ω (Best Case)



Example 1.15

We show that $n^3 \in \Omega(n^2)$. Because, if $n \geq 1$,

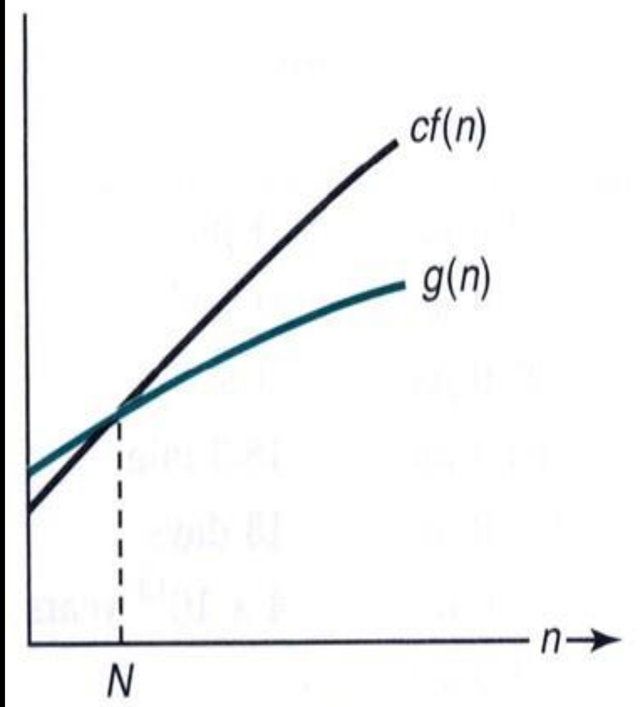
$$n^3 \geq 1 \times n^2,$$

we can take $c = 1$ and $N = 1$ to obtain our result.

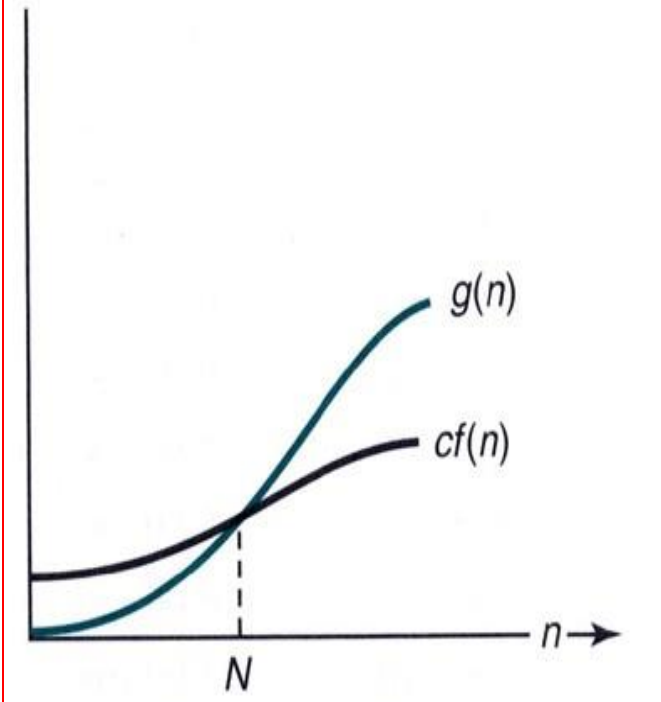
Figure 1.6(b) shows some exemplary members of $\Omega(n^2)$

If a function is in both $O(n^2)$ and $\Omega(n^2)$ we can conclude that eventually the function lies beneath some pure quadratic function on a graph and eventually it lies above some pure quadratic function on a graph. That is, eventually it is at least as *good* as some pure quadratic function and eventually it is at least as bad as some pure quadratic function. We can therefore conclude that its growth is similar to that of a pure quadratic function. This is precisely the result we want for our rigorous notion of order. We have the following definition.

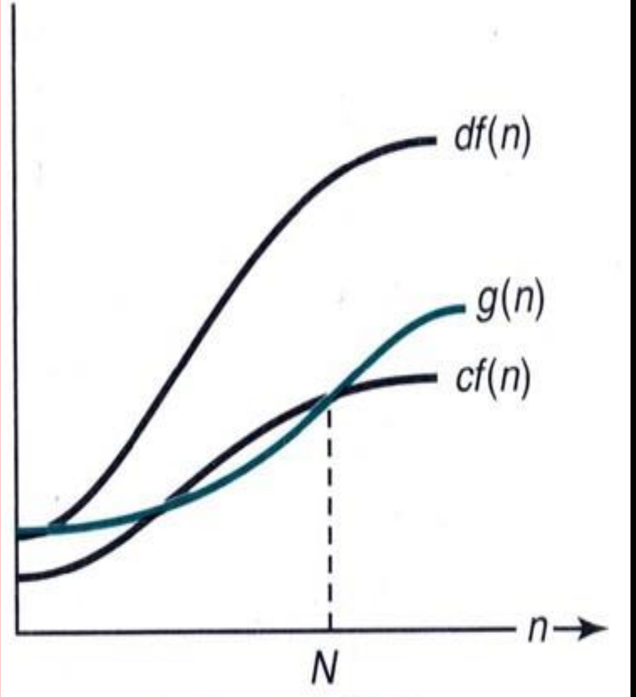
Illustrating "**big O** (Worst Case)", **Ω** (Best Case), and **Θ** (Average Case)



