# Problem Solving Strategies and Computational Approaches SCS1304

## Handout 3 : Asymptotic notation

Prof Prasad Wimalaratne PhD(Salford),SMIEEE

# Overview

- Analysis of Algorithms
    - Order of Algorithms
    - Practical Considerations
- Every-case time complexity
- Worst-case time complexity
- Best-case time complexity
- Average-case time complexity

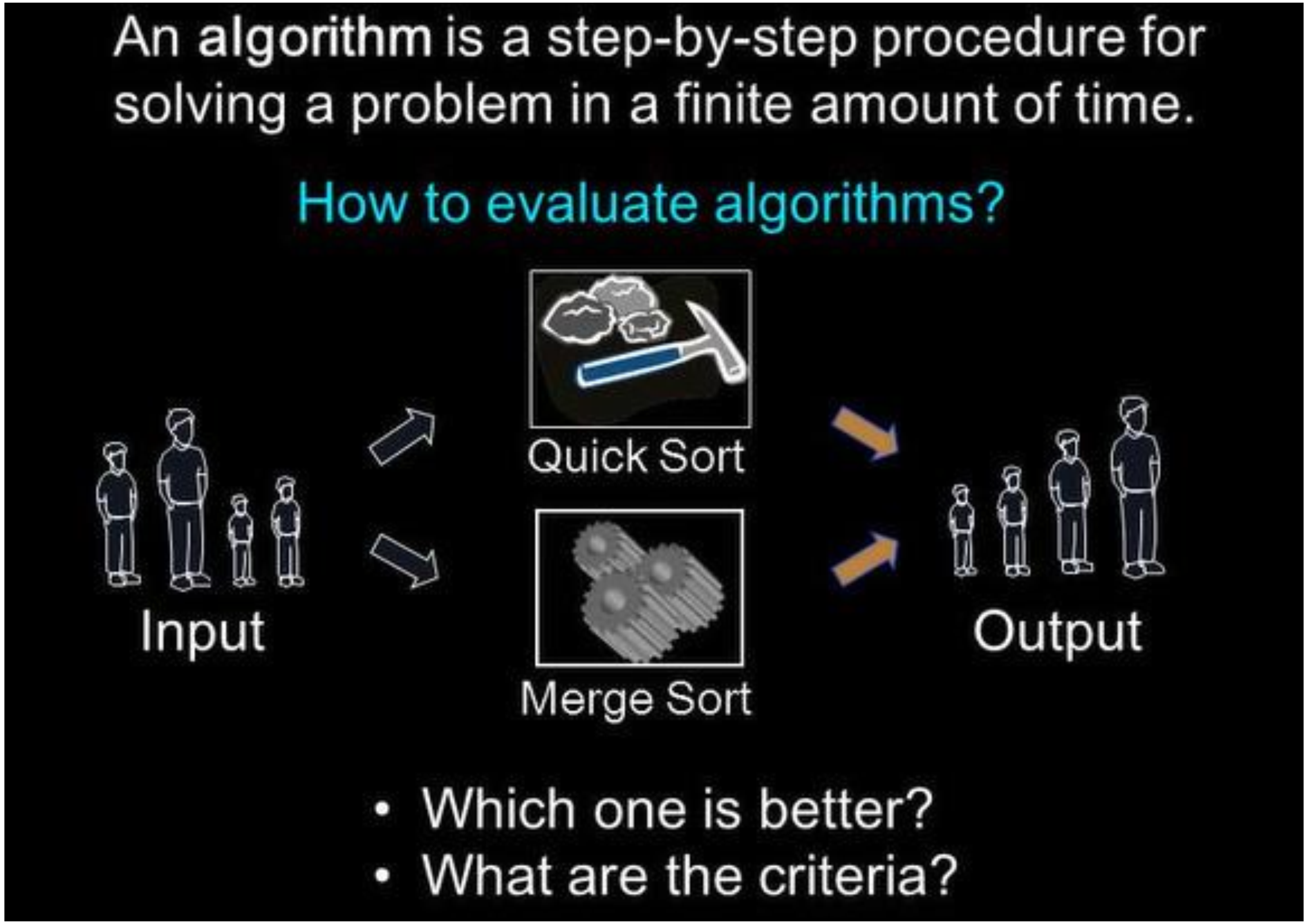
# [ Chapter 1 ] (part2)

# Algorithms :

# Efficiency, Analysis,

# And Order

# Analysis of Algorithms



An **algorithm** is a step-by-step procedure for solving a problem in a finite amount of time.

How to evaluate algorithms?

- Which one is better?
- What are the criteria?

# Analysis of Algorithms

- Analysis involves evaluating algorithms to understand their efficiency, performance, and behavior:

1. Correctness: Ensuring that an algorithm produces the correct output for all possible valid inputs.
2. Complexity Analysis: Determining the time and space complexity of an algorithm. This helps in predicting and comparing the efficiency of different algorithms.
3. Empirical Analysis: Testing algorithms on real data sets to measure their actual performance in practice. This complements theoretical complexity analysis.

# Order of Algorithms

- “Order” categorizes algorithms based on their efficiency and complexity:
  1. Comparison of Sorting: Algorithms like Merge Sort, Quick Sort, and Heap Sort are categorized by their time complexity for sorting elements in arrays or lists.
  2. Searching Algorithms: Techniques like Binary Search and Linear Search are evaluated based on their time complexity for finding elements in data structures.
  3. Dynamic Programming: Algorithms that break down complex problems into smaller subproblems and use memoization or tabulation to optimize solutions.

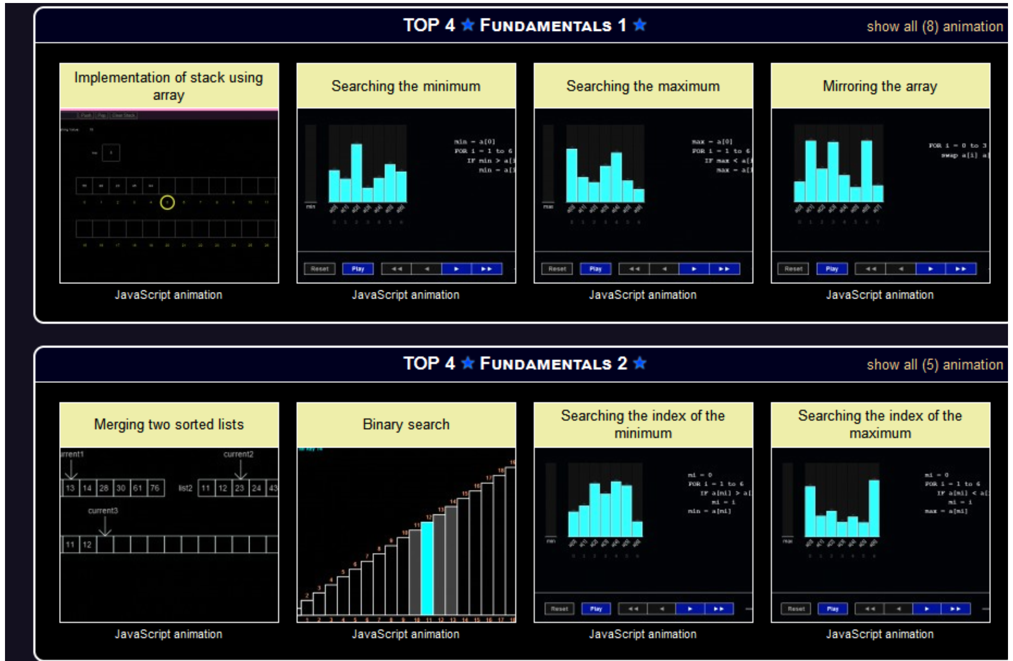
# Practical Considerations

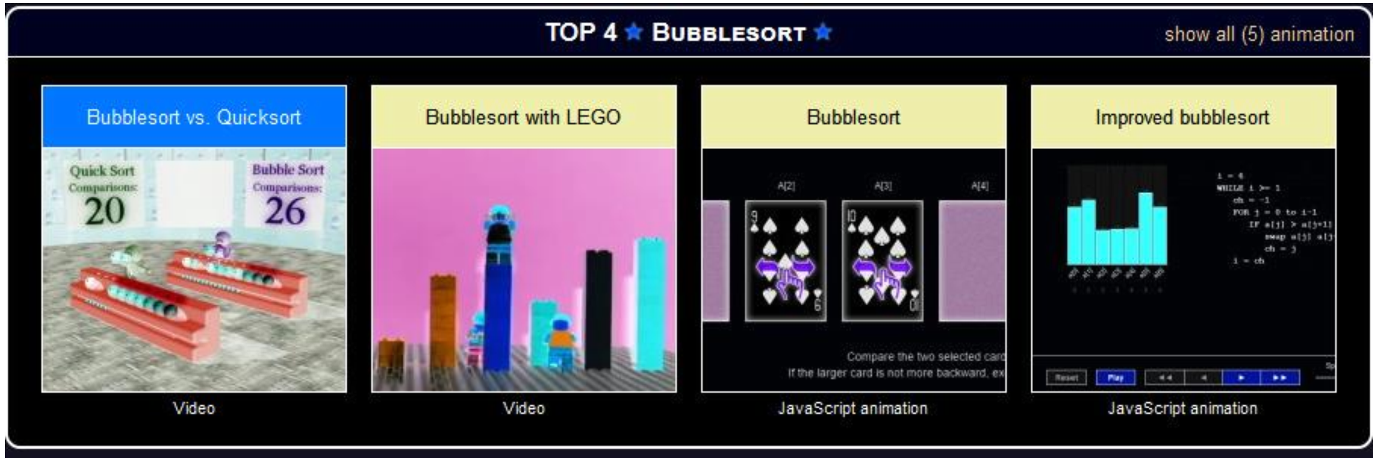
1. Algorithm Selection: Choosing the right algorithm based on problem requirements, input size, and desired performance characteristics.
2. Optimization Techniques: Strategies such as pruning, memoization, and parallelism to improve algorithm efficiency and reduce time or space complexity.
3. Algorithm Design Paradigms: Understanding different approaches like divide and conquer, greedy algorithms, and dynamic programming for solving diverse computational problems efficiently.

(to be be covered in upcoming lessons)

# Algorithm Animations and Visualizations

- [collection of computer science algorithm animations and visualizations]()
- Refer the following URL  for visualizing commonly used algorithms
- https://algoanim.ide.sk/





# Analysis of Algorithms

- In general, the running time of an algorithm increases with the size of the input, and the total running time is roughly proportional to how many times some basic operation (such as a comparison instruction) is carried out.
- We therefore analyze the algorithm's efficiency by determining the number of times some basic operation is carried out as a function of the size of the input.
- After determining the input size, we pick some instruction or group of instructions such that the total work done by the algorithm is roughly proportional to the number of times this instruction or group of instructions is done.
- We call this instruction or group of instructions the **basic operation** in the algorithm.
    - E.g how many times x is compared with elements of the array in earlier examples ?

# Analysis of Algorithms ctd..

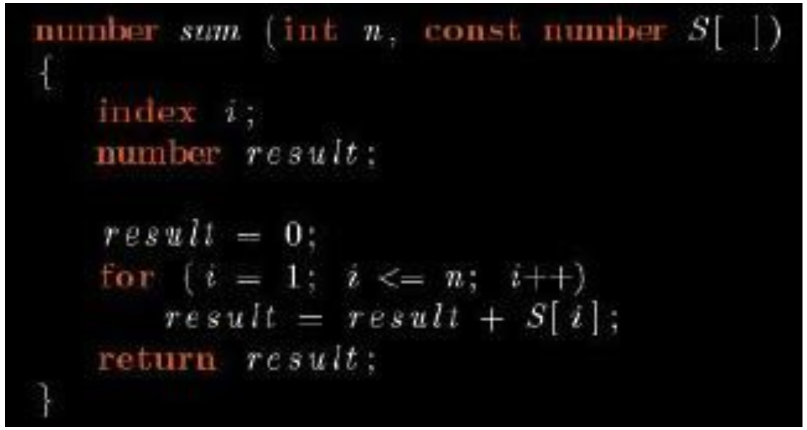
- In general, a time complexity analysis of an algorithm is the determination of how many times the basic operation is done for each value of the input size. Although we do not want to consider the details of how an algorithm is implemented, we will ordinarily assume that the basic operation is implemented as efficiently as possible.

# Analysis of Algorithms

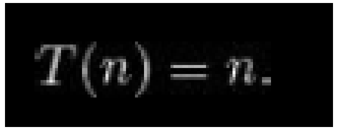
- T(n) is called the every-case time complexity of the algorithm, and the determination of T(n) is called an every-case time complexity analysis.
- Examples

# Every-Case Time Complexity (Add Array Members)

- Other than control instructions, the only instruction in the loop is the one that adds an item in the array to sum. Therefore, we will call that instruction the **basic operation.**
- Basic operation: the addition of an item in the array to sum.
- Input size: n, the number of items in the array.
- Regardless of the values of the numbers in the array, there are n passes through the for loop.
- Therefore, the basic operation is always done n times and =>
- in the case of an algorithm that sorts by comparing keys, we can consider the comparison instruction or the assignment instruction as the basic operation.



```
number sum (int n, const number S[ ])
{
   index i;
   number result;

   result = 0;
   for (i = 1; i <= n; i++)
      result = result + S[i];
   return result;
}
```

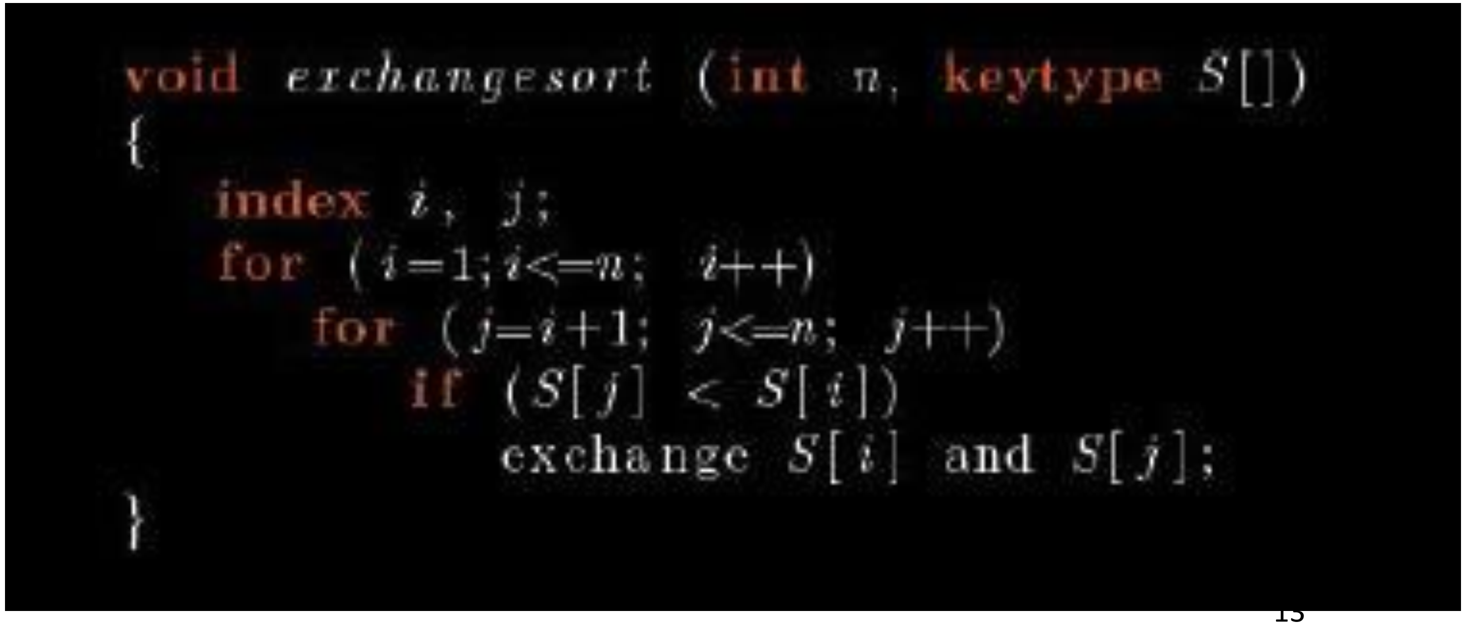


$T(n) = n.$

# Every-Case Time Complexity (Exchange Sort)

- We will analyze the number of comparisons here.
- Basic operation: the comparison of S[j] with S[i].
- Input size: n, the number of items to be sorted.
- We must determine how many passes there are through the for-j loop.

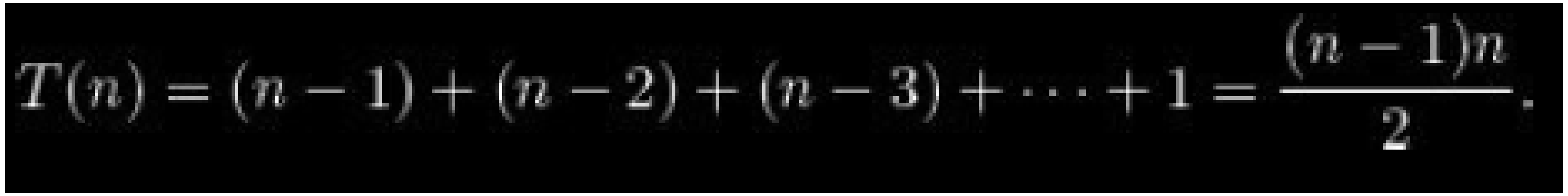
Ex. Trace the exchange sort algorithm for an array of 5 elements.



```
void exchangesort (int n, keytype S[])
{
    index i, j;
    for (i=1; i<=n; i++)
        for (j=i+1; j<=n; j++)
            if (S[j] < S[i])
                exchange S[i] and S[j];
}
```

# Every-Case Time Complexity (Exchange Sort) ctd..

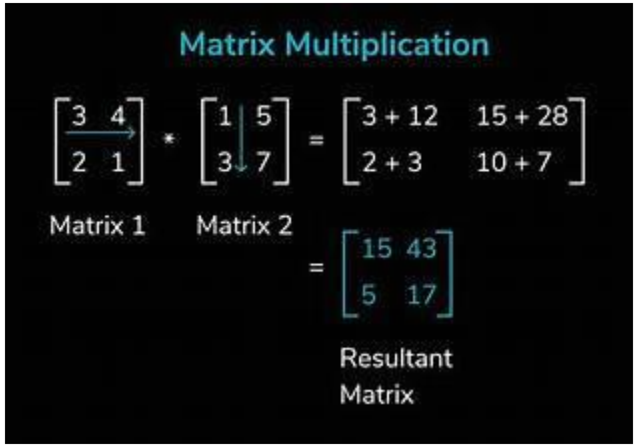
- For a given n there are always n − 1 passes through the for-i loop.
- In the first pass through the for-i loop, there are n − 1 passes through the for-j loop, in the second pass there are n − 2 passes through the for-j loop, in the third pass there are n−3 passes through the for-j loop, … , and in the last pass there is one pass through the for-j loop.
- Therefore, the total number of passes through the for-j loop is given by



$$T(n) = (n-1) + (n-2) + (n-3) + \cdots + 1 = \frac{(n-1)n}{2}.$$

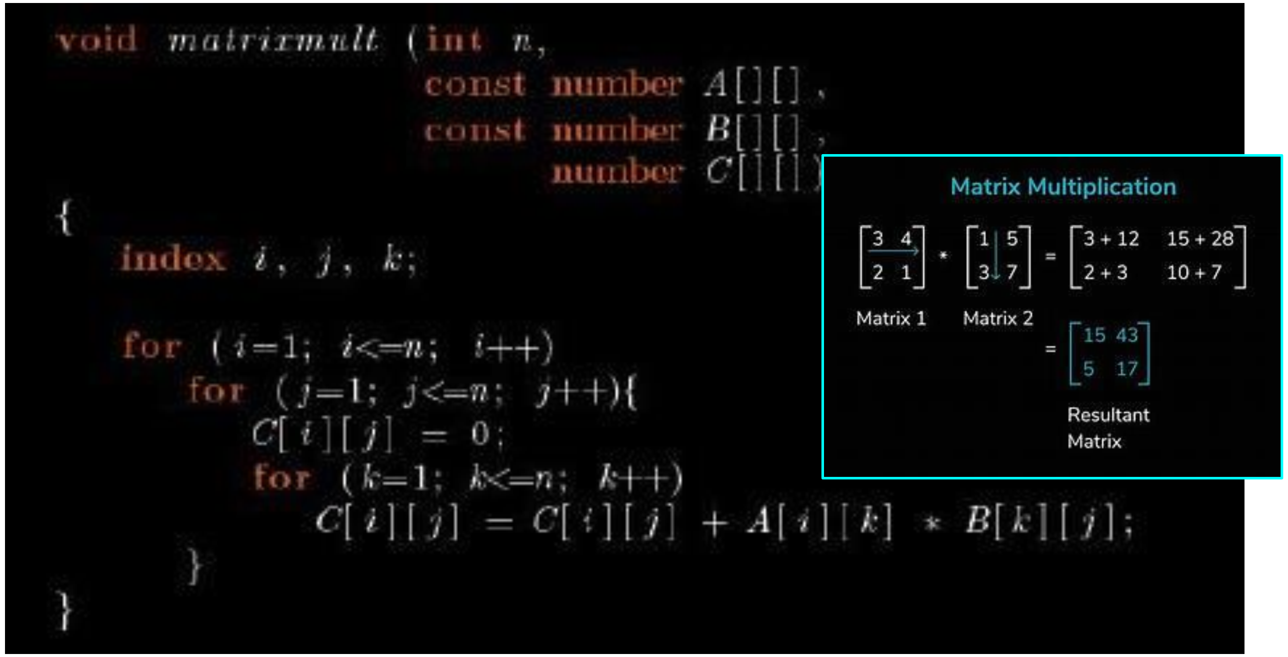
# Every-Case Time Complexity (Matrix Multiplication)

- The only instruction in the innermost for loop is the one that does a multiplication and an addition. The algorithm can be implemented in such a way that fewer additions are done than multiplications.
- Therefore, we will consider only the multiplication instruction to be the basic operation.
- Basic operation: multiplication instruction in the innermost for loop.
- Input size: n, the number of rows and columns.