(a) $g(n) \in O(f(n))$



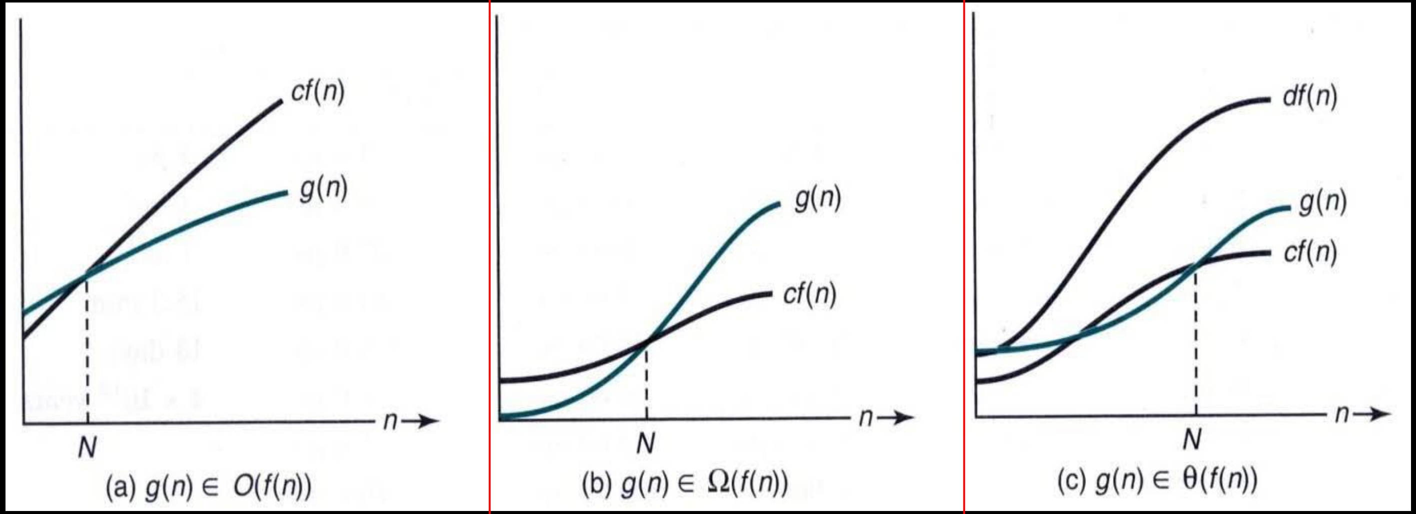
(b) $g(n) \in \Omega(f(n))$



(c) $g(n) \in \theta(f(n))$

Ω (Best Case) Notation: Definition

- Meaning of $g(n) \in \Omega(f(n))$
  - Although *g(n)* starts out below *cf(n)* in the figure, eventually it goes above *cf(n)* and stays there.
  - If *g(n)* is the time complexity for some algorithm, eventually the running time of the algorithm will be **at least as bad** as *f(n)*
  - *f(n)* is called as an asymptotic lower bound (*of what?*)
    - (i.e. *g(n)* cannot run faster than *f(n)*, eventually)



Ω (Best Case) Notation: Example

- Meaning of $n^2+10n \in \Omega(n^2)$
  - Take *c* = *1* and *N* = *0*.
  - For all integer $n \geq 0$, it holds that $n^2+10n \geq n^2$
  - Therefore, n^2 is an asymptotic lower bound for the time complexity function of n^2+10n.
    - (i.e., n^2+10n belongs to $\Omega(n^2)$)
- $5n^2 \in \Omega(n^2)$
  - Take *c* = *1* and *N* = *0*.
  - For all integer $n \geq 0$, it holds that $5n^2 \geq 1\times n^2$
  - Therefore, n^2 is an asymptotic lower bound for the time complexity function of $5n^2$.

Ω (Best Case) Notation: Examples

- $T(n) = \frac{n(n-1)}{2}$
  - Because, **for $n \geq 2$,** **$n - 1 \geq n/2$**, so it holds that $\frac{n(n-1)}{2} \geq \frac{n}{2} \times \frac{n}{2} = \frac{1}{4} n^2$
  - Therefore, we can take *c = 1/4* and *N = 2*, to conclude that $T(n) \in \Omega(n^2)$.
- $n^3 \in \Omega(n^2)$
  - Because, **for $n \geq 1$, it holds that $n^3 \geq 1 \times n^2$**
  - Therefore, we can take *c = 1* and *N = 1*, to conclude that $n^3 \in \Omega(n^2)$