Matrix Multiplication
$\begin{bmatrix} 3 & 4 \\ 2 & 1 \end{bmatrix} * \begin{bmatrix} 1 & 5 \\ 3 & 7 \end{bmatrix} = \begin{bmatrix} 3+12 & 15+28 \\ 2+3 & 10+7 \end{bmatrix}$
Matrix 1
Matrix 2
$= \begin{bmatrix} 15 & 43 \\ 5 & 17 \end{bmatrix}$
Resultant Matrix

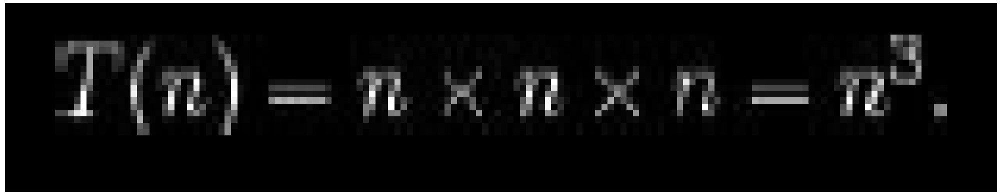
# Every-Case Time Complexity (Matrix Multiplication)

- There are always n passes through the for -i loop, in each pass there are always n passes through the for -j loop, and in each pass through the for -j loop there are always n passes through the for-k loop. Because the basic operation is inside the for-k loop,



```
void matrixmult (int n,
                 const number A[][],
                 const number B[][],
                 number C[][])
{
   index i, j, k;

   for (i=1; i<=n; i++)
      for (j=1; j<=n; j++){
         C[i][j] = 0;
         for (k=1; k<=n; k++)
            C[i][j] = C[i][j] + A[i][k] * B[k][j];
      }
}
```

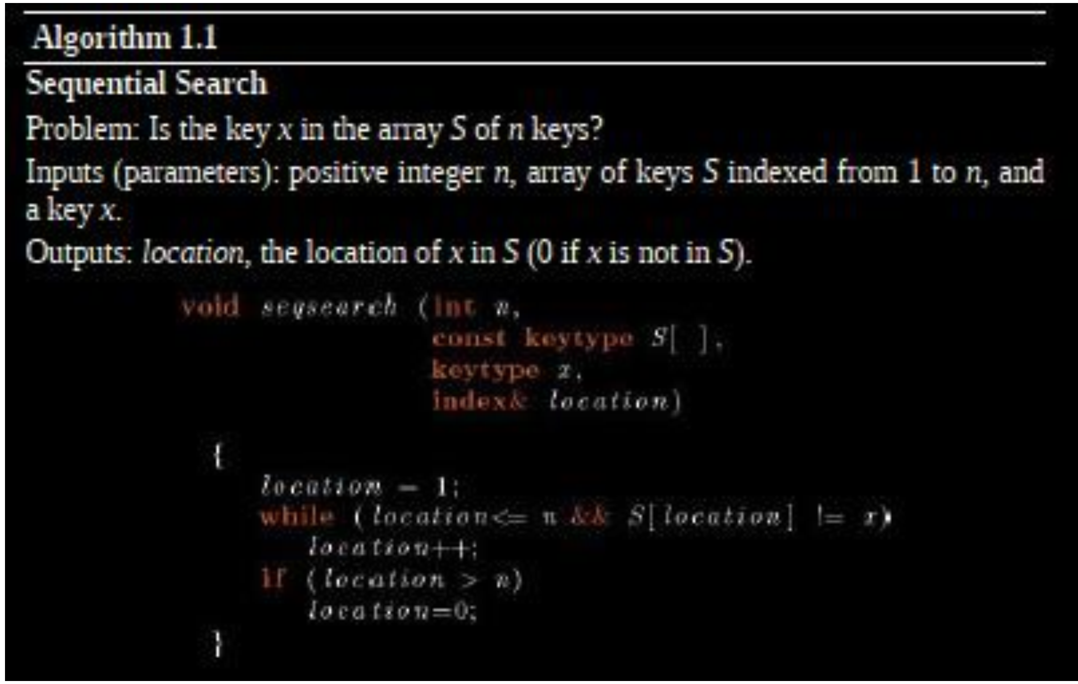
$$\begin{bmatrix} 3 & 4 \\ 2 & 1 \end{bmatrix} * \begin{bmatrix} 1 & 5 \\ 3 & 7 \end{bmatrix} = \begin{bmatrix} 3+12 & 15+28 \\ 2+3 & 10+7 \end{bmatrix} = \begin{bmatrix} 15 & 43 \\ 5 & 17 \end{bmatrix}$$



$$T(n) = n \times n \times n = n^3.$$

# Every-Case Time Complexity

- the basic operation in Sequential Search is not done the same number of times for all instances of size n.
- So this algorithm does not have an every-case time complexity.
- This is true for many algorithms.
- However, this does not mean that we cannot analyze such algorithms, because there are three other analysis techniques that can be tried.



**Algorithm 1.1**

**Sequential Search**

Problem: Is the key *x* in the array *S* of *n* keys?

Inputs (parameters): positive integer *n*, array of keys *S* indexed from 1 to *n*, and a key *x*.

Outputs: *location*, the location of *x* in *S* (0 if *x* is not in *S*).

```
void seqsearch (int n,
                const keytype S[ ],
                keytype x,
                index& location)
{
   location = 1;
   while (location<= n && S[location] != x)
      location++;
   if (location > n)
      location=0;
}
```

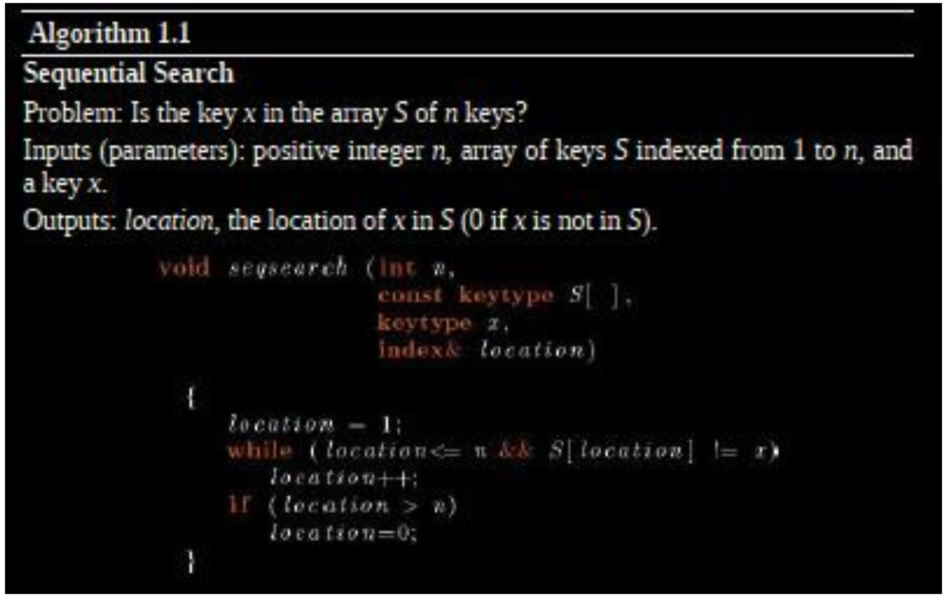
# Worst-case time complexity analysis

- The first is to consider the maximum number of times the basic operation is done.
- For a given algorithm, W(n) is defined as the maximum number of times the algorithm will ever do its basic operation for an input size of n.
- W(n) is called the worst-case time complexity of the algorithm, and the determination of W(n) is called a worst-case time complexity analysis.
- If T(n) exists, then clearly W(n) = T(n). The following is an analysis of W(n) in a case in which T(n) does not exist.

# Worst-Case Time Complexity (Sequential Search)

- Basic operation: the comparison of an item in the array with x.
- Input size: n, the number of items in the array.
- The basic operation is done at most n times, which is the case if x is the last item in the array or if x is not in the array. Therefore,

$$W(n) = n.$$



**Algorithm 1.1**

**Sequential Search**

Problem: Is the key *x* in the array *S* of *n* keys?

Inputs (parameters): positive integer *n*, array of keys *S* indexed from 1 to *n*, and a key *x*.

Outputs: *location*, the location of *x* in *S* (0 if *x* is not in *S*).

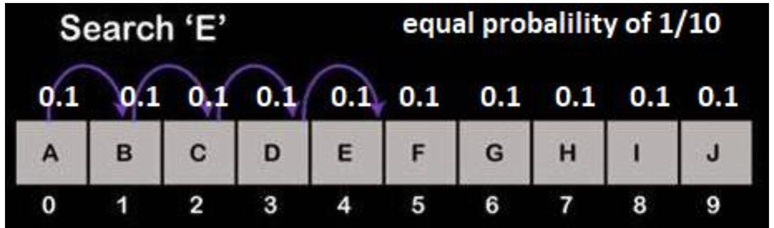
```
void seqsearch (int n,
                const keytype S[ ],
                keytype x,
                index& location)

 {
    location = 1;
    while (location<= n && S[location] != x)
       location++;
    if (location > n)
       location=0;
 }
```

# Worst-Case Time Complexity (Sequential Search)

- Although the worst-case analysis informs us of the absolute maximum amount of time consumed, in some cases we may be more interested in knowing how the algorithm performs on the average.
- For a given algorithm, A(n) is defined as the average (expected value) of the number of times the algorithm does the basic operation for an input size of n.
- A(n) is called the average-case time complexity of the algorithm, and the determination of A(n) is called an average-case time complexity analysis.
- As is the case for W(n), if T(n) exists, then A(n) = T(n).

# Average -Case Time Complexity (Sequential Search)

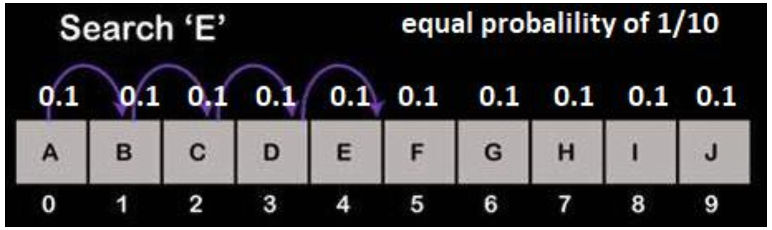
- To compute A(n), we need to assign probabilities to all possible inputs of size n.
- It is important to assign probabilities based on all available information.
- For example, our next analysis will be an average-case analysis of Sequential Search (Algorithm 1.1).
- Assumption: We will assume that if x is in the array, it is equally likely to be in any of the array slots.
- If we know only that x may be somewhere in the array, our information gives us no reason to prefer one array slot over another.



| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 0.1 | 0.1 | 0.1 | 0.1 | 0.1 | 0.1 | 0.1 | 0.1 | 0.1 | 0.1 |
| A | B | C | D | E | F | G | H | I | J |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

# Average -Case Time Complexity (Sequential Search) ctd.

- Therefore, it is reasonable to assign equal probabilities to all array slots.
- This means that we are determining the average search time when we search for all items the same number of times.



|  |  |  |  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|---|---|
| 0.1 | 0.1 | 0.1 | 0.1 | 0.1 | 0.1 | 0.1 | 0.1 | 0.1 | 0.1 |
| A | B | C | D | E | F | G | H | I | J |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

- If we have information indicating that the inputs will not arrive according to this distribution, we should not use this distribution in our analysis.

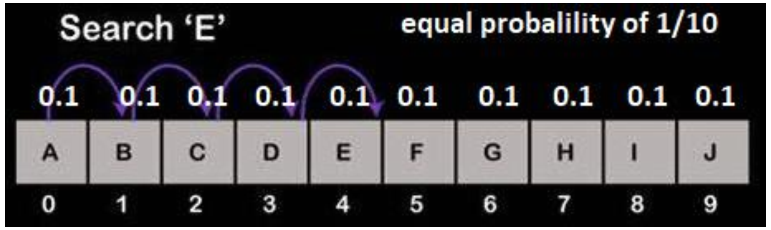
# Average -Case Time Complexity (Sequential Search) ctd..

- For example, if the array contains first names and we are searching for names that have been chosen at random from all people in a country, an array slot containing the common names will probably be searched more often than one containing the uncommon names
- We should not ignore this information and assume that all slots are equally likely.
- it is usually harder to analyze the average case than it is to analyze the worst case.

# Average-Case Time Complexity (Sequential Search)

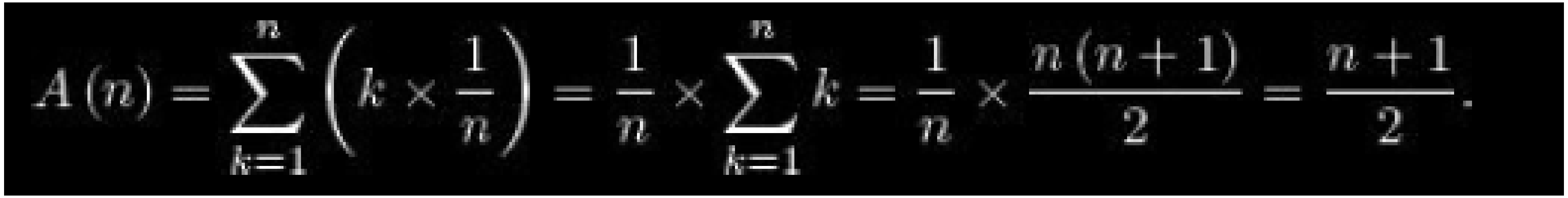
- Basic operation: the comparison of an item in the array with x.
- Input size: n, the number of items in the array.

- We first analyze the case in which it is known that x is in S, where the items in S are all distinct, and where we have no reason to believe that x is more likely to be in one array slot than it is to be in another.
- Based on this information, for $1 \le n$, the probability that x is in the $k^{th}$ array slot is 1/n.



# Average-Case Time Complexity (Sequential Search) ctd..

- If x is in the $k^{th}$ array slot, the number of times the basic operation is done to locate x (and, therefore, to exit the loop) is k.
- This means that the average time complexity is given by



$$A(n) = \sum_{k=1}^{n}\left(k \times \frac{1}{n}\right) = \frac{1}{n} \times \sum_{k=1}^{n} k = \frac{1}{n} \times \frac{n(n+1)}{2} = \frac{n+1}{2}.$$

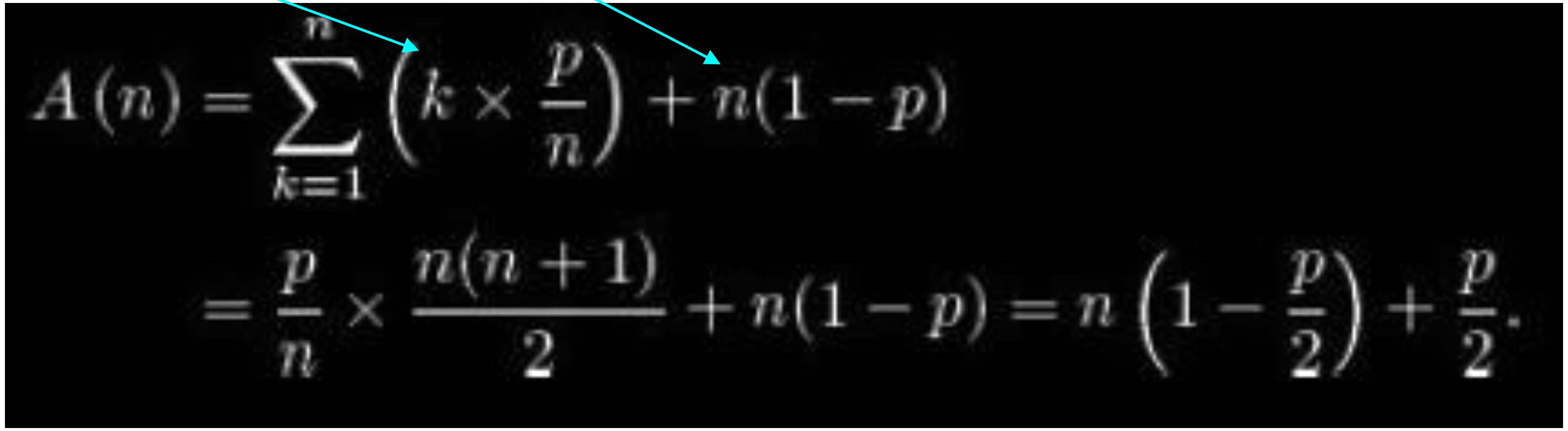
- As we would expect, on the average, **about half the array** is searched.

# Average-Case Time Complexity (Sequential Search) ctd..

- Next we analyze the case in which x may not be in the array.
- To analyze this case we must assign some probability p to the event that x is in the array.
- If x is in the array, we will again assume that it is equally likely to be in any of the slots from 1 to n.
- The probability that x is in the k$^{th}$ slot is then p/n, and the probability that it is not in the array is 1 − p.

# Average-Case Time Complexity (Sequential Search) ctd..

- Recall(Algo 1.2) that there are k passes through the loop if x is found in the kth slot, and n passes through the loop if x is not in the array.
- The average time complexity is therefore given by



$$A(n) = \sum_{k=1}^{n}\left(k \times \frac{p}{n}\right) + n(1-p)$$
$$= \frac{p}{n} \times \frac{n(n+1)}{2} + n(1-p) = n\left(1 - \frac{p}{2}\right) + \frac{p}{2}.$$

- The last step in this triple equality is derived with algebraic manipulations. If p = 1, A(n) = (n + 1)/2, as before, whereas if p = 1/2, A(n) = 3n/4 + 1/4. This means that about 3/4 of the array is searched on the average.

# Average-Case Time Complexity (Sequential Search) ctd..

- Caution: An average can be called “typical” only if the actual cases do not deviate much from the average (that is, only if the standard deviation is small).

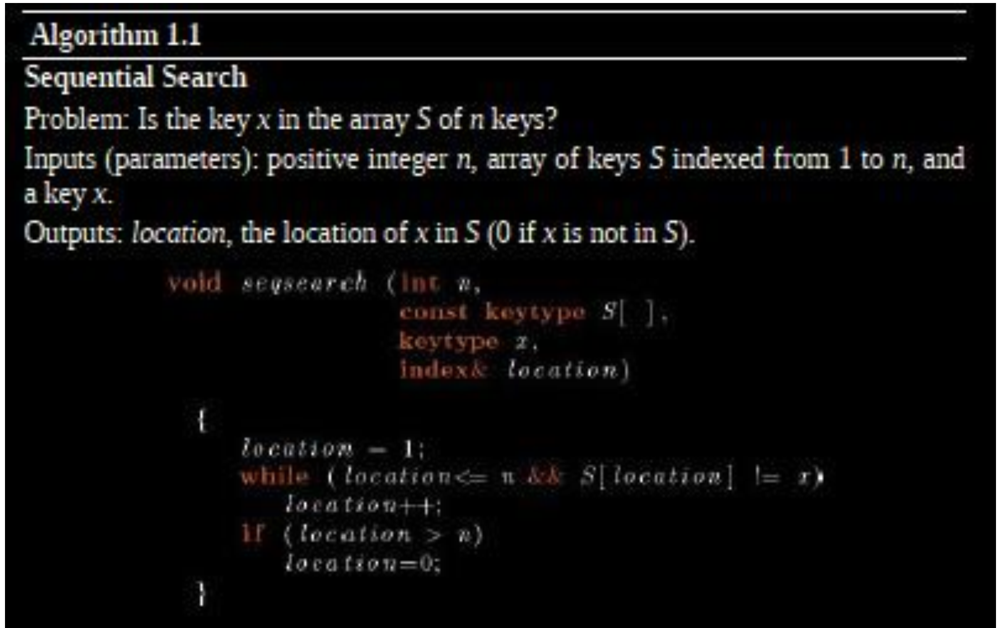
# Best-case time complexity analysis.

- A final type of time complexity analysis is the determination of the smallest number of times the basic operation is done.
- For a given algorithm, B(n) is defined as the minimum number of times the algorithm will ever do its basic operation for an input size of n.
- So B(n) is called the best-case time complexity of the algorithm, and the determination of B(n) is called a best-case time complexity analysis.
- As is the case for W(n) and A(n), if T(n) exists, then B(n) = T(n). Let’s determine B(n) for Algorithm 1.1.

# Best-Case Time Complexity (Sequential Search)

- Basic operation: the comparison of an item in the array with x.
- Input size: n, the number of items in the array.
- Because n ≥ 1, there must be at least one pass through the loop, If x = S[1], there will be one pass through the loop regardless of the size of n.
- Therefore,

$$B(n) = 1.$$



**Algorithm 1.1**

**Sequential Search**

Problem: Is the key *x* in the array *S* of *n* keys?

Inputs (parameters): positive integer *n*, array of keys *S* indexed from 1 to *n*, and a key *x*.

Outputs: *location*, the location of *x* in *S* (0 if *x* is not in *S*).

```
void seqsearch (int n,
                const keytype S[ ],
                keytype x,
                index& location)
{
   location = 1;
   while (location<= n && S[location] != x)
      location++;
   if (location > n)
      location=0;
}
```

# Time Complexity

- For algorithms that do not have every-case time complexities, we do worst-case and average-case analyses much more often than best-case analyses.
- An average-case analysis is valuable because it tells us how much time the algorithm would take when used many times on many different inputs.
- This would be useful, for example, in the case of a sorting algorithm that was used repeatedly to sort all possible inputs.
- Often, a relatively slow sort can occasionally be tolerated if, on the average, the sorting time is good. Quicksort, that does exactly this.
- It is one of the most popular sorting algorithms.

# Time Complexity

- As noted previously, an average-case analysis would not suffice in a system that monitored a nuclear power plant.
  - Ex: Life Critical Systems or Mission Critical Systems? State examples
- In this case, a worst-case analysis would be more useful because it would give us an upper bound on the time taken by the algorithm.
- For both the applications just discussed, a best-case analysis would be of little value.

# Further examples

- Best Case : Best case performance used in computer science to describe an algorithm’s behavior under optimal conditions. An example of best case performance would be trying to sort a list that is already sorted using some sorting algorithm. e.g. [1,2,3] --> [1,2,3]
- Average Case : Average case performance measured using the average optimal conditions to solve the problem. For example a list that is neither best case nor, worst case order that you want to be sorted in a certain order. e.g. [2,1,5,3] --> [1,2,3,5] OR [ 2,1,5,3] --> [5,3,2,1]
- Worst Case : Worst case performance used to analyze the algorithm's behavior under worst case input and least possible to solve the problem. It determines when the algorithm will perform worst for the given inputs. An example of the worst case performance would be a list of names already sorted in ascending order that you want to sort in descending order. e.g. [1,2,3,5] --> [5,3,2,1]

https://everythingcomputerscience.com/algorithms/Algorithm_Analysis.html

# Analysis of an Algorithm

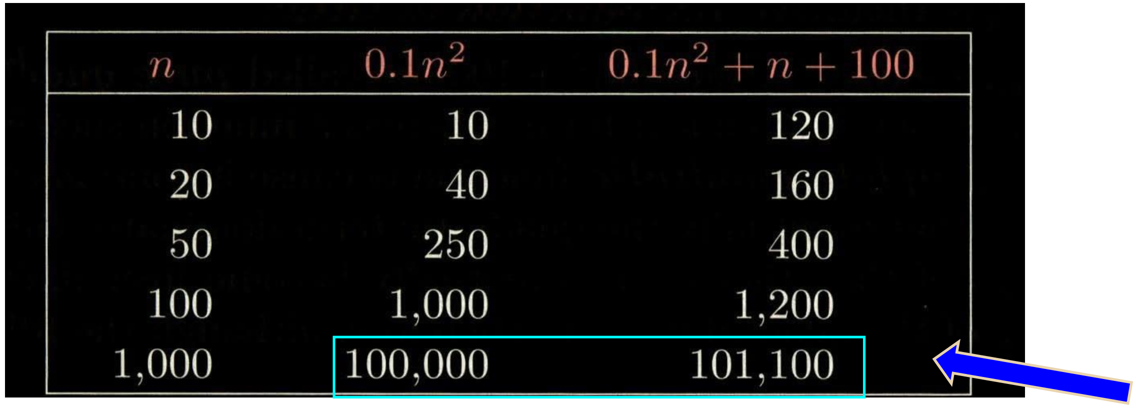
- "analysis of an algorithm" means an efficiency analysis in terms of either time or memory.
- Algorithms with time complexities such as n and 100n are called linear-time algorithms because their time complexities are linear in the input size n, whereas algorithms with time complexities such as $n^2$ and $0.01n^2$ are called quadratic time algorithms because their time complexities are quadratic in the input size n.

# Analysis of an Algorithm ctd..

- Any linear-time algorithm is eventually more efficient than any quadratic-time algorithm.
- In the theoretical analysis of an algorithm, we are interested in eventual behavior.
- Next we will show how **algorithms can be grouped according to their eventual behavior**.
- In this way we can readily determine whether one algorithm's eventual behavior is better than another's.

# An Intuitive Introduction to Order

- Functions such as $5n^2$ and $5n^2 + 100$ are called **pure quadratic** functions because they contain no linear term, whereas a function such as $0.1n^2 + n + 100$ is called a **complete quadratic** function because it contains a linear term. Table shows that eventually the quadratic term dominates this function.



| $n$ | $0.1n^2$ | $0.1n^2 + n + 100$ |
|---|---|---|
| 10 | 10 | 120 |
| 20 | 40 | 160 |
| 50 | 250 | 400 |
| 100 | 1,000 | 1,200 |
| 1,000 | 100,000 | 101,100 |

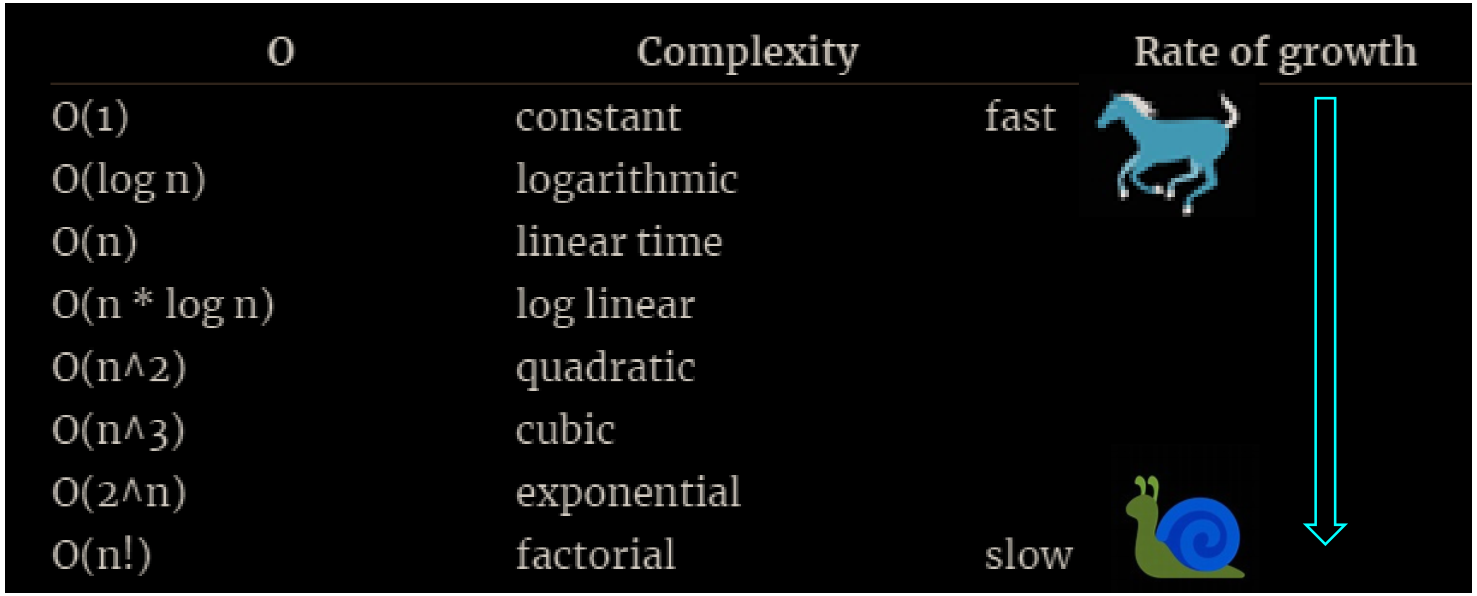
- That is, the values of the other terms eventually become insignificant compared with the value of the quadratic term.

# An Intuitive Introduction to Order

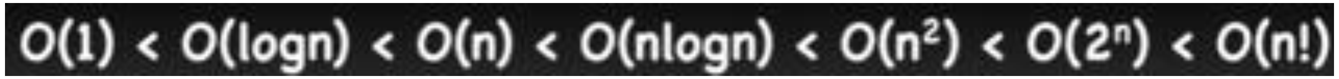
- Therefore, although the function is **not a pure quadratic function**, we can **classify** it with the **pure quadratic** functions.
- This means that if some algorithm has this time complexity, we can call the algorithm a quadratic-time algorithm.
- Intuitively, it seems that we should always be able to throw away low-order terms when classifying complexity functions.
- For example, it seems that we should be able to classify $0.1n^3 + 10n^2 + 5n + 25$ with pure cubic functions.

# Representative Order Functions

- $\Theta(\lg n)$
- $\Theta(n)$ : linear
- $\Theta(n \lg n)$ log-Linear
- $\Theta(n^2)$ : quadratic
- $\Theta(n^3)$ : cubic
- $\Theta(2^n)$ : exponential
- $\Theta(n!)$ : combinatorial



| O | Complexity | Rate of growth |
|---|---|---|
| O(1) | constant | fast |
| O(log n) | logarithmic | |
| O(n) | linear time | |
| O(n * log n) | log linear | |
| O(n^2) | quadratic | |
| O(n^3) | cubic | |
| O(2^n) | exponential | |
| O(n!) | factorial | slow |

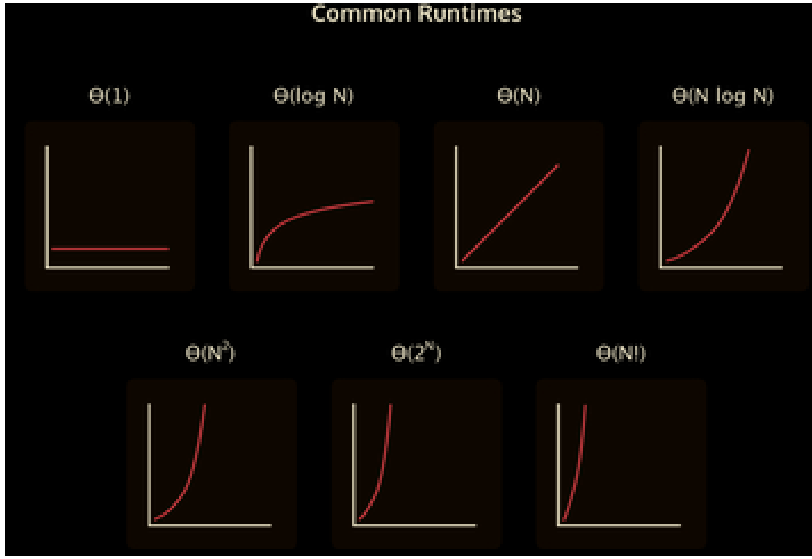


$O(1) < O(logn) < O(n) < O(nlogn) < O(n^2) < O(2^n) < O(n!)$

- In Big-O complexity analysis, it doesn't matter what the logarithm base is. (they are asymptotically the same, i.e. they differ by only a constant factor/implies that it's only a constant factor difference):
    - $O(\log_2 N) = O(\log_{10} N) = O(\log_e N)$
- Mostly in mathematics it implicitly mean to base e. In Computing it tend to favors base 2, but it does not actually matter.

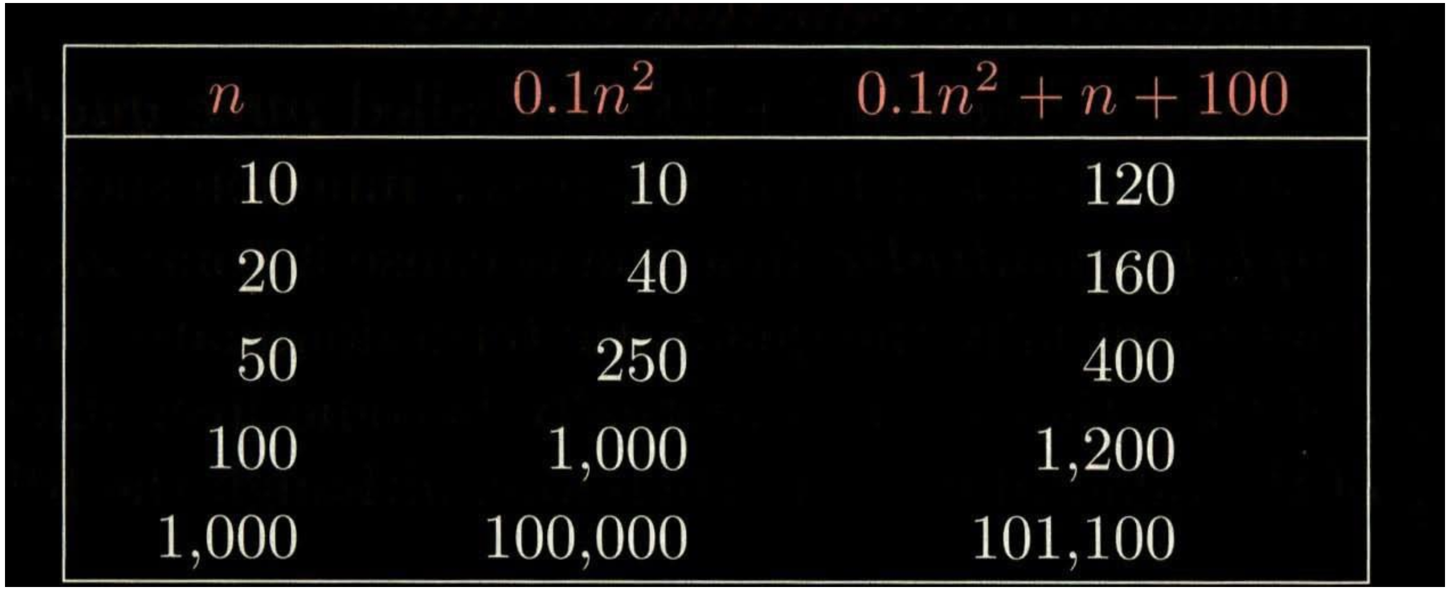
# Common runtimes examples

- O(1) Constant Time: The algorithm’s running time does not depend on the size of the input; it performs a fixed number of operations.
- O(log n) Logarithmic Time: The algorithm’s running time grows logarithmically with the size of the input.
- O(n) Linear Time: The algorithm’s running time scales linearly with the size of the input.
- O(n log n) Linearithmic Time: The algorithm’s running time grows in proportion to n times the logarithm of n.
- O($n^2$) Quadratic Time: The algorithm’s running time is directly proportional to the square of the input size.
- O(2^n) Exponential Time: The algorithm’s running time doubles with each increase in the input size.



# Example

- The  quadratic term eventually determines

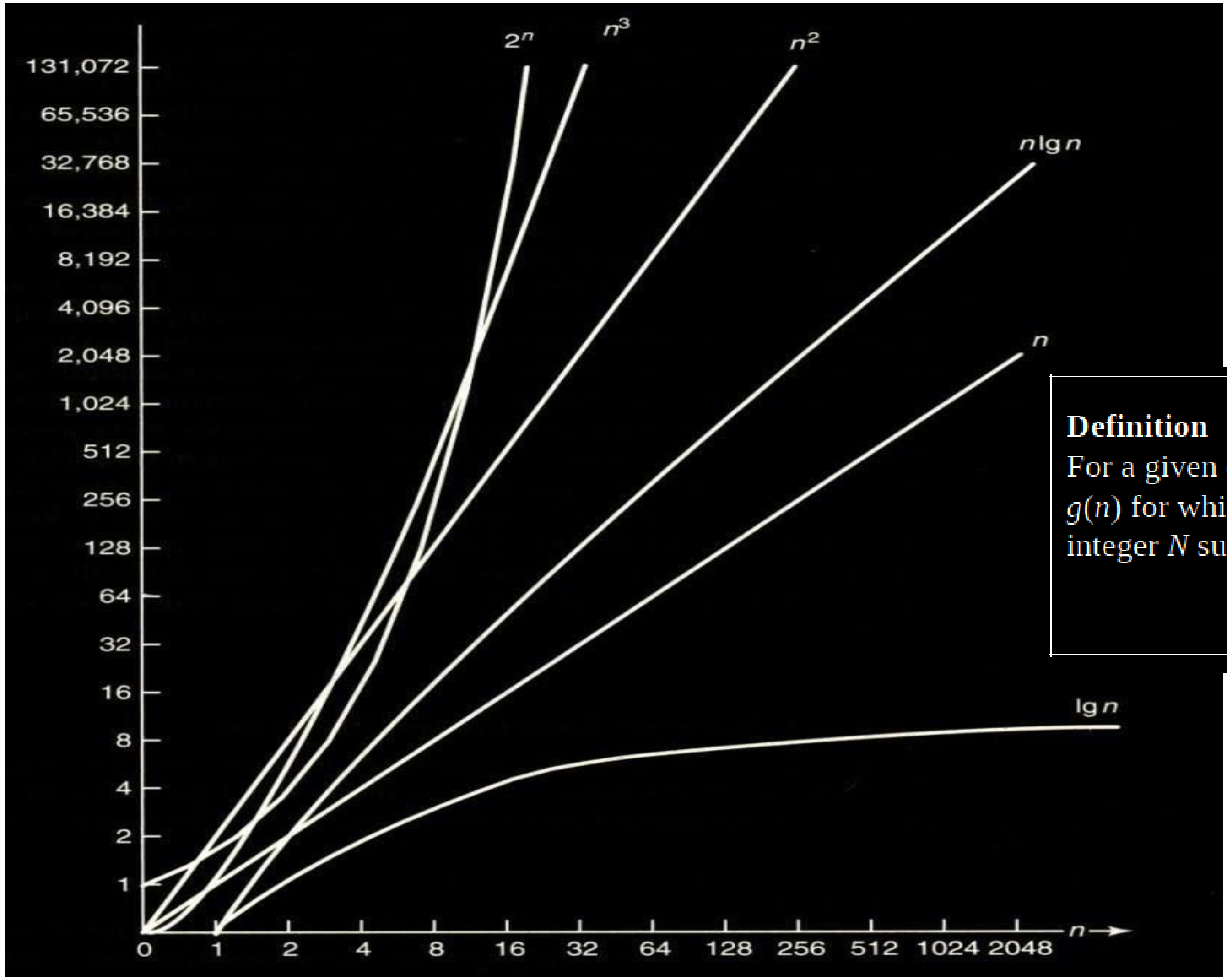


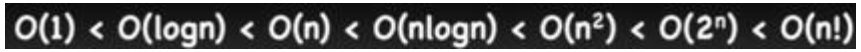
| $n$ | $0.1n^2$ | $0.1n^2 + n + 100$ |
|---:|---:|---:|
| 10 | 10 | 120 |
| 20 | 40 | 160 |
| 50 | 250 | 400 |
| 100 | 1,000 | 1,200 |
| 1,000 | 100,000 | 101,100 |

# Big O

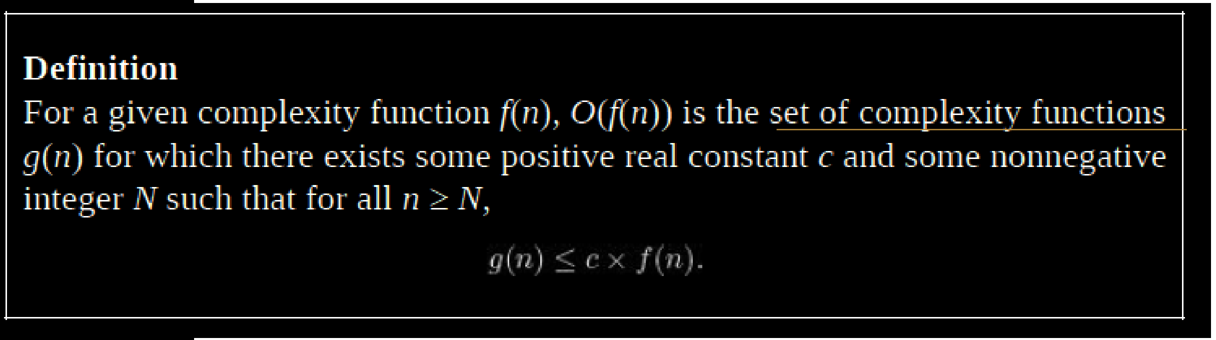
- The “O” in Big O stands for “order ” while the value within parentheses indicates the growth rate of the algorithm.
- In the case of O(N), we refer to it as complexity. This implies that the execution time of the algorithm increases proportionally with respect, to the size of the input. If we double our input size we can expect twice as much execution time.
- drop the non-dominant terms, then how about n*log n?"
  - While it is true that we drop the non-dominant terms in Big O, that's generally when we are adding two different complexities, such as $n^2 + n$. Here, we are using multiplication. We can not simplify n * log n any further, so we keep both terms.

# Growth Rates of Some Complexity Functions





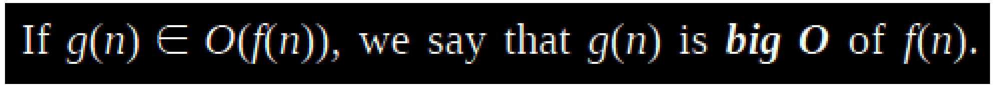
O(1) < O(logn) < O(n) < O(nlogn) < O(n²) < O(2ⁿ) < O(n!)



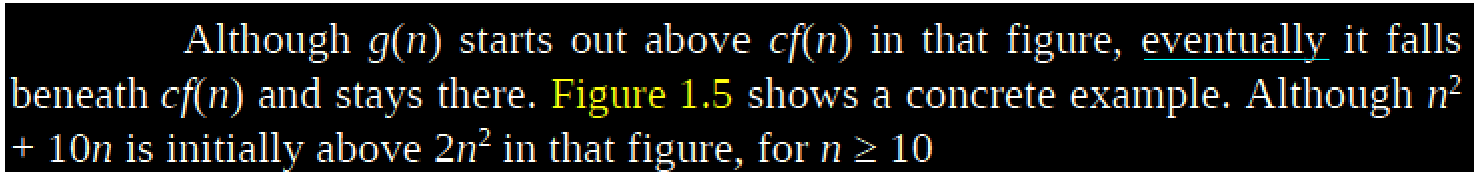
**Definition**

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 nonnegative integer $N$ such that for all $n \geq N$,

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

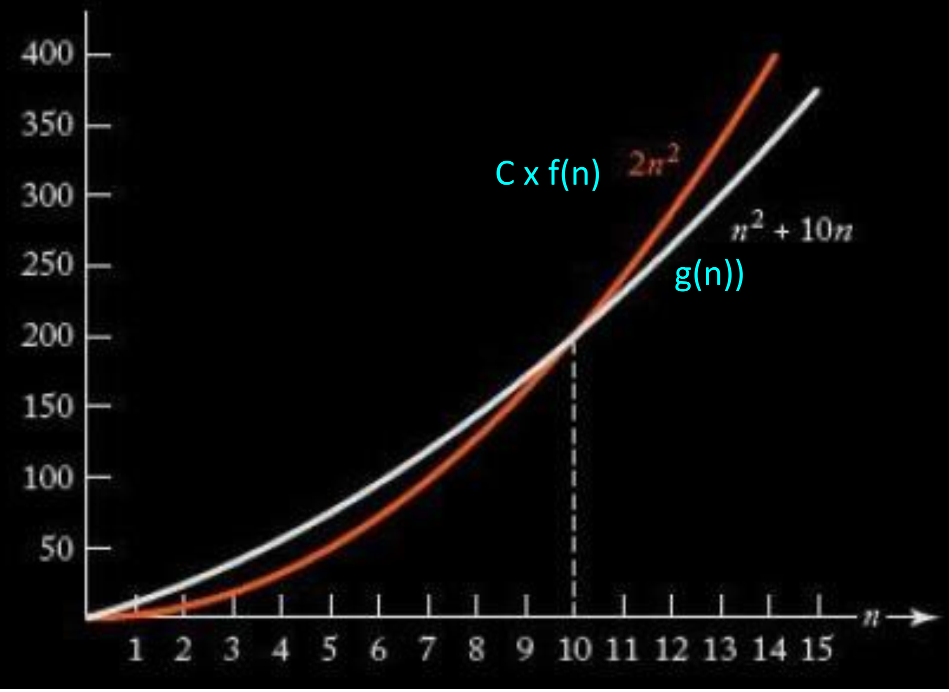


If $g(n) \in O(f(n))$, we say that $g(n)$ is ***big O*** of $f(n)$.