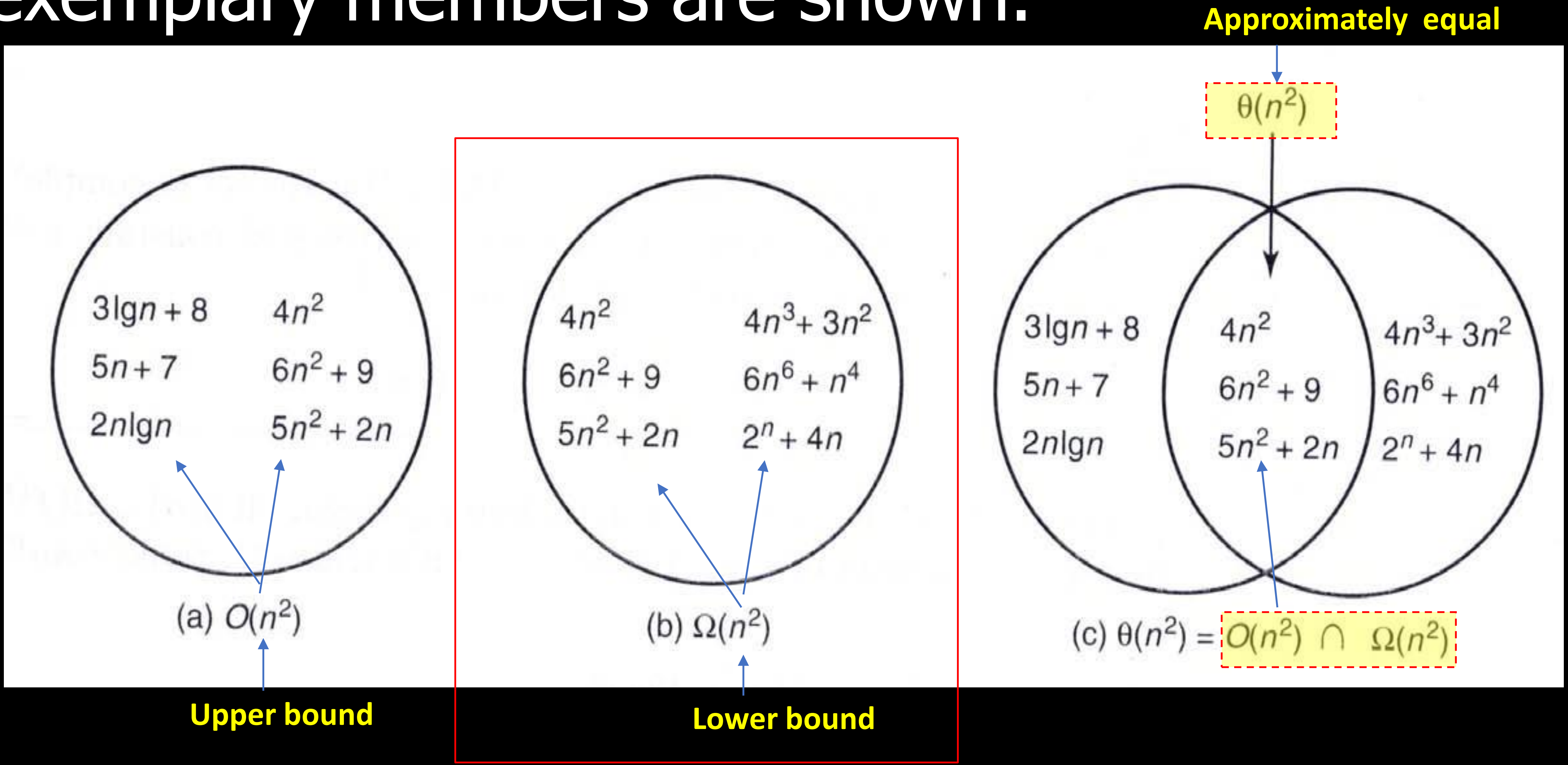
Lower bound

Ω (Best Case) Notation: Example

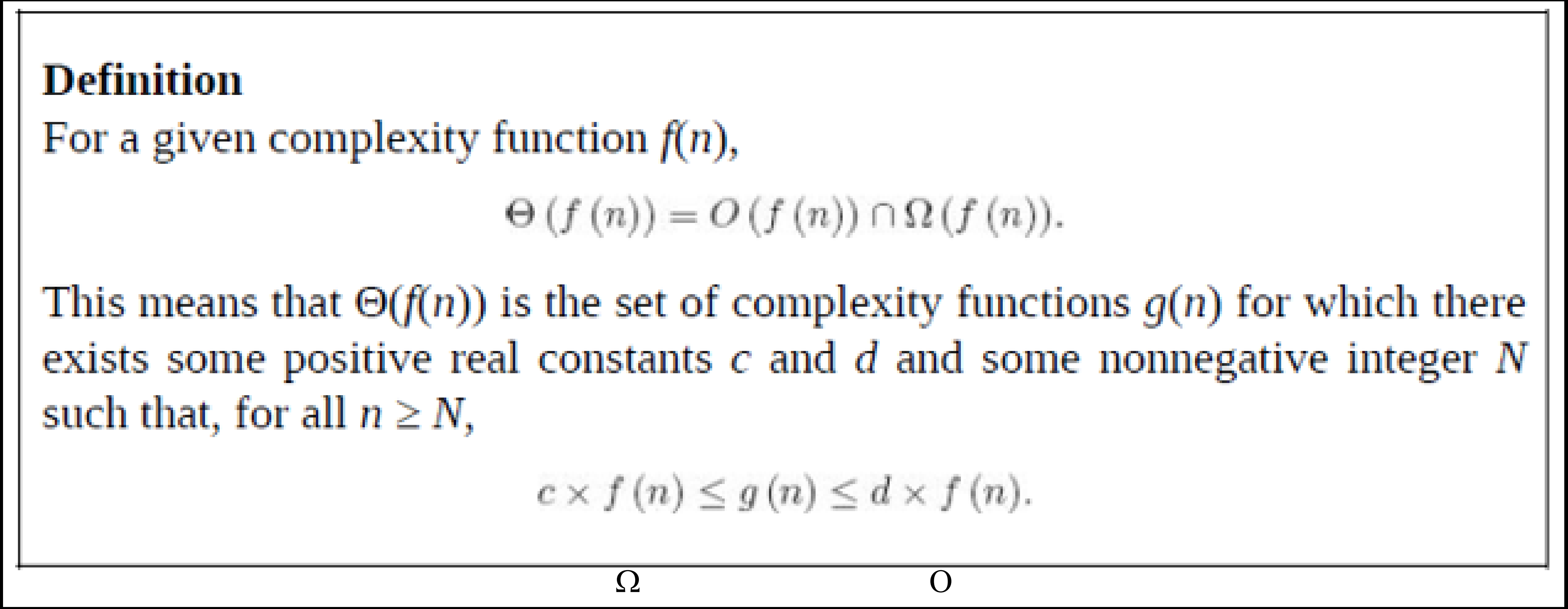
- $n \in \Omega(n^2)$??
  - Proof by contradiction.
  - Suppose it is true that $n \in \Omega(n^2)$.
  - Then, for all integer $n \geq N$, there must exist some positive real number $c > 0$, and **non-negative integer** N.
  - Let us divide both sides by cn.
  - Then, we will get $1/c \geq n$, which is **impossible**.
  - Therefore, n does not belong to $\Omega(n^2)$.

$$n \notin \Omega(n^2)$$

Figure 1.6 The sets $O(n^2)$, $\Omega(n^2)$, $\Theta(n^2)$. Some exemplary members are shown.