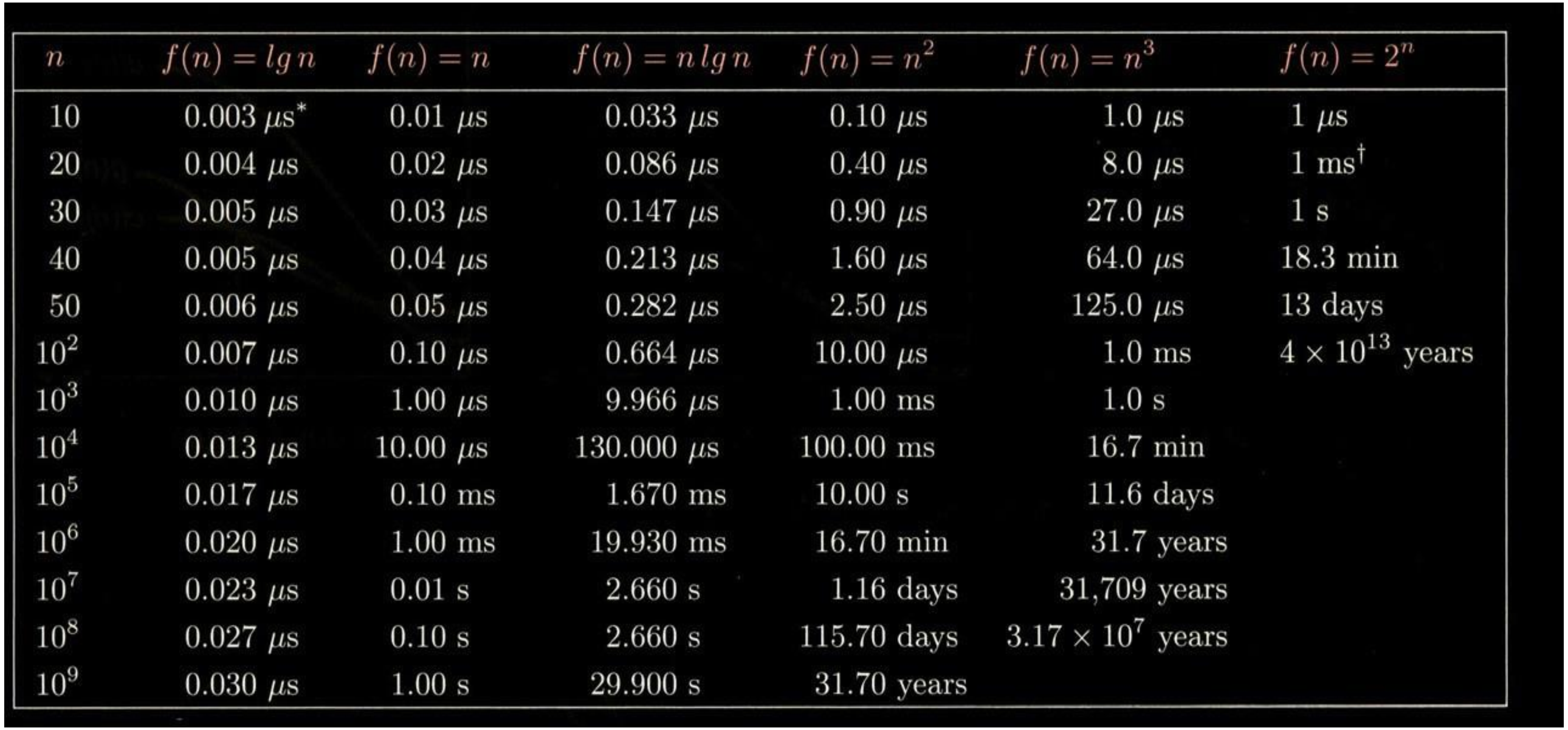


Although $g(n)$ starts out above $cf(n)$ in that figure, eventually it falls beneath $cf(n)$ and stays there. Figure 1.5 shows a concrete example. Although $n^2 + 10n$ is initially above $2n^2$ in that figure, for $n \geq 10$

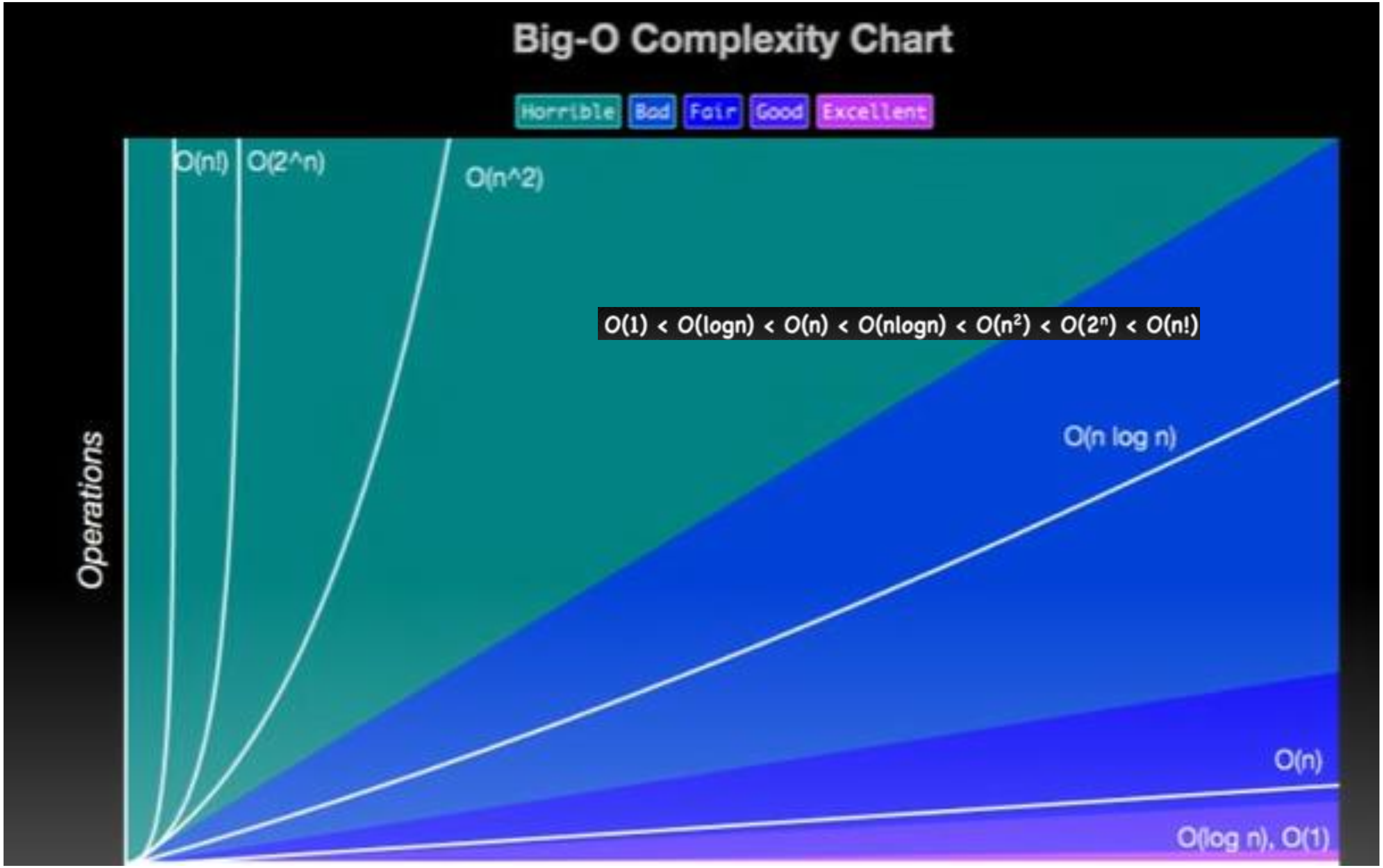
**Figure 1.5** The function $n^2 + 10n$ eventually stays beneath the function $2n^2$.



# Execution Times for Algorithms with the Given Time Complexities



| $n$ | $f(n) = lg\,n$ | $f(n) = n$ | $f(n) = n\,lg\,n$ | $f(n) = n^2$ | $f(n) = n^3$ | $f(n) = 2^n$ |
|---|---|---|---|---|---|---|
| 10 | 0.003 $\mu$s$^*$ | 0.01 $\mu$s | 0.033 $\mu$s | 0.10 $\mu$s | 1.0 $\mu$s | 1 $\mu$s |
| 20 | 0.004 $\mu$s | 0.02 $\mu$s | 0.086 $\mu$s | 0.40 $\mu$s | 8.0 $\mu$s | 1 ms$^\dagger$ |
| 30 | 0.005 $\mu$s | 0.03 $\mu$s | 0.147 $\mu$s | 0.90 $\mu$s | 27.0 $\mu$s | 1 s |
| 40 | 0.005 $\mu$s | 0.04 $\mu$s | 0.213 $\mu$s | 1.60 $\mu$s | 64.0 $\mu$s | 18.3 min |
| 50 | 0.006 $\mu$s | 0.05 $\mu$s | 0.282 $\mu$s | 2.50 $\mu$s | 125.0 $\mu$s | 13 days |
| $10^2$ | 0.007 $\mu$s | 0.10 $\mu$s | 0.664 $\mu$s | 10.00 $\mu$s | 1.0 ms | $4 \times 10^{13}$ years |
| $10^3$ | 0.010 $\mu$s | 1.00 $\mu$s | 9.966 $\mu$s | 1.00 ms | 1.0 s | |
| $10^4$ | 0.013 $\mu$s | 10.00 $\mu$s | 130.000 $\mu$s | 100.00 ms | 16.7 min | |
| $10^5$ | 0.017 $\mu$s | 0.10 ms | 1.670 ms | 10.00 s | 11.6 days | |
| $10^6$ | 0.020 $\mu$s | 1.00 ms | 19.930 ms | 16.70 min | 31.7 years | |
| $10^7$ | 0.023 $\mu$s | 0.01 s | 2.660 s | 1.16 days | 31,709 years | |
| $10^8$ | 0.027 $\mu$s | 0.10 s | 2.660 s | 115.70 days | $3.17 \times 10^7$ years | |
| $10^9$ | 0.030 $\mu$s | 1.00 s | 29.900 s | 31.70 years | | |

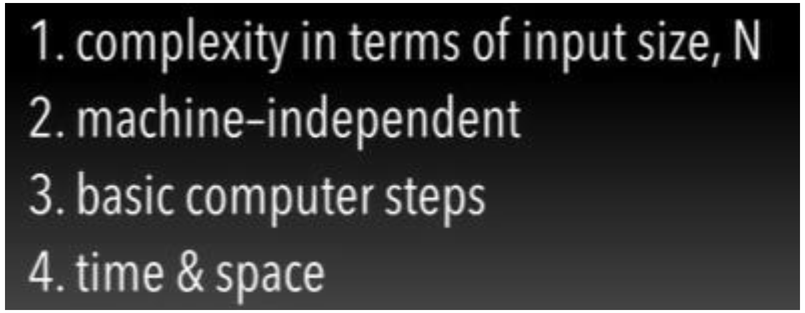


$O(1) < O(\log n) < O(n) < O(n\log n) < O(n^2) < O(2^n) < O(n!)$

Big-O notation in 5 minutes
https://www.youtube.com/watch?v=__vX2sjlpXU

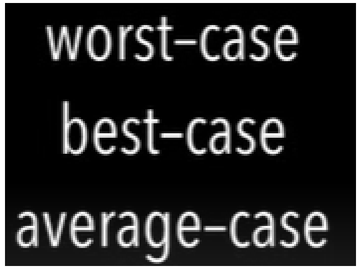
# Big O?

- Simplified Analysis of an Algorithms Efficiency



1. complexity in terms of input size, N
2. machine-independent
3. basic computer steps
4. time & space

# Type of Measurement?



worst-case

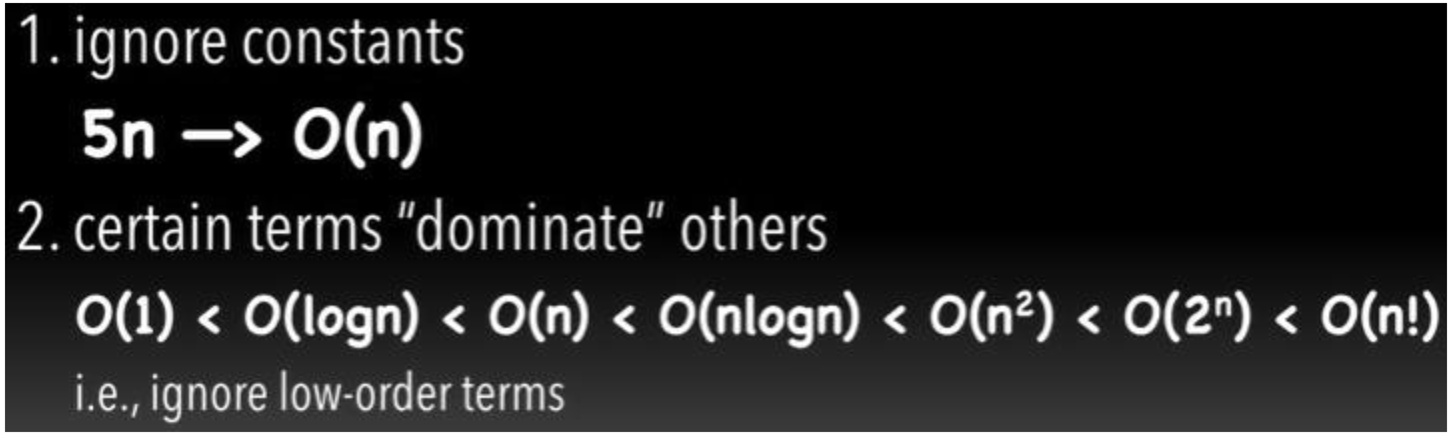
best-case

average-case

Big-O notation in 5 minutes
https://www.youtube.com/watch?v=__vX2sjlpXU

# Big O?

- General Rules



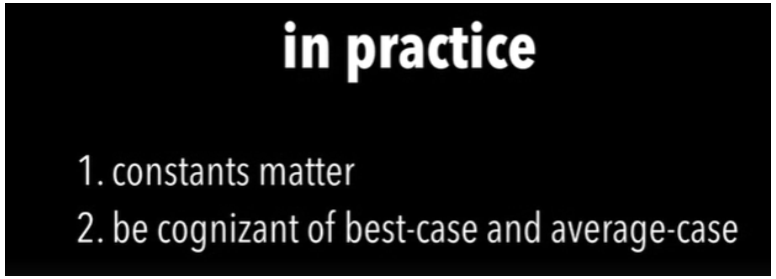
1. ignore constants

   **5n —> O(n)**

2. certain terms “dominate” others

   **$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(2^n) < O(n!)$**

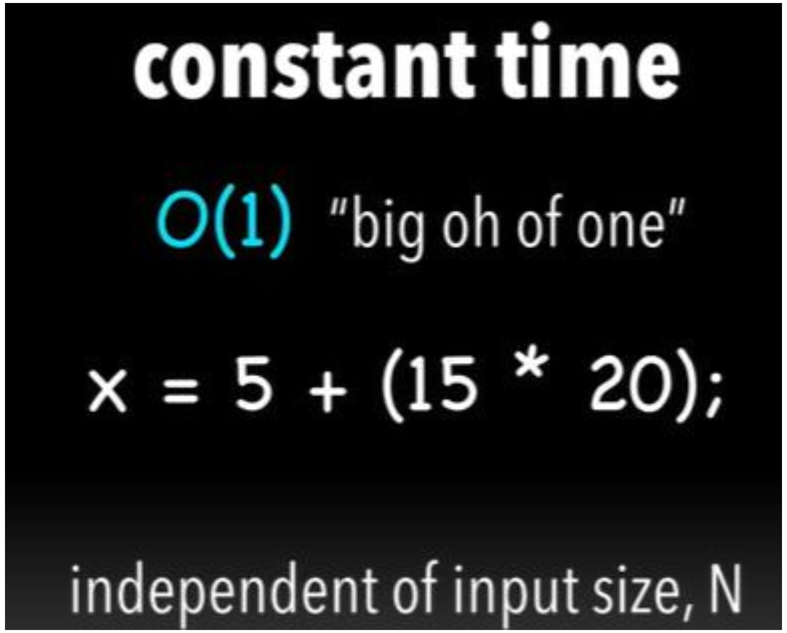
   i.e., ignore low-order terms



**in practice**

1. constants matter
2. be cognizant of best-case and average-case

Big-O notation in 5 minutes
https://www.youtube.com/watch?v=__vX2sjlpXU

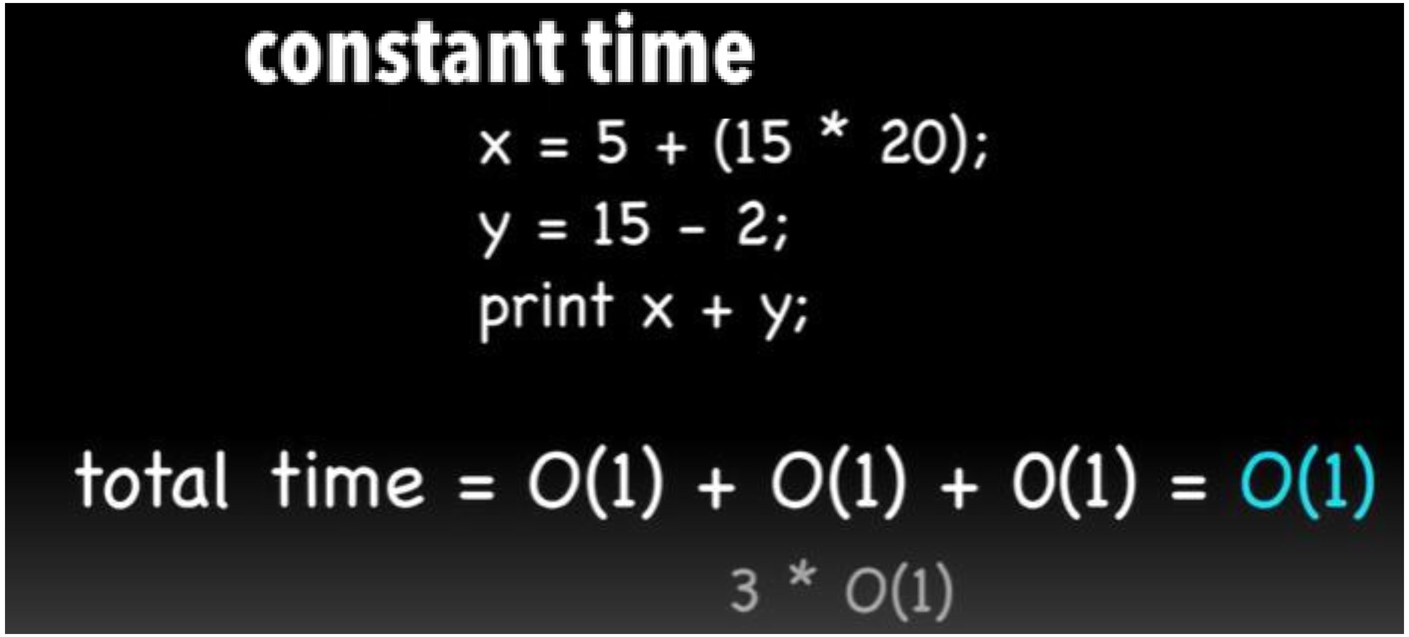


## constant time

O(1) "big oh of one"

```
x = 5 + (15 * 20);
```

independent of input size, N

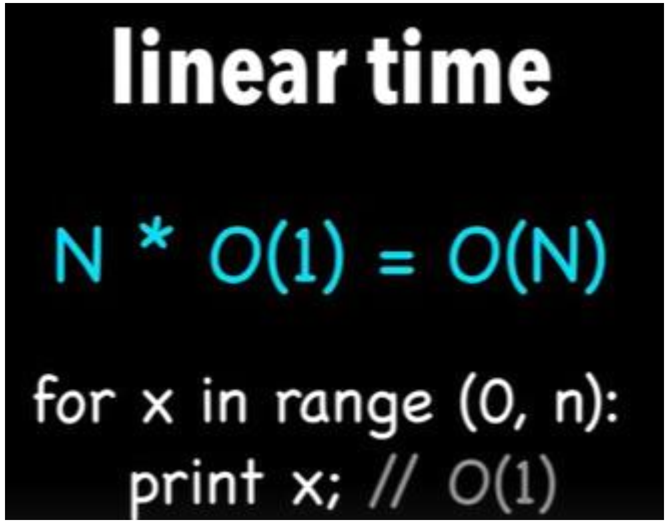


## constant time

```
x = 5 + (15 * 20);
y = 15 - 2;
print x + y;
```

total time = O(1) + O(1) + O(1) = O(1)

3 * O(1)

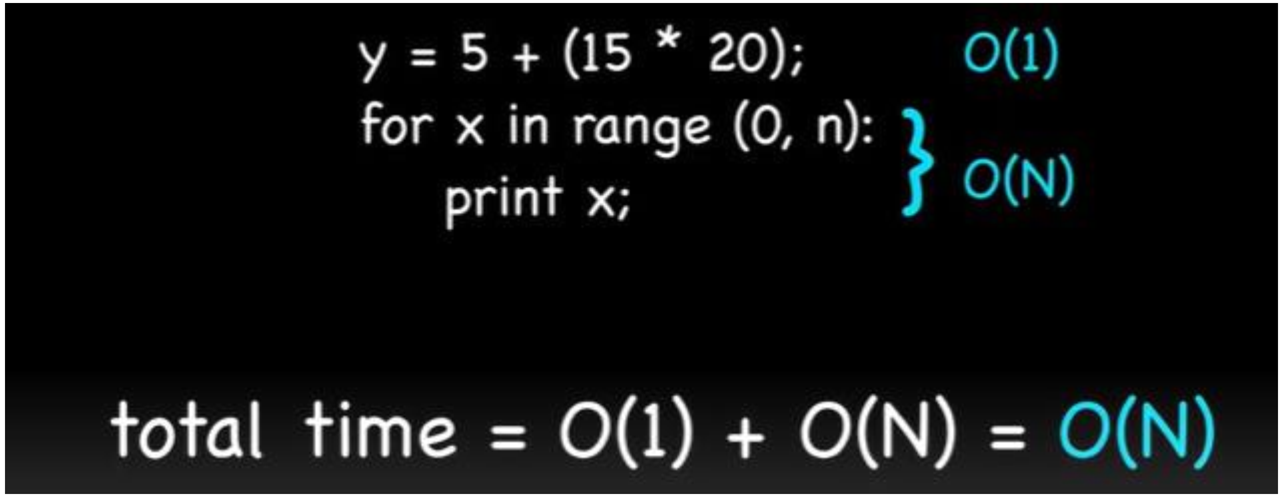


## linear time

N * O(1) = O(N)

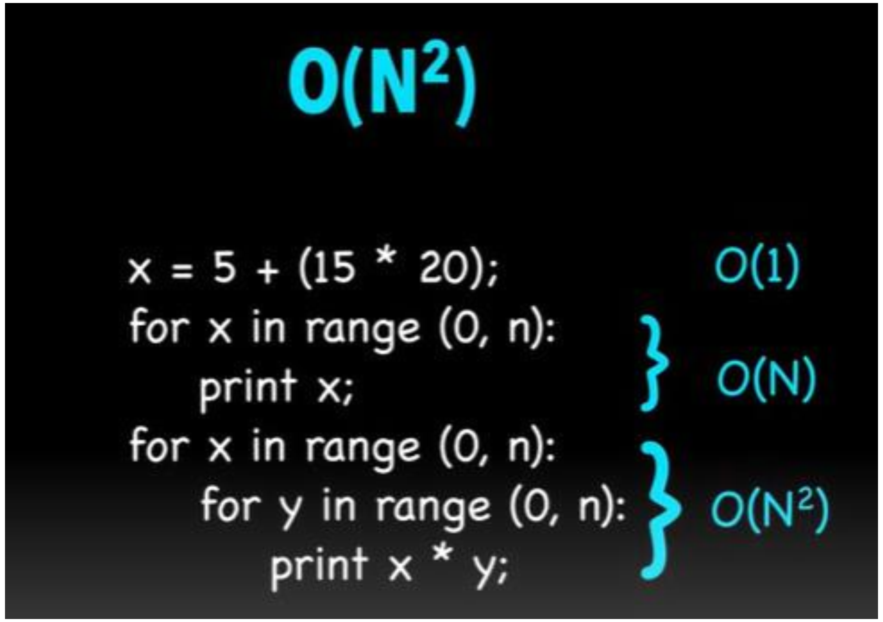
```
for x in range (0, n):
    print x; // O(1)
```

Big-O notation in 5 minutes
https://www.youtube.com/watch?v=__vX2sjlpXU



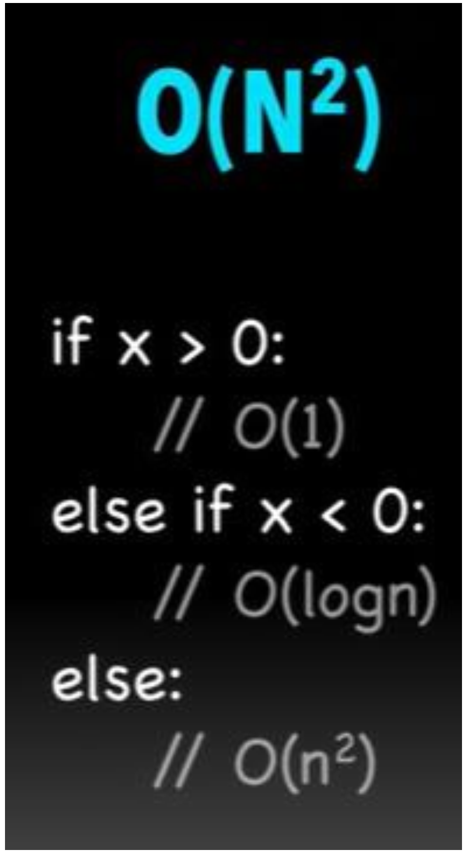
```
y = 5 + (15 * 20);          O(1)
for x in range (0, n):  }
     print x;           }   O(N)
```

total time = O(1) + O(N) = O(N)



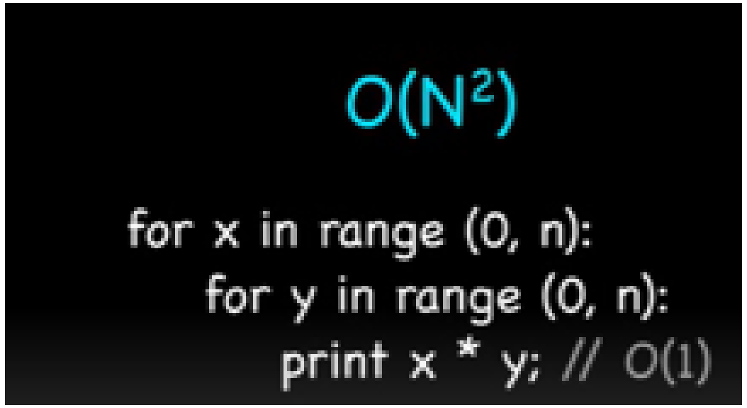
O(N²)

```
x = 5 + (15 * 20);                O(1)
for x in range (0, n):       }
     print x;                }    O(N)
for x in range (0, n):       }
     for y in range (0, n):  }    O(N²)
          print x * y;       }
```



O(N²)

```
if x > 0:
     // O(1)
else if x < 0:
     // O(logn)
else:
     // O(n²)
```



O(N²)

```
for x in range (0, n):
     for y in range (0, n):
          print x * y; // O(1)
```

Big-O notation in 5 minutes
https://www.youtube.com/watch?v=__vX2sjlpXU

# Examples

- The algorithm complexity ignores the constant value in algorithm analysis and takes only the highest order.
- Suppose we had an algorithm that takes, $5n^3+n+4$ time to calculate all the steps, then the algorithm analysis ignores all the **lower order polynomials and constants** and takes only $O(n^3)$.

https://everythingcomputerscience.com/algorithms/Algorithm_Analysis.html

# Examples ctd..

## Simple Statement

This statement takes O(1) time.

```
int y= n + 25;
```

Constant Time Complexity O(1):
The time complexity of a function (or set of statements) is considered as O(1) if it doesn’t contain a loop, recursion, and call to any other non-constant time function.
i.e. set of non-recursive and non-loop statements

## If Statement

The worst case O(n) if the if statement is in a loop that runs n times, best case O(1)

```
if( n> 100)
{
    …
}else{
    ..
}
```

https://everythingcomputerscience.com/algorithms/Algorithm_Analysis.html

# Further Examples ctd..

For  While Loops

The for loop takes n time to complete and so it is O(n).

```
for(int i=0; i<n ;I ++)
{
   ..
}
```

https://everythingcomputerscience.com/algorithms/Algorithm_Analysis.html

# Examples ctd..

- The while loop takes n time as well to complete and so it is O(n).

```
int i=0;
while( i<n)
{
    ..
    i++;
}
```

- If the for loop takes n time and i increases or decreases by a constant, the cost is O(n)

```
for(int i = 0; i < n; i+=5)
   sum++;

for(int i = n; i > 0; i-=5)
   sum++;
```

denoted as O(n), is a measure of the growth of the running time of an algorithm proportional to the size of the input. In an O(n) algorithm, the running time increases linearly with the size of the input.

https://everythingcomputerscience.com/algorithms/Algorithm_Analysis.html

# Examples ctd..

- If the for loop takes n time and i increases or decreases by a multiple, the cost is O(log(n))

```
for(int i = 1; i < =n; i*=2)
   sum++;

for(int i = n; i > 0; i/=2)
   sum++;
```

The time Complexity of a loop is considered as O(log n) if the loop variables are divided/multiplied by a constant amount. And also for recursive calls in the recursive function, the Time Complexity is considered as O(log n).

https://everythingcomputerscience.com/algorithms/Algorithm_Analysis.html

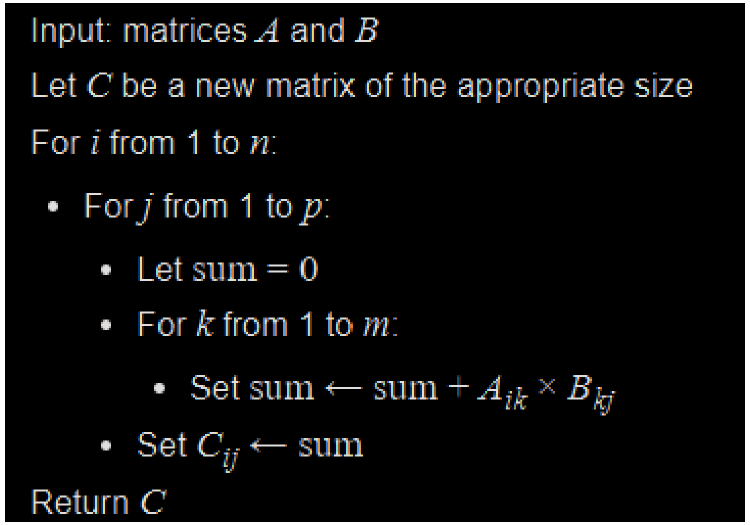
# Examples ctd..

## Nested loops

If the nested loops contain sizes n and m, the cost is O(nm)

```
for(int i=0; i<n; i++)
{
   for(int i=0; i<m; i++){
     ..
   }
}
```

Ex: What is the time complexity if C = AB for an n × m matrix A and an m × p matrix B, (then C is an n × p matrix with entries)



```
Input: matrices $A$ and $B$
Let $C$ be a new matrix of the appropriate size
For $i$ from 1 to $n$:
  • For $j$ from 1 to $p$:
      • Let $\mathrm{sum} = 0$
      • For $k$ from 1 to $m$:
          • Set $\mathrm{sum} \leftarrow \mathrm{sum} + A_{ik} \times B_{kj}$
      • Set $C_{ij} \leftarrow \mathrm{sum}$
Return $C$
```

Refer: https://www.geeksforgeeks.org/how-to-analyse-loops-for-complexity-analysis-of-algorithms/

https://everythingcomputerscience.com/algorithms/Algorithm_Analysis.html

# Examples ctd..

- If the first loop runs n times and the inner loop runs log(n) times or (vice versa), the cost is O(n*log(n))

```
for(int i=0; i<n ; i++)
{
   for(int j=1; i<=n; j*=2){
      ..
      ..
   }
}
```

https://everythingcomputerscience.com/algorithms/Algorithm_Analysis.html

# Examples ctd..

- If the first loop runs $n^2$ times and the inner loop runs n times or (vice versa), the cost is $O(n^3)$

```
for(int j=0 ;j<n*n ;j++)
{
   for(int i=0 ;i<n ;i++){
      ..
       ..
   }
}
```

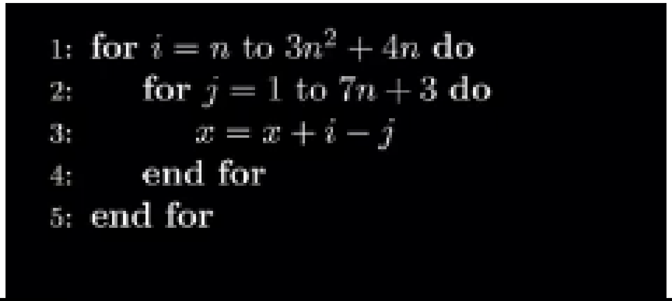
https://everythingcomputerscience.com/algorithms/Algorithm_Analysis.html

# Examples ctd..

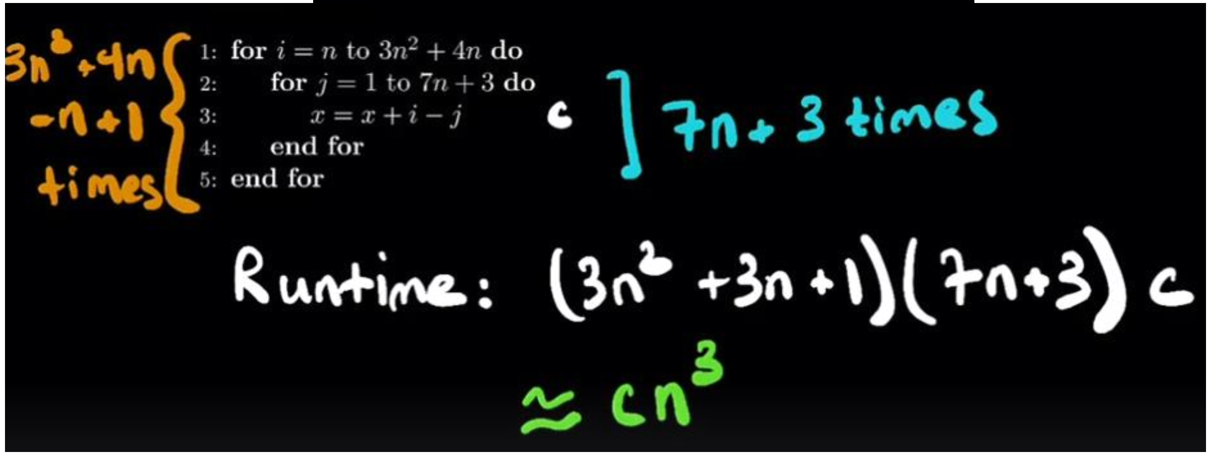
- If the first loop runs n times and the inner second loop runs $n^2$ times and the third loop runs $n^2$, then $O(n^5)$

```
for(int i = 0; i < n; i++)
    for( int j = 0; j < n * n; j++)
        for(int k = 0; k < j; k++)
            sum++;
```

Ex: What is O(?)



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



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

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

$c$ ] $7n+3$ times

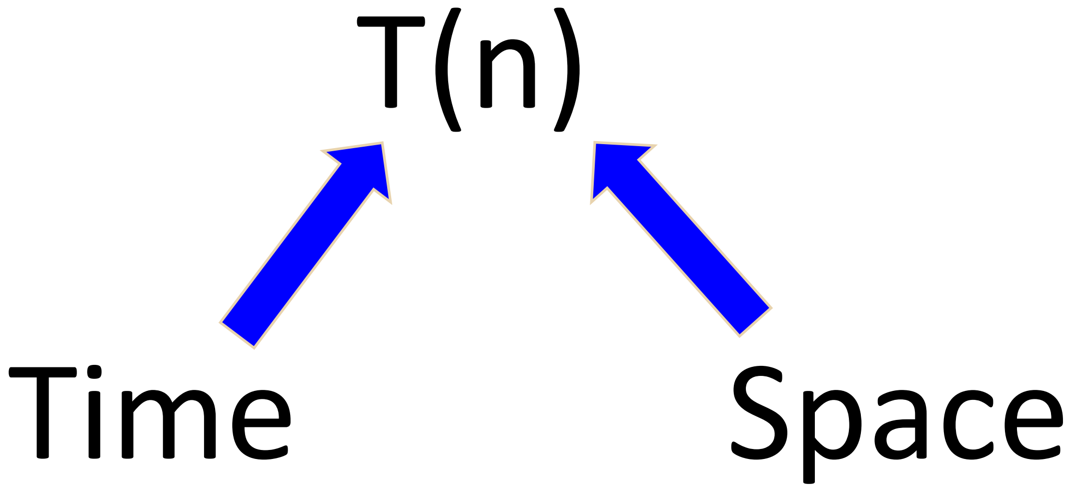
Runtime: $(3n^2+3n+1)(7n+3)c$

$\approx cn^3$

https://everythingcomputerscience.com/algorithms/Algorithm_Analysis.html

# References

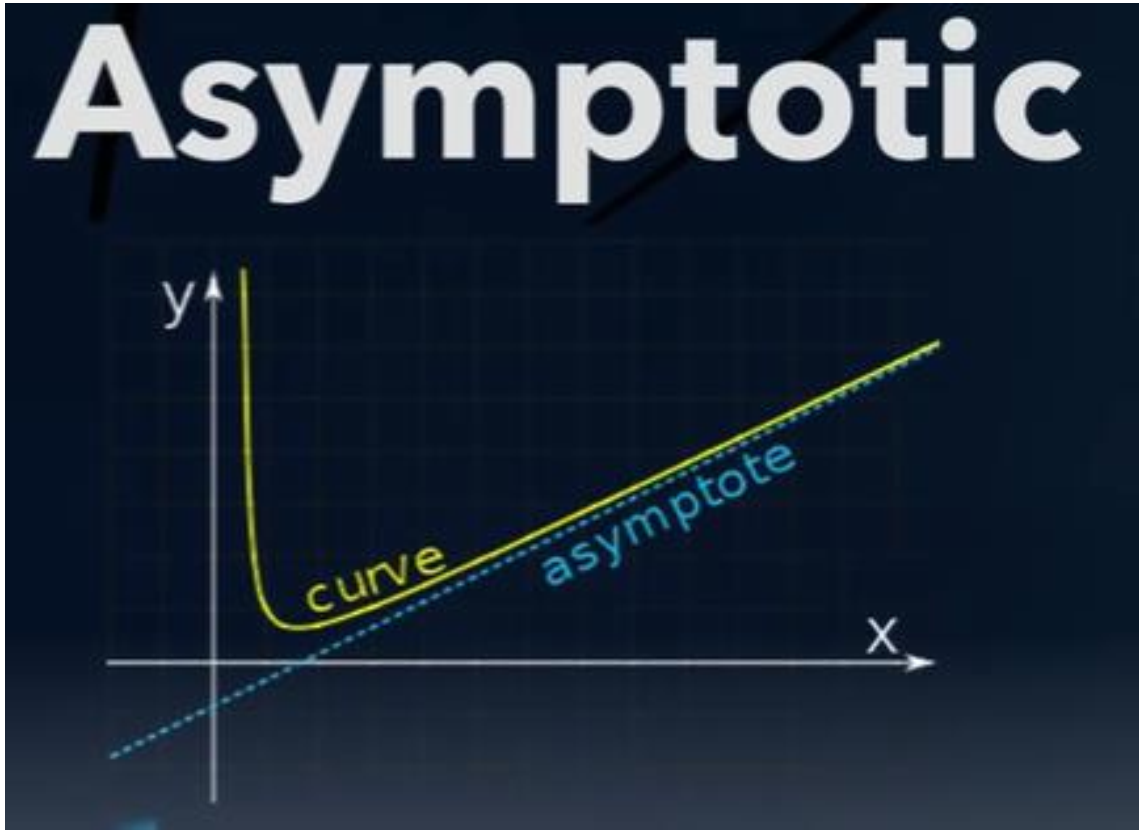
- Refer [Asymptotic Notations 101: Big O, Big Omega, & Theta (Asymptotic Analysis Bootcamp)]
  - https://www.youtube.com/watch?v=0oDAlMwTrLo&t=50s

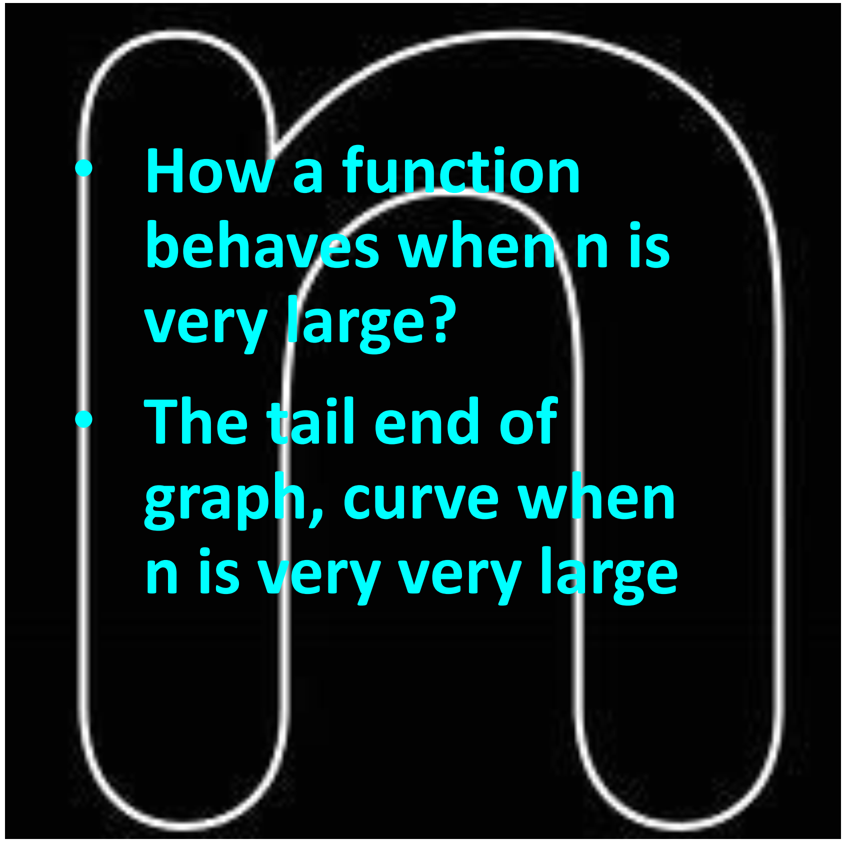


# Asymptotic Notations
# What does asymptotic mean?

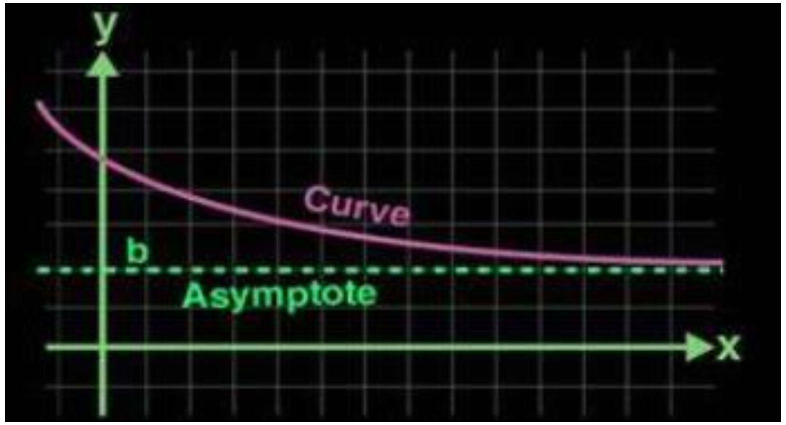
- The word asymptotic means approaching a certain value which could be considered as the limit. The value can range from any finite natural number to an infinite value.
- Definition

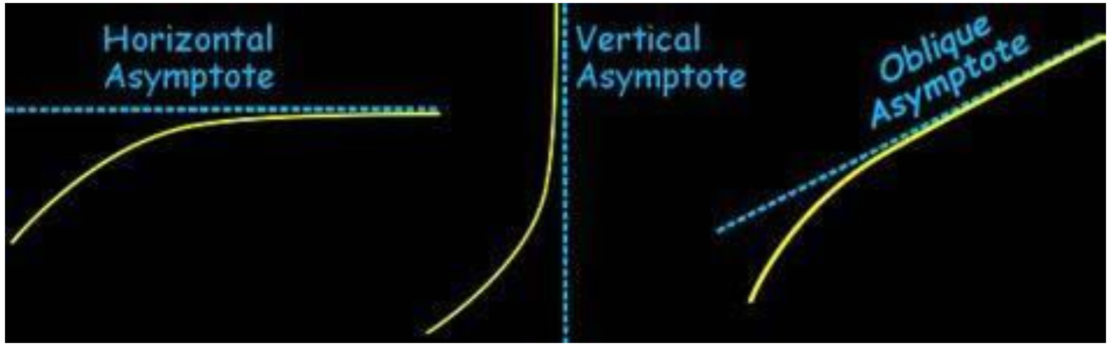
  “Asymptotic Notation is used to decide the asymptotic running time of an algorithm, when the input size n is very large, i.e, n-> ∞ . ‘n’ belongs to the set of natural numbers. “