Average Case Theeta



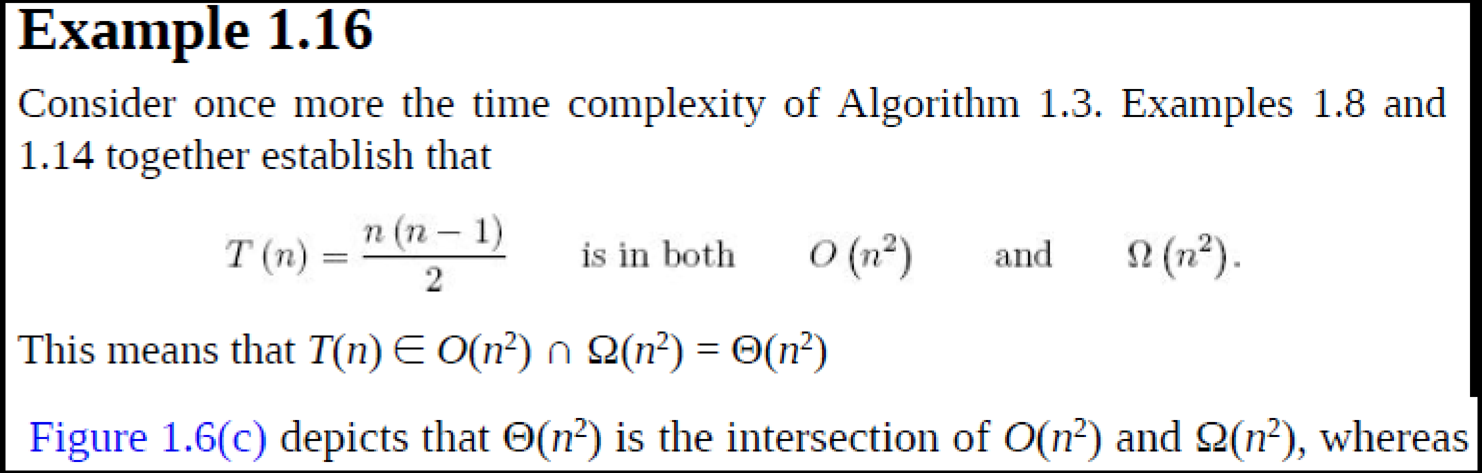
Definition
For a given complexity function $f(n)$,

$$\Theta(f(n)) = O(f(n)) \cap \Omega(f(n)).$$

This means that $\Theta(f(n))$ is the set of complexity functions $g(n)$ for which there exists some positive real constants c and d and some nonnegative integer N such that, for all $n \geq N$,

$$c \times f(n) \leq g(n) \leq d \times f(n).$$

Ω $\quad$ O



Example 1.16

Consider once more the time complexity of Algorithm 1.3. Examples 1.8 and 1.14 together establish that

$$T(n) = \frac{n(n-1)}{2} \quad \text{is in both} \quad O(n^2) \quad \text{and} \quad \Omega(n^2).$$

This means that $T(n) \in O(n^2) \cap \Omega(n^2) = \Theta(n^2)$

Figure 1.6(c) depicts that $\Theta(n^2)$ is the intersection of $O(n^2)$ and $\Omega(n^2)$, whereas

Rigorous Definition to Order: Θ

- Definition: (Asymptotic Tight Bound)
  - For a given complexity function $f(n)$, $\Theta(f(n))$ is the set of complexity functions $g(n)$ for which there exists **some positive real constants c and d** and **some non-negative integer N** such that for all $n \geq N$,

$$c \times f(n) \leq g(n) \leq d \times f(n)$$

- $g(n) \in \Theta(f(n))$, we say that $g(n)$ is order of $f(n)$.
- Example: $T(n) = \frac{n(n-1)}{2}$ $\quad T(n) \in \Theta\left(n^2\right)$

Illustrating “big O”, Ω, and Θ

$$c \times f(n) \le g(n) \le d \times f(n)$$

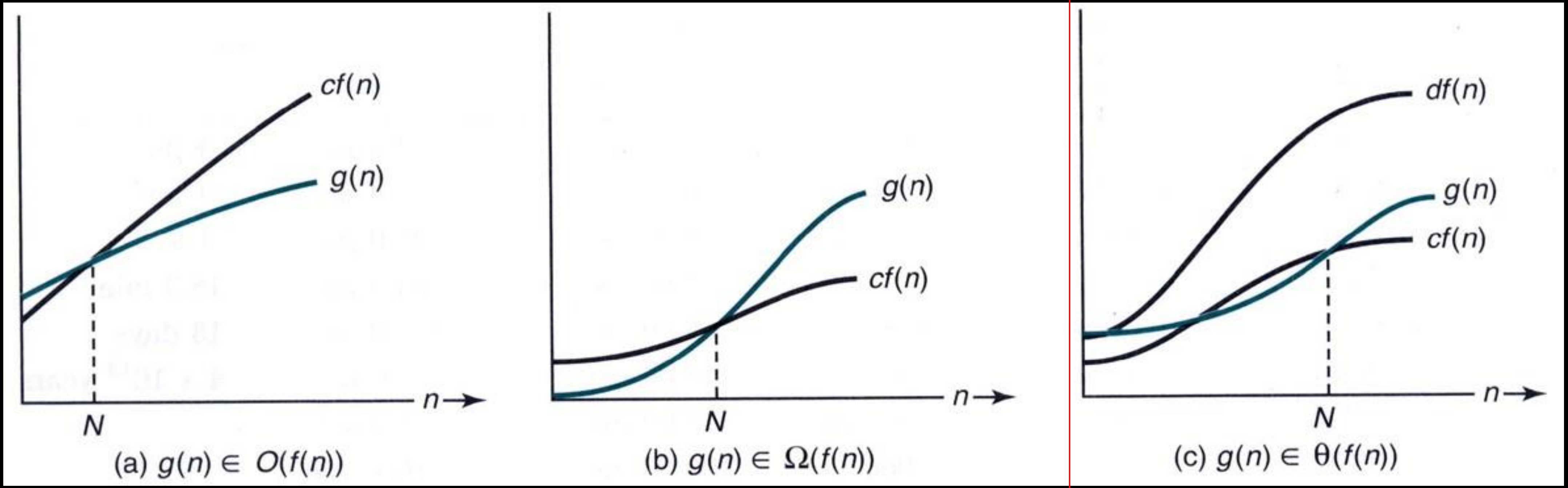
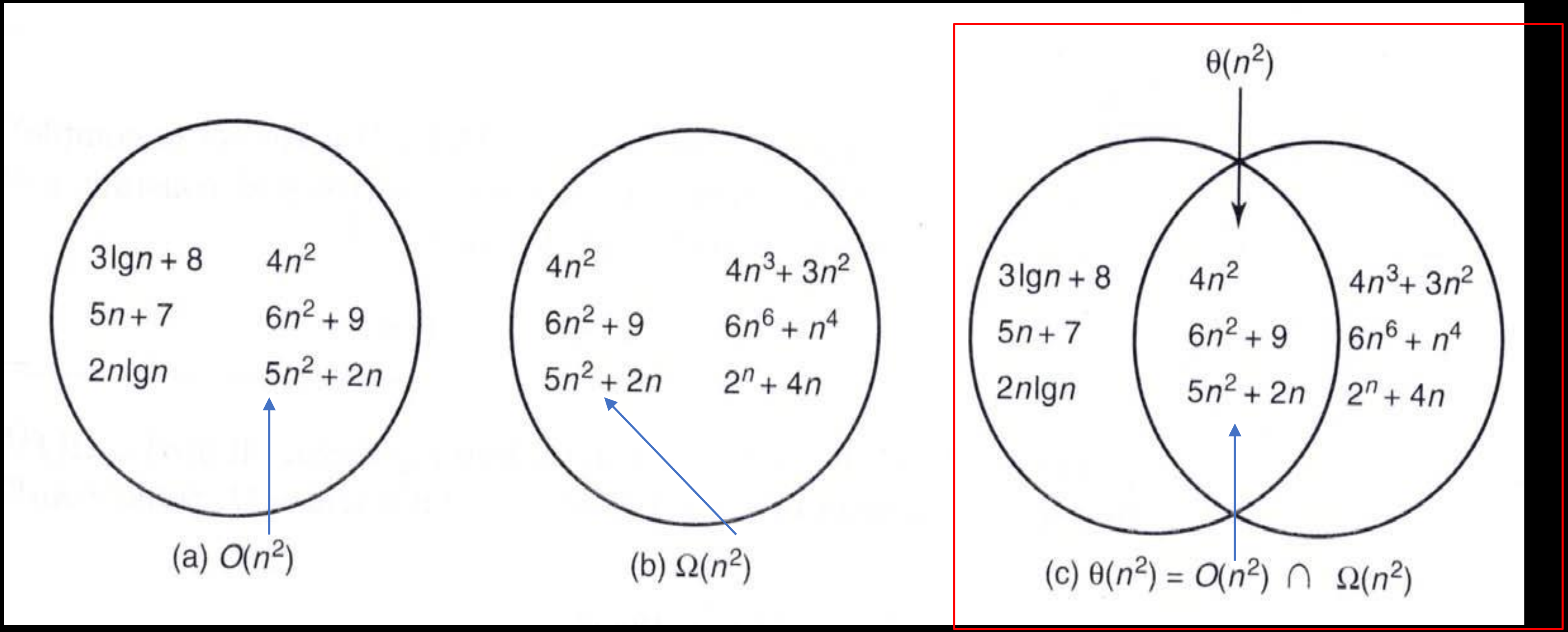