- 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

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

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

# 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.

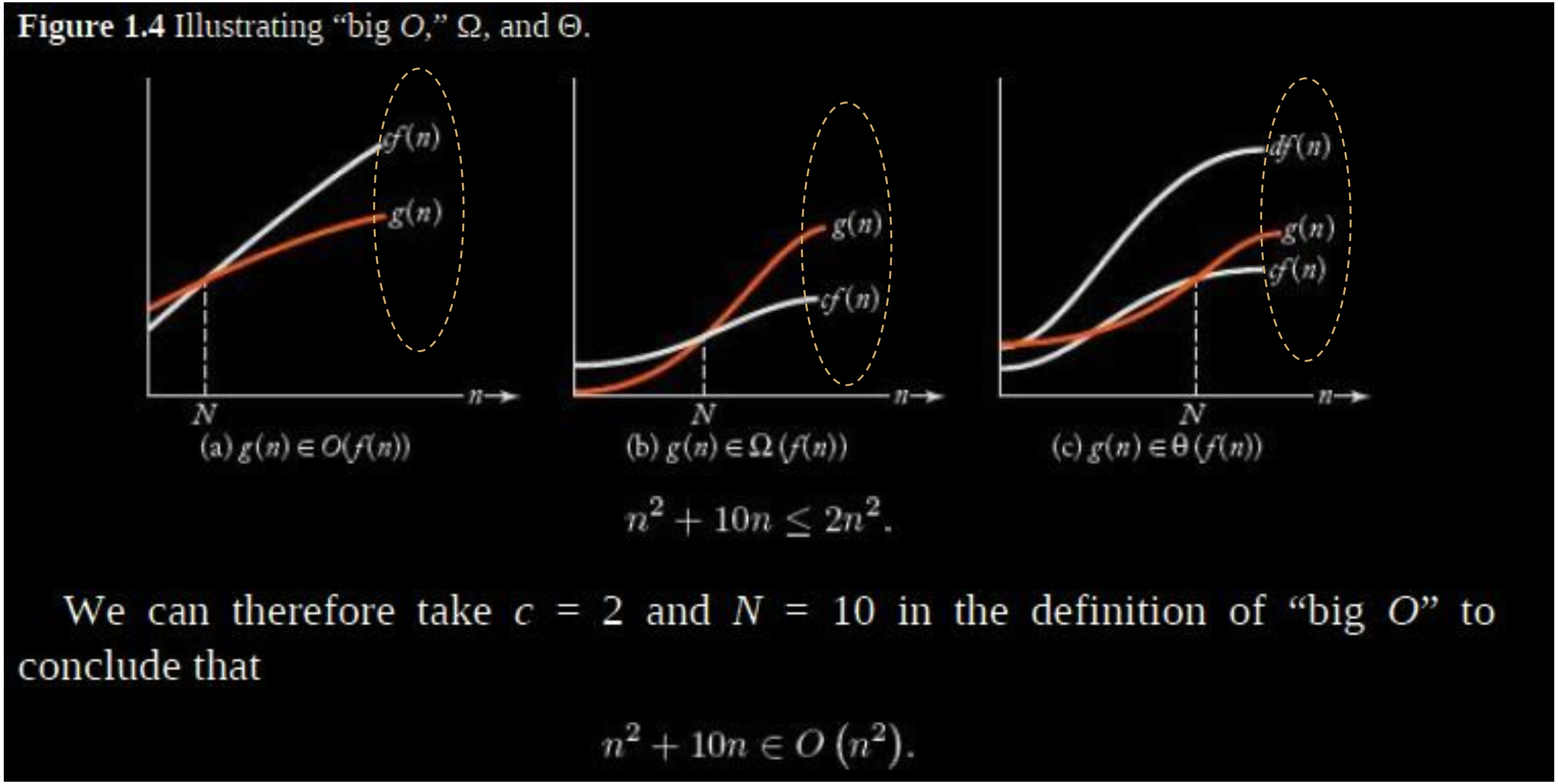


**Figure 1.4** Illustrating “big $O$,” Ω, and Θ.

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

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

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

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

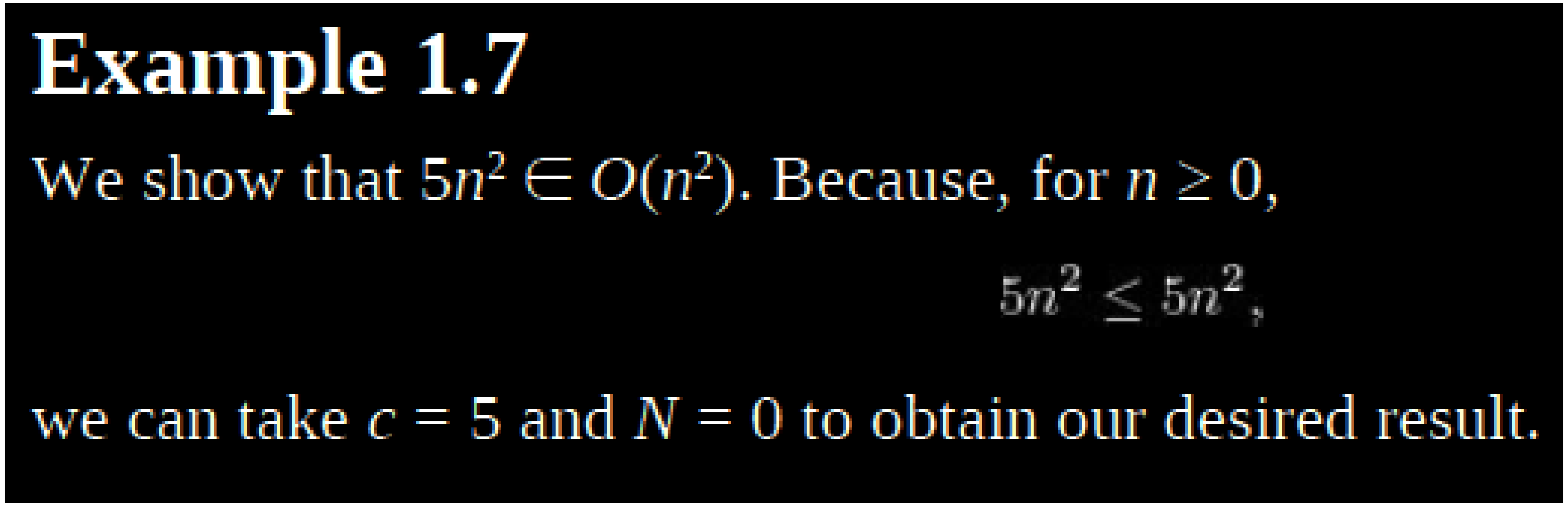
We can therefore take $c = 2$ and $N = 10$ in the definition of “big $O$” to conclude that

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

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 **c$n^2$** 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.



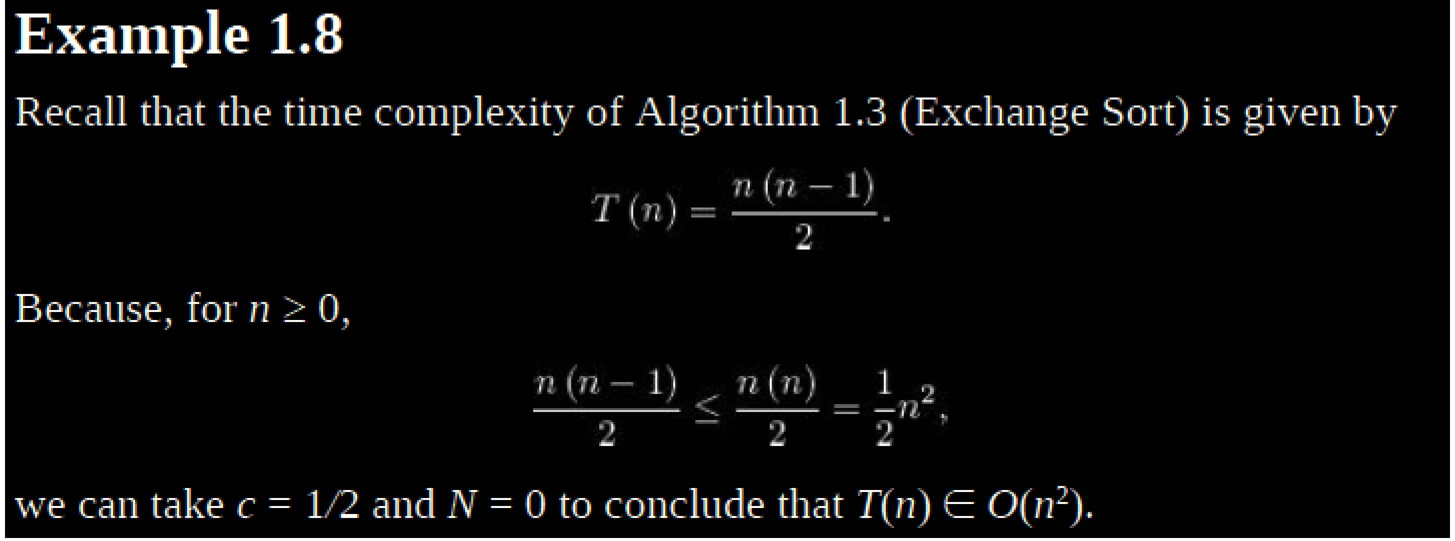
**Example 1.7**

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

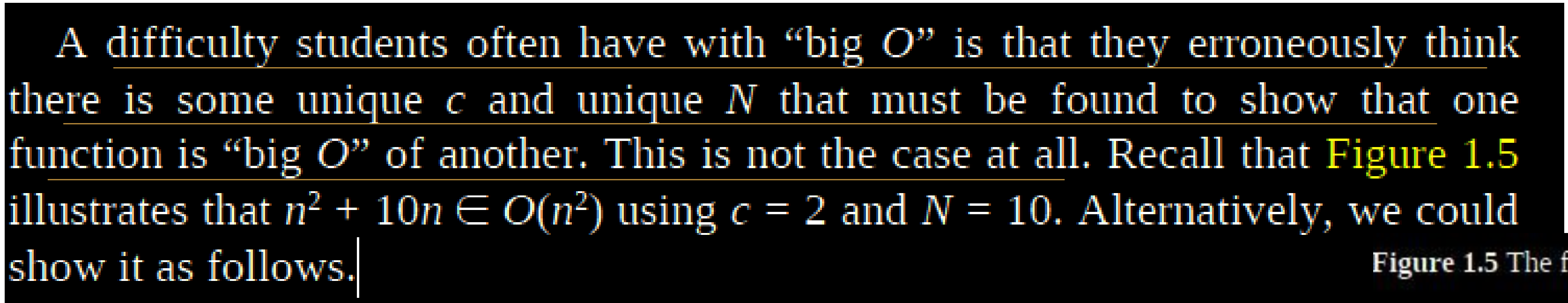
$$T(n) = \frac{n(n-1)}{2}.$$

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.

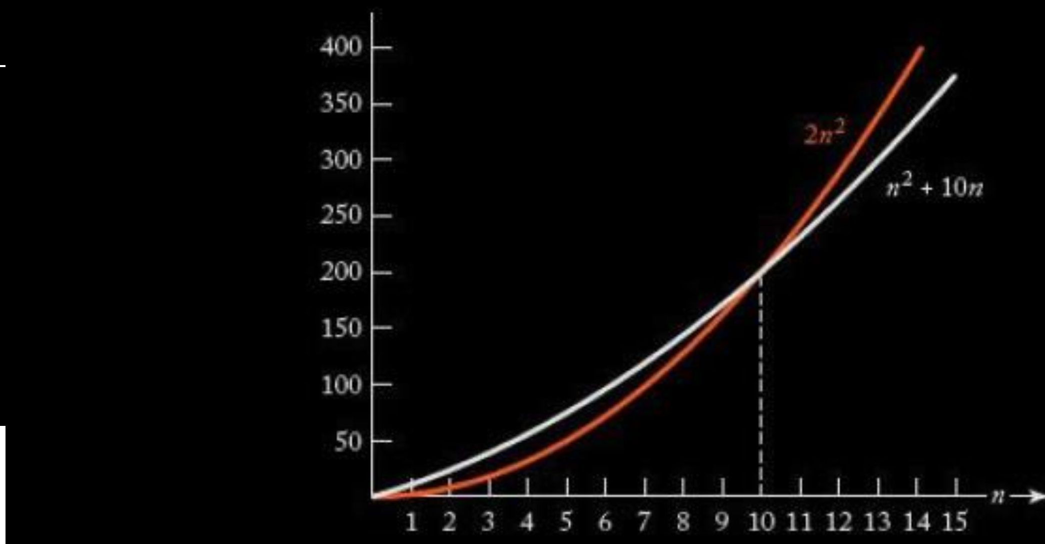
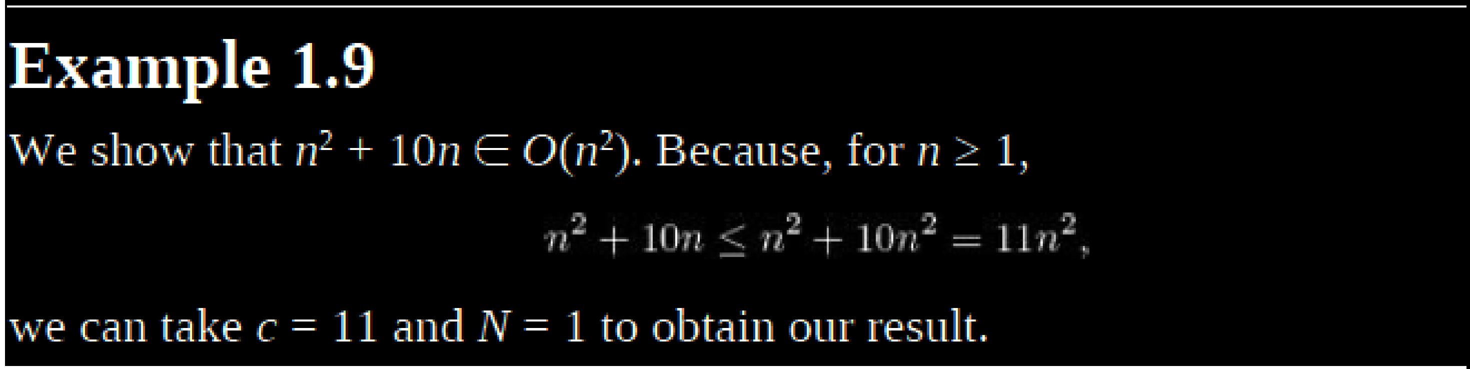


**Figure 1.5** The function $n^2 + 10n$ eventually stays beneath the function $2n^2$.



## Example 1.9

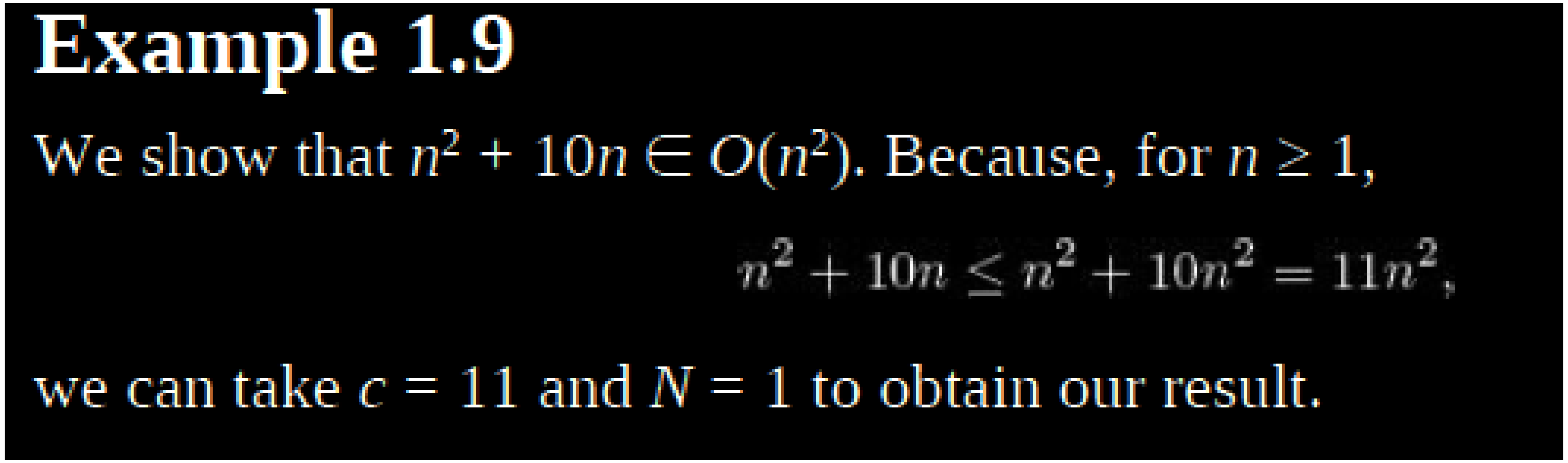
We show that $n^2 + 10n \in O(n^2)$. Because, for $n \geq 1$,

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

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.



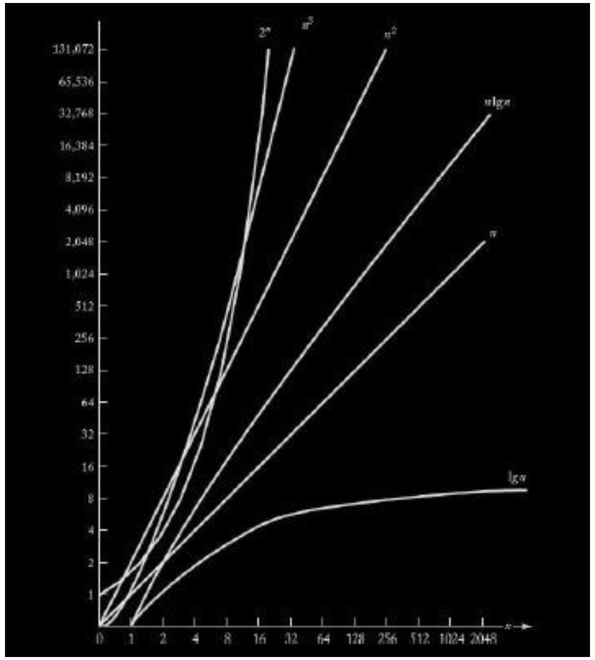
### Example 1.9

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).



# Meaning?

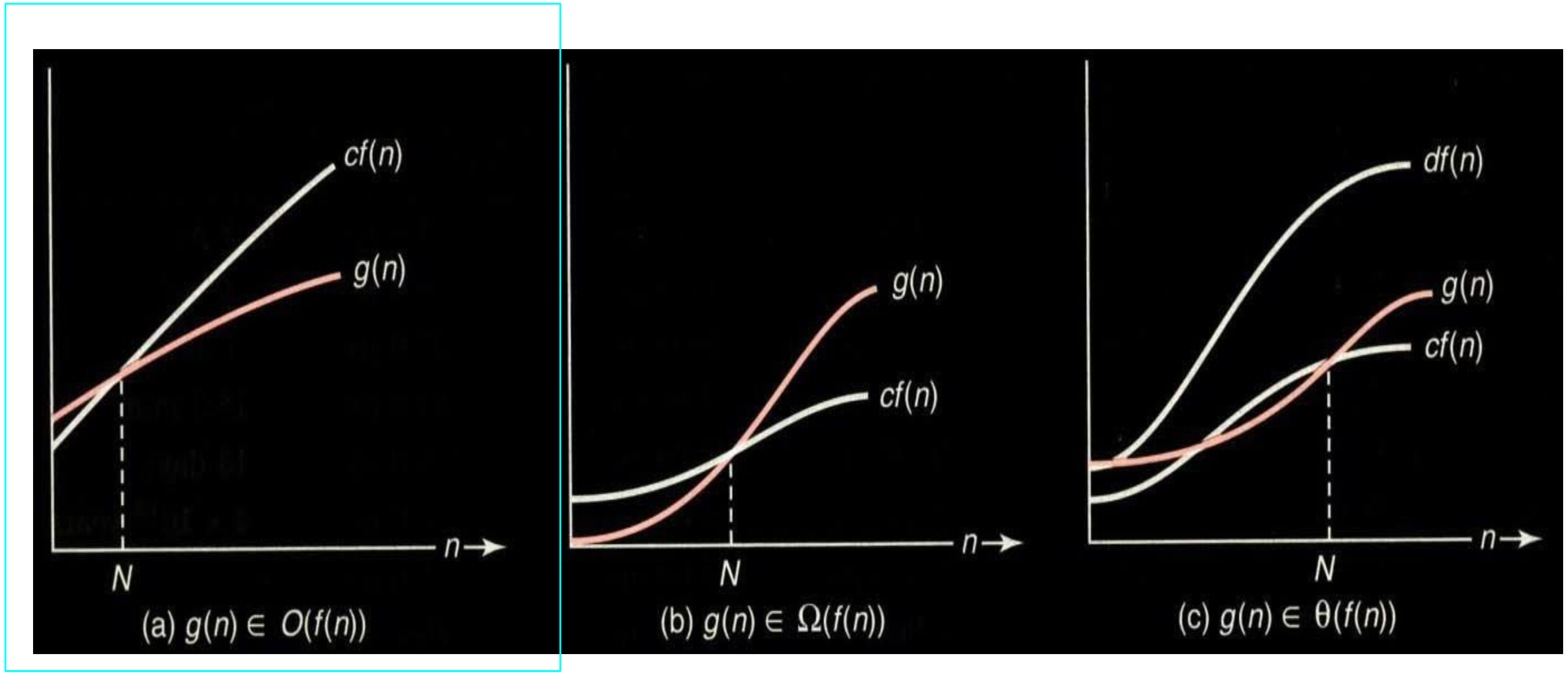
- Big O - what the slowest  and  most space an algo can use
- Omega – what the fastest and  least space an algo can use
- ***    g(n) is **about the behavior** than a mathematical function.