Figure 1.6 The sets $O(n^2)$, $\Omega(n^2)$, $\Theta(n^2)$. Some exemplary members are shown.

Approximately equal



Upper bound   Lower bound

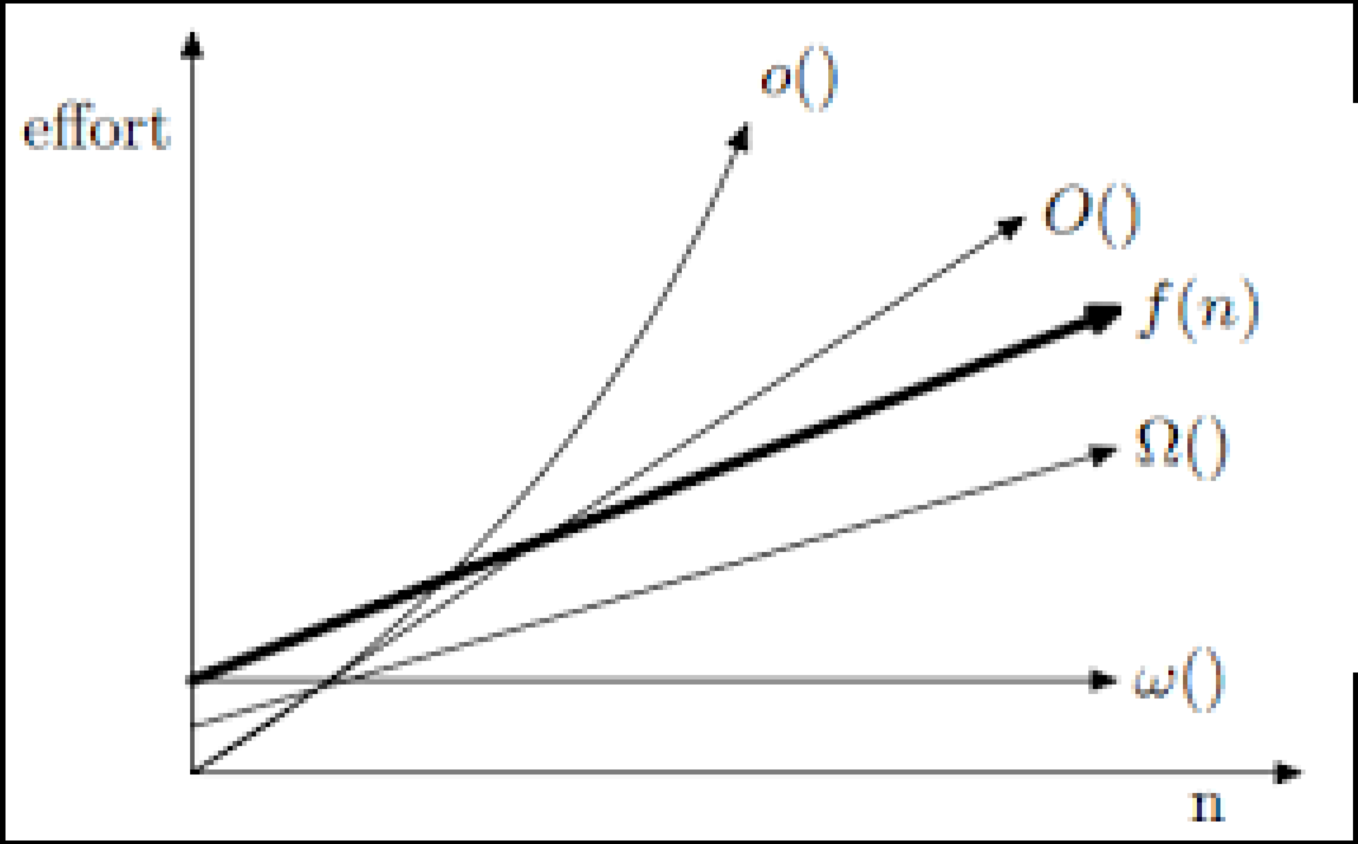
Rigorous Definition to Order: Small *o*

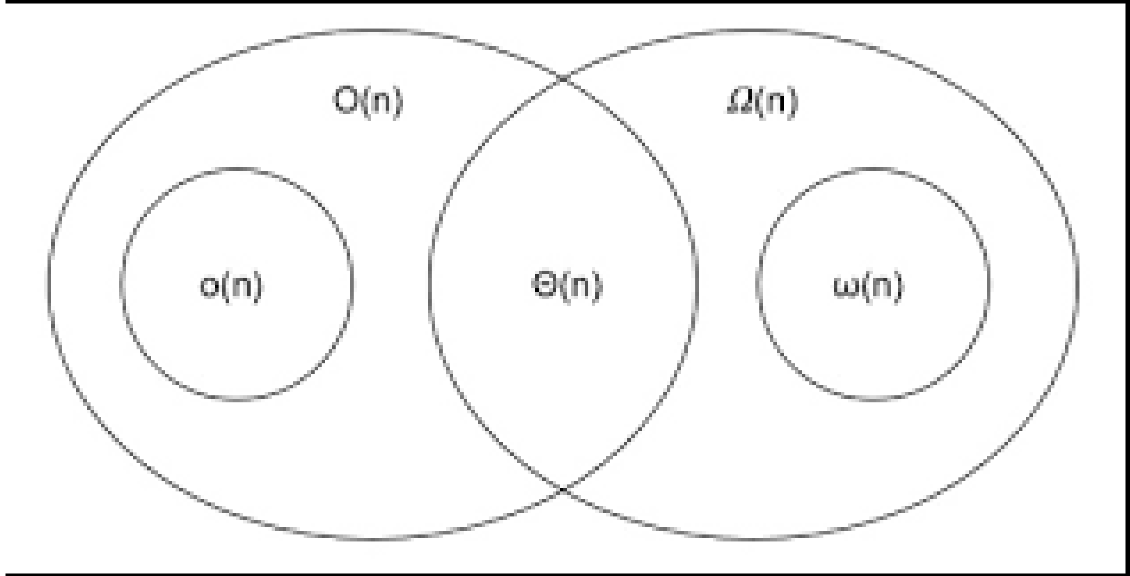
- Definition:
  - For a given complexity function $f(n)$, $o(f(n))$ is the set of complexity functions $g(n)$ satisfying the following: For ***every*** positive real constant c there **exists** a non-negative integer N such that for all $n \geq N$,

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

  - $g(n) \in o(f(n))$

Big O vs Small o and Big Omega vs small omega





Big *O* vs. Small *o*

- Difference
  - Big *O*: For a given complexity function $f(n)$, $O(f(n))$ is the set of complexity functions $g(n)$ for which there exists *some* positive real constant c and some non-negative integer N such that for all $n \geq N$
  - Small *o*: For a given complexity function $f(n)$, $o(f(n))$ is the set of complexity functions $g(n)$ satisfying the following: For *every* positive real constant c there exists a non-negative integer N such that for all $n \geq N$,

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

  - If $g(n) \in o(f(n))$, $g(n)$ is eventually **much better** than $f(n)$.

  - Big O and small o both asymptotic notations that specify upper-bounds for functions and running times of algorithms. **However, the difference is that big-O may be asymptotically tight while little-o makes sure that the upper bound is not asymptotically tight

Small *o* Notation: Example

- $n \in o(n^2)$
  - Suppose $c > 0$. We need to find an N such that, for $n \geq N$, $\boldsymbol{n \leq cn^2}$.
  - If we divide both sides by cn,
  - Then, we get $\boldsymbol{1/c \leq n}$
  - Therefore, it sufficient to choose any $N \geq 1/c$.
  - For example, **if $\boldsymbol{c=0.00001}$**, we must take N equal to at least 100,000.
  - That is, for $N \geq 100{,}000$, $\boldsymbol{n} \leq 0.00001\boldsymbol{n^2}$.