- 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)

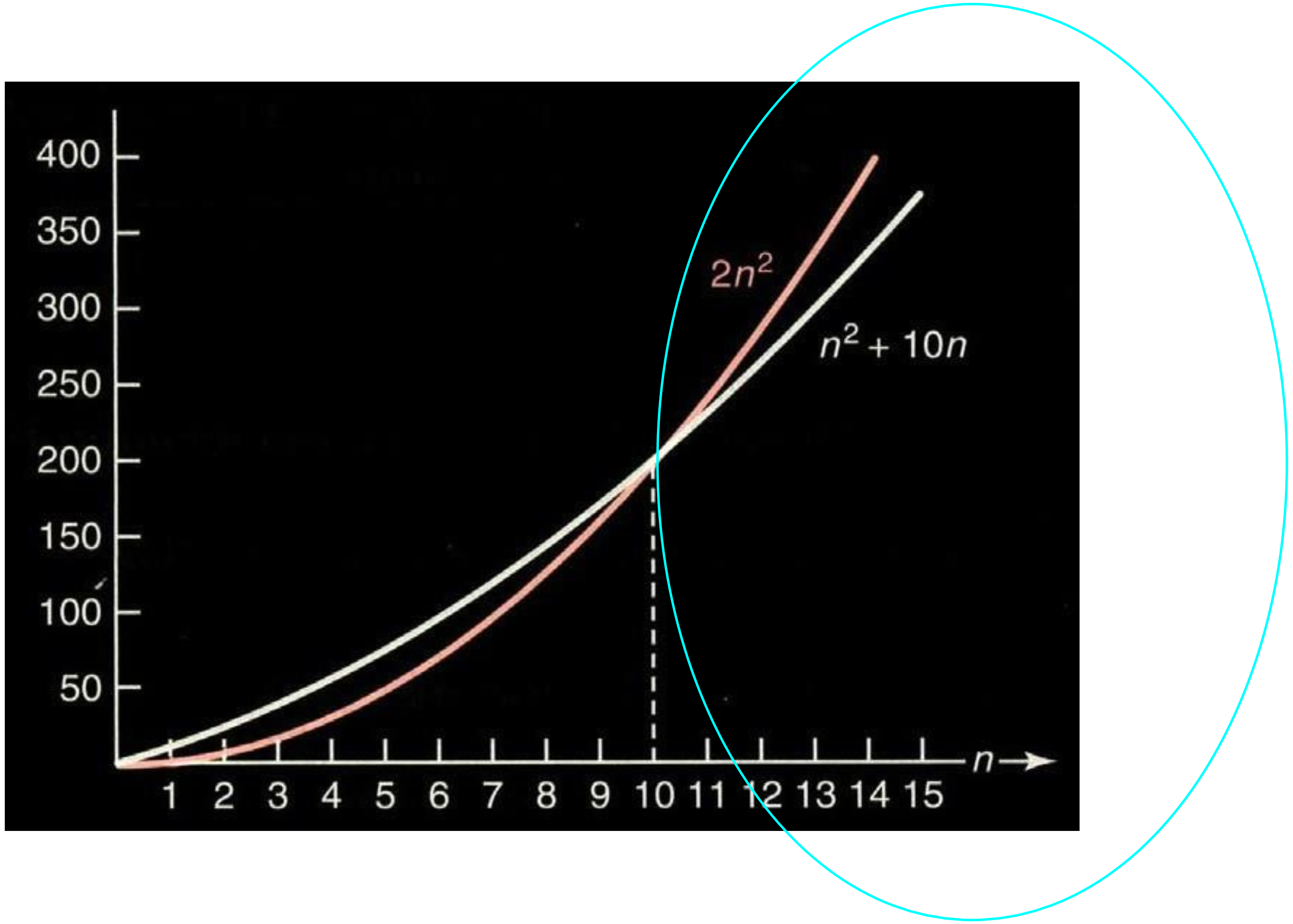


(a) $g(n) \in O(f(n))$ (b) $g(n) \in \Omega(f(n))$ (c) $g(n) \in \theta(f(n))$

# 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

- $5n^2 \in O(n^2)$
  - Take $c = 5$ and $N = 0$, then for all $n$ such that $n \geq N, \quad 5n^2 \leq cn^2$.

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

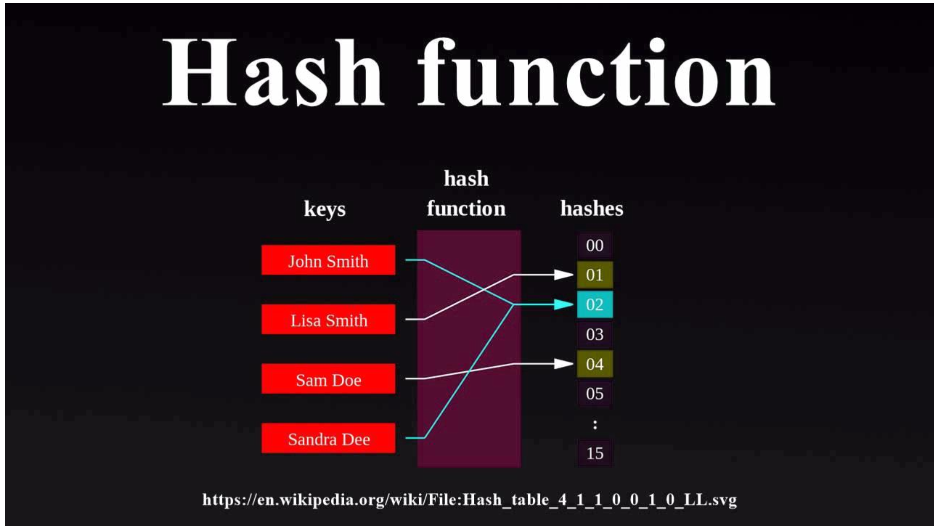
# Big O Notation: More Examples (Cont'd)

- $n \in O(n^2)$
  - 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:
  1. math operations/calculations e.g x = x+1
  2. accessing an array via the index  e.g x = S[i]
  3. accessing a hash via the key
  4. pushing and popping on a stack
  5. insertion and removal from a queue
  6. returning a value from a function



https://en.wikipedia.org/wiki/File:Hash_table_4_1_1_0_0_1_0_LL.svg

Further Reading: Hash Functions and 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

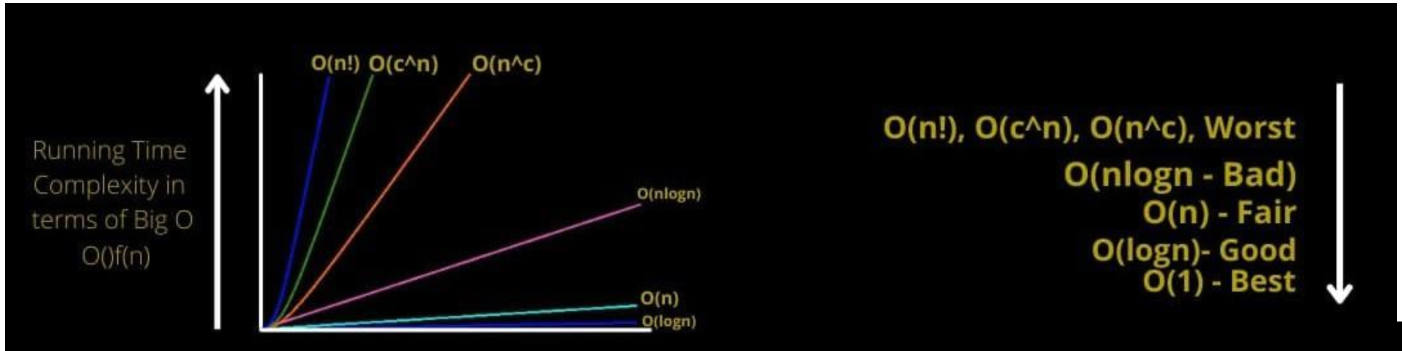
- O(**n**) means that the run-time increases at the same pace as the size of input.
- O($\mathbf{n^2}$) means that the calculation runs in quadratic time, which is the squared size of the input.
- In programming, many of the basic sorting algorithms have a worst-case run time of O($n^2$):
  - e.g Bubble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

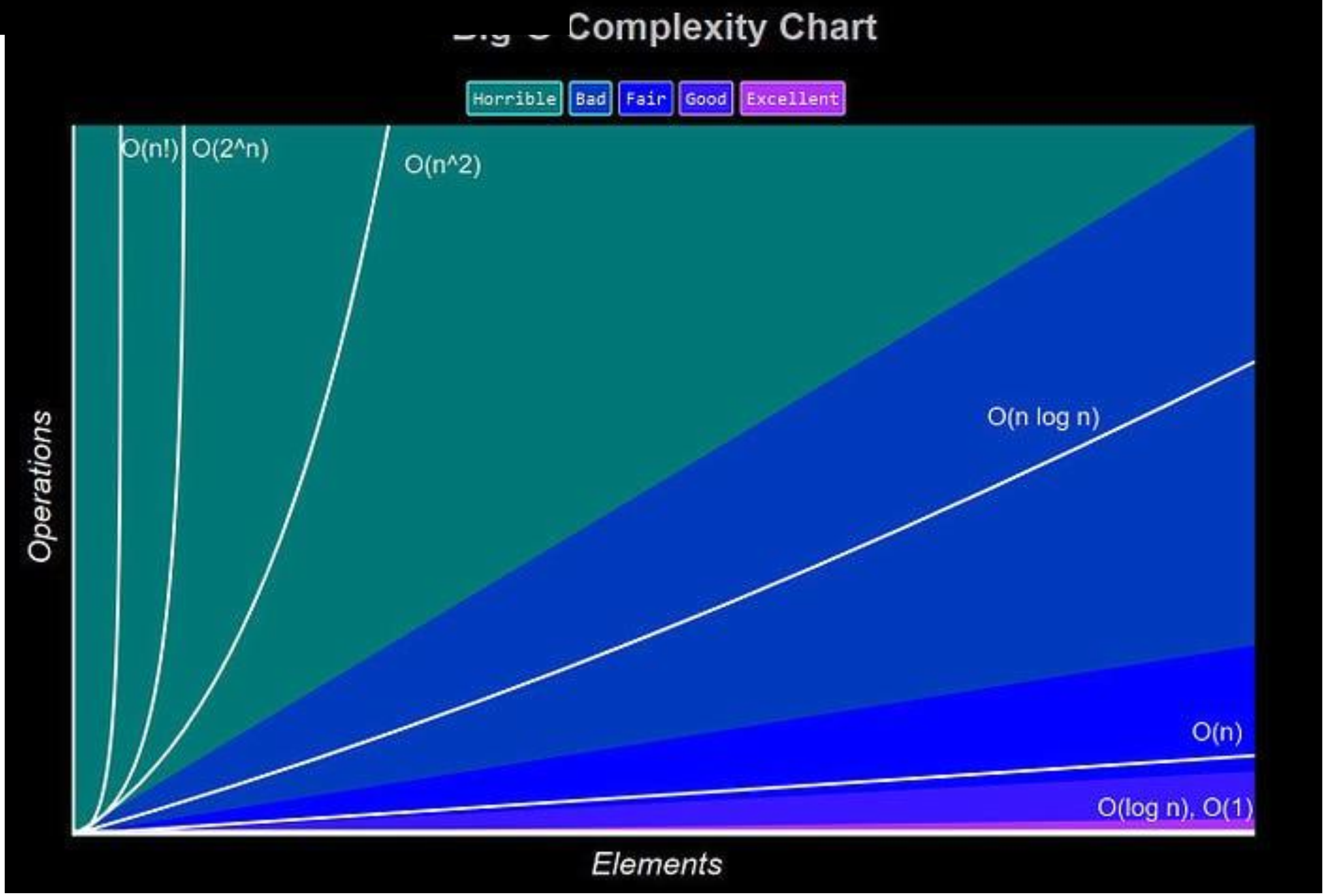
# O(log n) → Logarithmic Time

- 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(n2) and O(n).

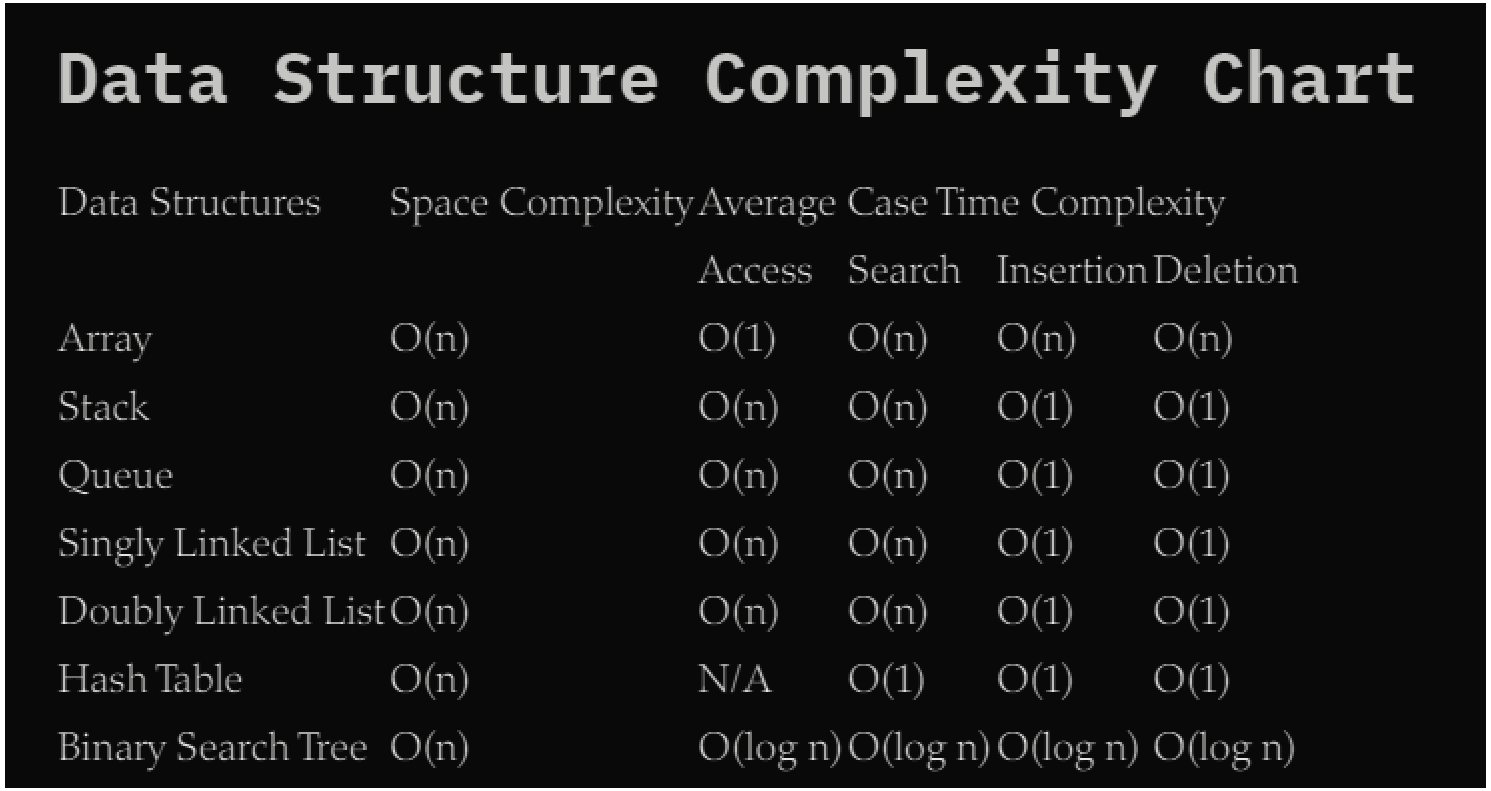
# Factorial O(n!)

- 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’s 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.



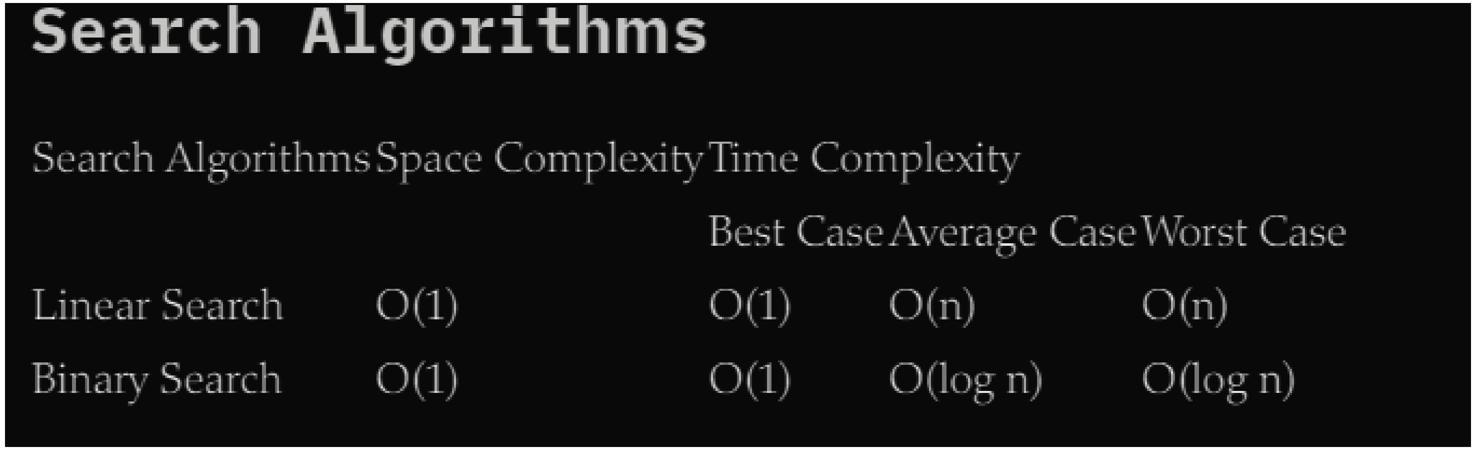


https://betterprogramming.pub/big-o-notation-a-simple-explanation-with-examples-a56347d1daca



## Data Structure Complexity Chart

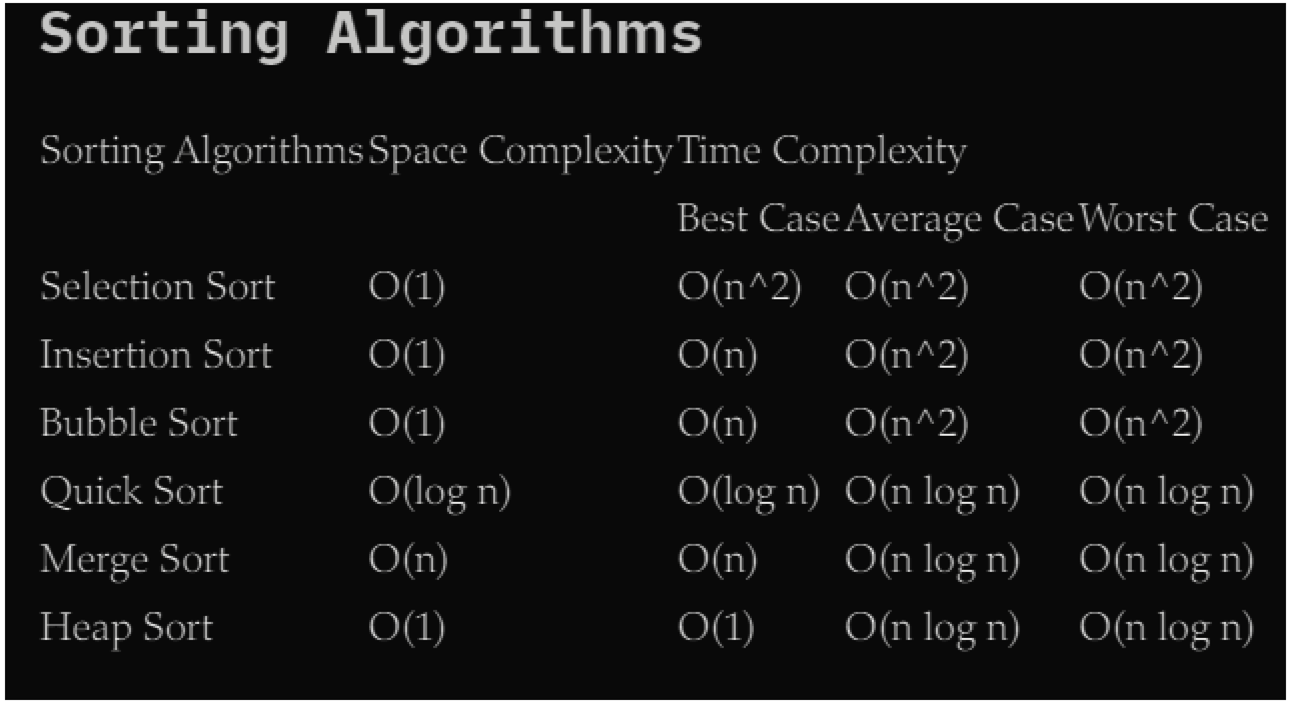
| Data Structures | Space Complexity | Average Case Time Complexity | | | |
|---|---|---|---|---|---|
| | | Access | Search | Insertion | Deletion |
| Array | O(n) | O(1) | O(n) | O(n) | O(n) |
| Stack | O(n) | O(n) | O(n) | O(1) | O(1) |
| Queue | O(n) | O(n) | O(n) | O(1) | O(1) |
| Singly Linked List | 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) |

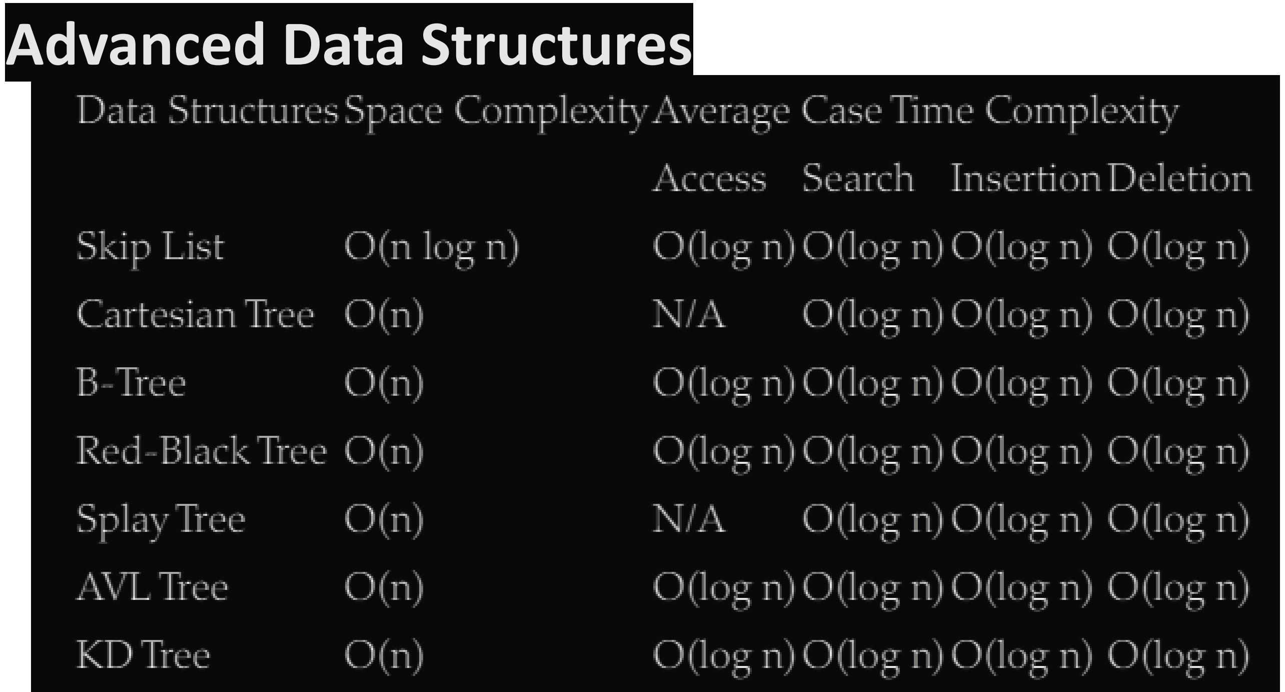
https://flexiple.com/algorithms/big-o-notation-cheat-sheet