Small *o* Notation: Example2

- $n \in o(5n)$?
  - Proof by contradiction.
  - Let c = 1/6. If $n \in o(5n)$, then there must exist some N such that, for $n \geq N$,
    $$\boldsymbol{n \leq \frac{1}{6} \times 5n = \frac{5}{6} n}$$
  - But it is impossible.
  - This contradiction proves that n is not in $o(5n)$.

Summary

1. best-case scenario of a time complexity can be described as Omega Big-Ω notation
2. average-case scenario of a time complexity can be described as Theta Big-Θ notation
3. worst-case scenario of time complexity and the most important that is the Big-O notation
4. O(1): **constant** time. Examples of accessing a number from an array. Calculating numbers…
5. O(log n): **Logarithmic** time. Uses the divide and conquer strategy.
   - The binary search and tree binary search are good examples of algorithms that have this time complexity.
   - An approximate real-world example would be to search a word in the dictionary.

https://javachallengers.com/big-o-notation-explanation/

Summary ctd..

6. O(n) – **Linear time**: When we traverse the whole array once, we have the O(n) time complexity.
    - When we store information in an array of n we will also have the O(n) for space complexity.
    - A real-world example could be reading a book.
7. O(N log N) – **Log-linear**/ linearithmic : The Merge sort, Quick Sort, Tim Sort, and Heap Sort are algorithms that have log-linear complexity.
    - Those algorithms use the divide and conquer strategy making it more effective than O(n^2).

https://javachallengers.com/big-o-notation-explanation/

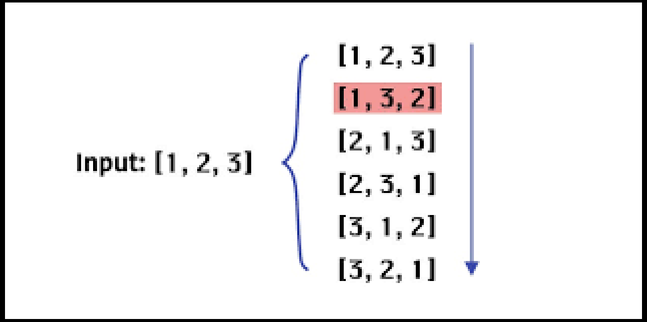
Summary ctd..

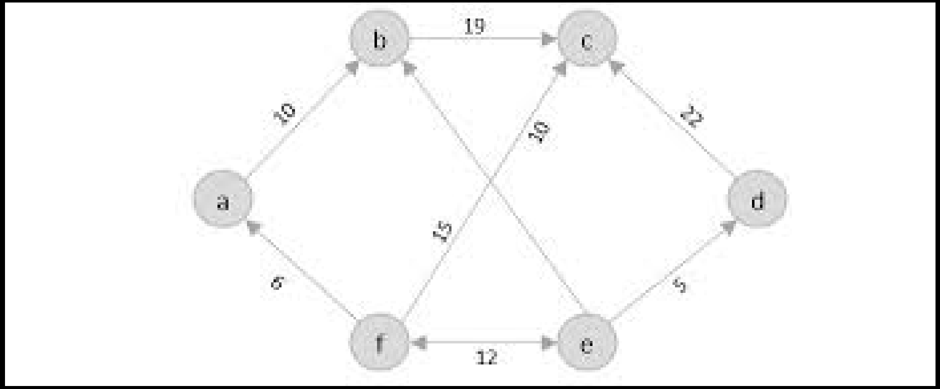
8. $O(N^2)$ – **Quadratic**: The Bubble Sort, Insertion Sort, and Select Sort are some of the algorithms that have the quadratic complexity.
   - If there are two nested loops traversing the whole array each time, then we have $O(n^2)$ of time complexity.
9. $O(N^3)$ – **Cubic**: When we have 3 nested loops and we traverse the whole array on those loops we will have the cubic time complexity.
10. $O(c^n)$ – **Exponential**: The exponential complexity grows very quickly.
    - A good example of this time complexity is when someone try to break a password.
    - Suppose it is a password that supports only numbers(0-9) and has 4 digits. That equivalates to $10^4 = 10000$ possible combinations.
    - The greater the exponent number the faster the number grows.

https://javachallengers.com/big-o-notation-explanation/

Summary ctd..

11. O(n!) – **Factorial**: The factorial of 3! is the same as 1 * 2 * 3 = 6. It grows in a similar way to exponential complexity.
    - The algorithms that have this time complexity are array permutation and traveling salesman.





Refer https://www.bigocheatsheet.com/
https://javachallengers.com/big-o-notation-explanation/

Questions?