## Sorting Algorithms

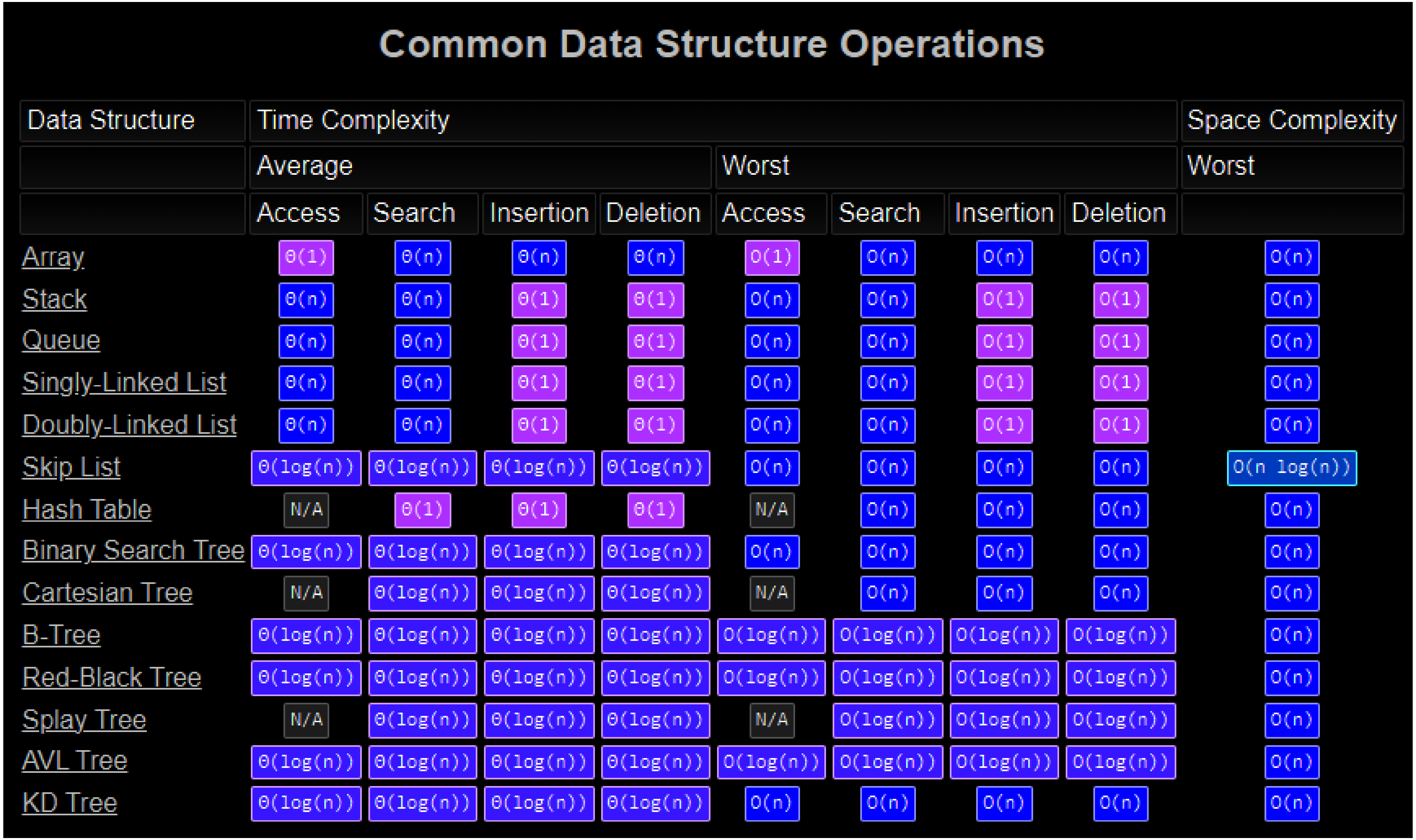
| Sorting Algorithms | Space Complexity | Time Complexity | | |
|---|---|---|---|---|
| | | Best Case | Average Case | Worst Case |
| Selection Sort | O(1) | O(n^2) | O(n^2) | O(n^2) |
| Insertion Sort | O(1) | O(n) | O(n^2) | O(n^2) |
| Bubble Sort | O(1) | O(n) | O(n^2) | O(n^2) |
| Quick Sort | O(log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

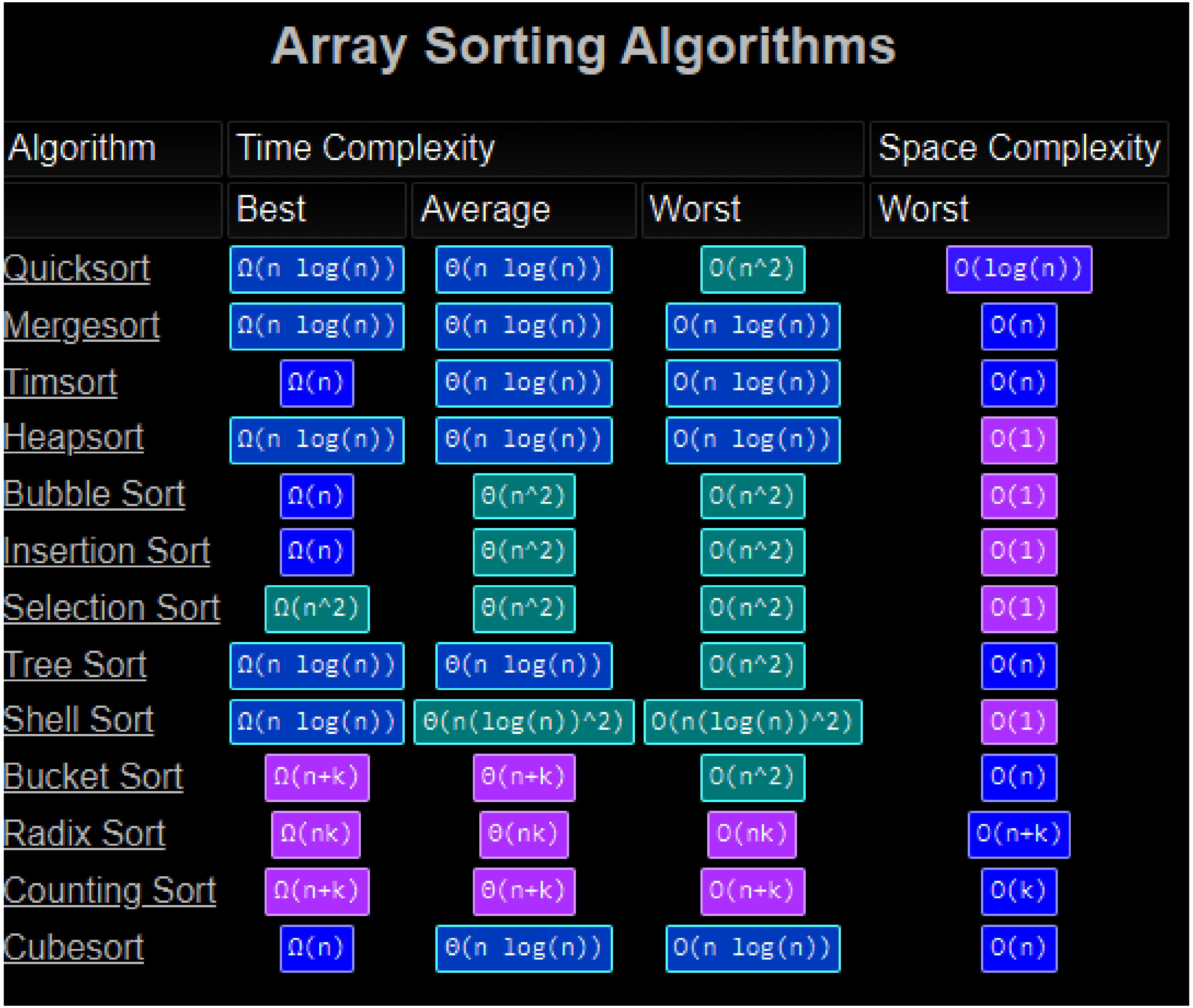
| Data Structures | Space Complexity | Average Case Time Complexity | | | |
|---|---|---|---|---|---|
| | | Access | Search | Insertion | Deletion |
| Skip List | O(n log n) | O(log n) | O(log n) | O(log n) | O(log n) |
| Cartesian Tree | O(n) | N/A | O(log n) | O(log n) | O(log n) |
| B-Tree | O(n)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

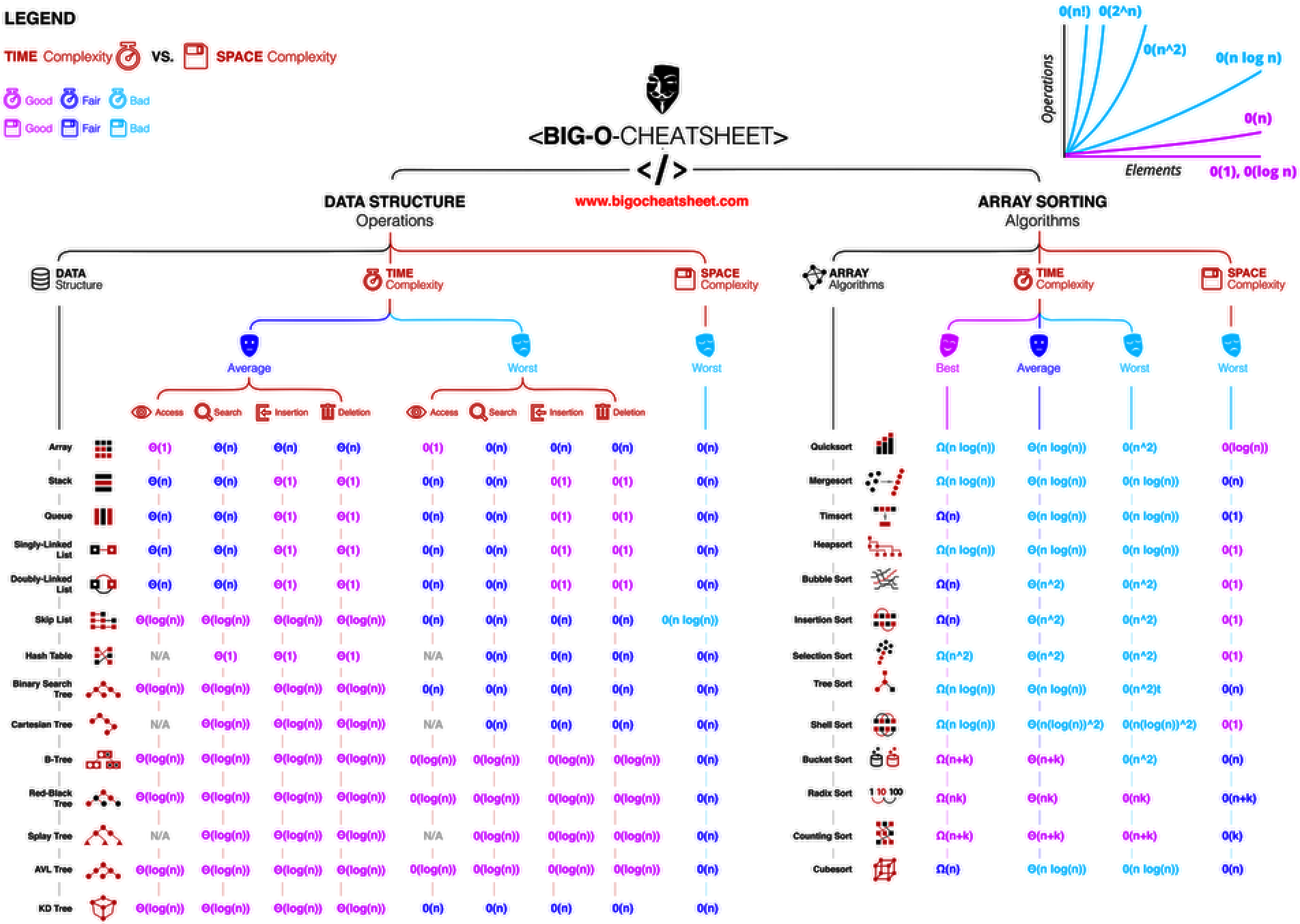
https://www.bigocheatsheet.com/



## 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/



LEGEND

TIME Complexity vs. SPACE Complexity

Good Fair Bad

Good Fair Bad

# <BIG-O-CHEATSHEET>

</>

www.bigocheatsheet.com

0(n!)
0(2^n)
0(n^2)
0(n log n)
0(n)
Operations
Elements
0(1), 0(log n)

## DATA STRUCTURE Operations

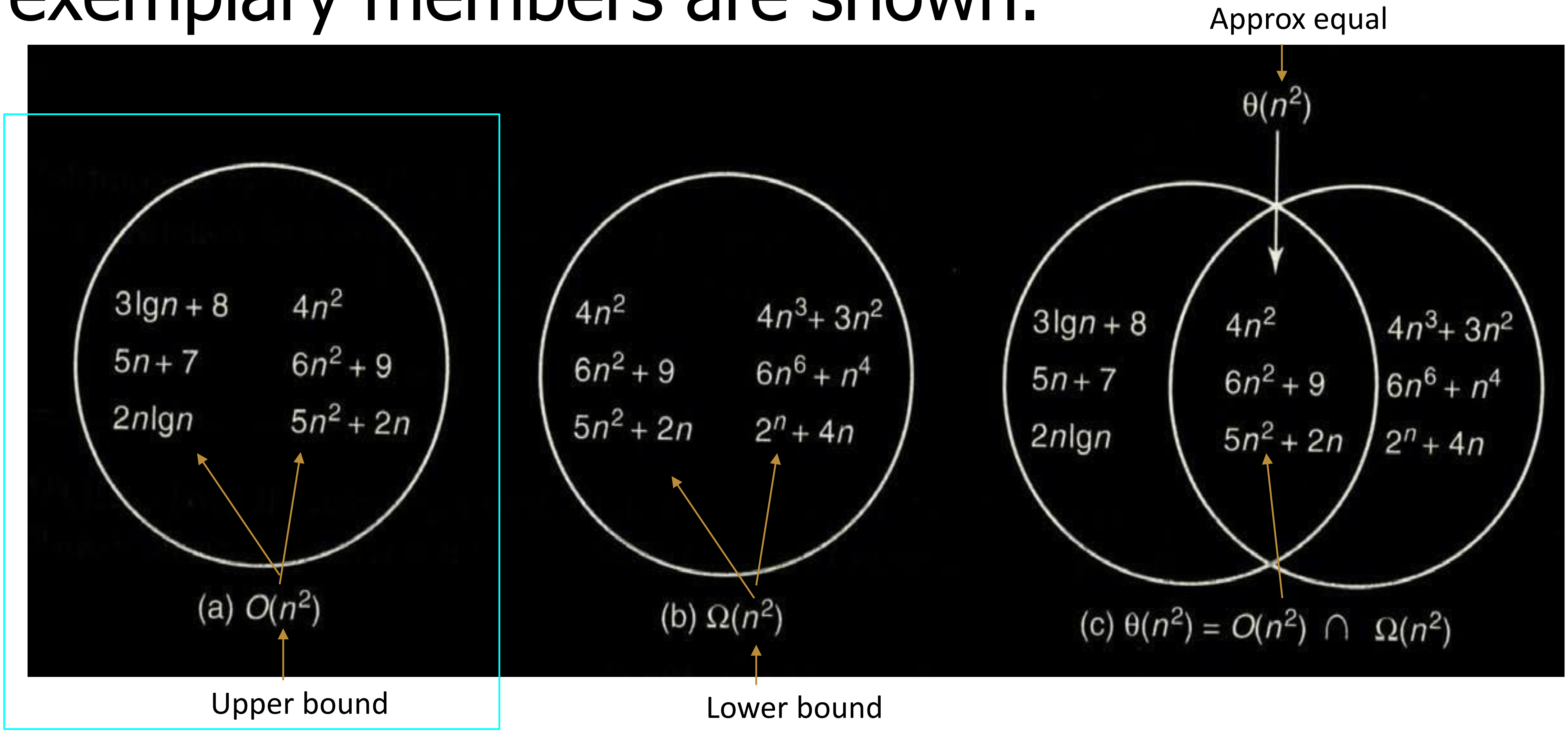
| DATA Structure | TIME Complexity |  |  |  |  |  |  |  | SPACE Complexity |
|---|---|---|---|---|---|---|---|---|---|
|  | Average |  |  |  | Worst |  |  |  | Worst |
|  | Access | Search | Insertion | Deletion | Access | Search | Insertion | Deletion |  |
| Array | Θ(1) | Θ(n) | Θ(n) | Θ(n) | 0(1) | 0(n) | 0(n) | 0(n) | 0(n) |
| Stack | Θ(n) | Θ(n) | Θ(1) | Θ(1) | 0(n) | 0(n) | 0(1) | 0(1) | 0(n) |
| Queue | Θ(n) | Θ(n) | Θ(1) | Θ(1) | 0(n) | 0(n) | 0(1) | 0(1) | 0(n) |
| Singly-Linked List | Θ(n) | Θ(n) | Θ(1) | Θ(1) | 0(n) | 0(n) | 0(1) | 0(1) | 0(n) |
| Doubly-Linked List | Θ(n) | Θ(n) | Θ(1) | Θ(1) | 0(n) | 0(n) | 0(1) | 0(1) | 0(n) |
| Skip List | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | 0(n) | 0(n) | 0(n) | 0(n) | 0(n log(n)) |
| Hash Table | N/A | Θ(1) | Θ(1) | Θ(1) | N/A | 0(n) | 0(n) | 0(n) | 0(n) |
| Binary Search Tree | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | 0(n) | 0(n) | 0(n) | 0(n) | 0(n) |
| Cartesian Tree | N/A | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | N/A | 0(n) | 0(n) | 0(n) | 0(n) |
| B-Tree | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | 0(log(n)) | 0(log(n)) | 0(log(n)) | 0(log(n)) | 0(n) |
| Red-Black Tree | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | 0(log(n)) | 0(log(n)) | 0(log(n)) | 0(log(n)) | 0(n) |
| Splay Tree | N/A | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | N/A | 0(log(n)) | 0(log(n)) | 0(log(n)) | 0(n) |
| AVL Tree | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | 0(log(n)) | 0(log(n)) | 0(log(n)) | 0(log(n)) | 0(n) |
| KD Tree | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | 0(n) | 0(n) | 0(n) | 0(n) | 0(n) |

## ARRAY SORTING Algorithms

| ARRAY Algorithms | TIME Complexity |  |  | SPACE Complexity |
|---|---|---|---|---|
|  | Best | Average | Worst | Worst |
| Quicksort | Ω(n log(n)) | Θ(n log(n)) | 0(n^2) | 0(log(n)) |
| Mergesort | Ω(n log(n)) | Θ(n log(n)) | 0(n log(n)) | 0(n) |
| Timsort | Ω(n) | Θ(n log(n)) | 0(n log(n)) | 0(1) |
| Heapsort | Ω(n log(n)) | Θ(n log(n)) | 0(n log(n)) | 0(1) |
| Bubble Sort | Ω(n) | Θ(n^2) | 0(n^2) | 0(1) |
| Insertion Sort | Ω(n) | Θ(n^2) | 0(n^2) | 0(1) |
| Selection Sort | Ω(n^2) | Θ(n^2) | 0(n^2) | 0(1) |
| Tree Sort | Ω(n log(n)) | Θ(n log(n)) | 0(n^2)t | 0(n) |
| Shell Sort | Ω(n log(n)) | Θ(n(log(n))^2) | 0(n(log(n))^2) | 0(1) |
| Bucket Sort | Ω(n+k) | Θ(n+k) | 0(n^2) | 0(n) |
| Radix Sort | Ω(nk) | Θ(nk) | 0(nk) | 0(n+k) |
| Counting Sort | Ω(n+k) | Θ(n+k) | 0(n+k) | 0(k) |
| Cubesort | Ω(n) | Θ(n log(n)) | 0(n log(n)) | 0(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

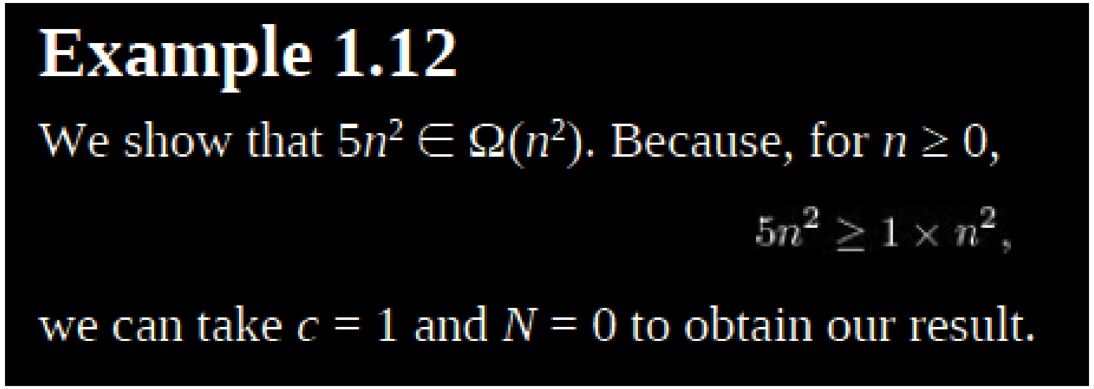
- Definition: (Asymptotic Lower Bound)
  - For a given complexity function $f(n)$, $\Omega(f(n))$ is the set of complexity functions $g(n)$ for which there exists some positive real constant $c$ and some non-negative integer $N$ such that for all $n \geq N$,

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

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

# Ω

- 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.

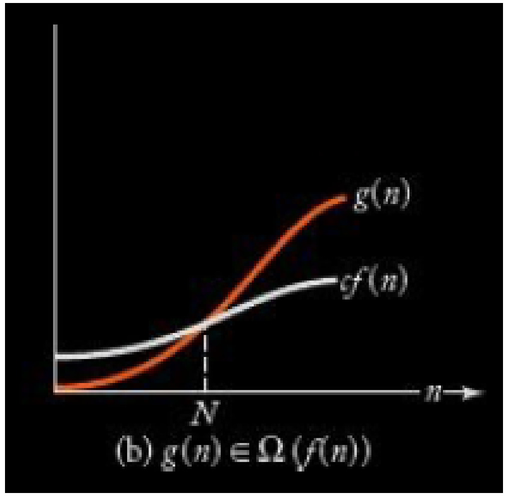


**Example 1.12**

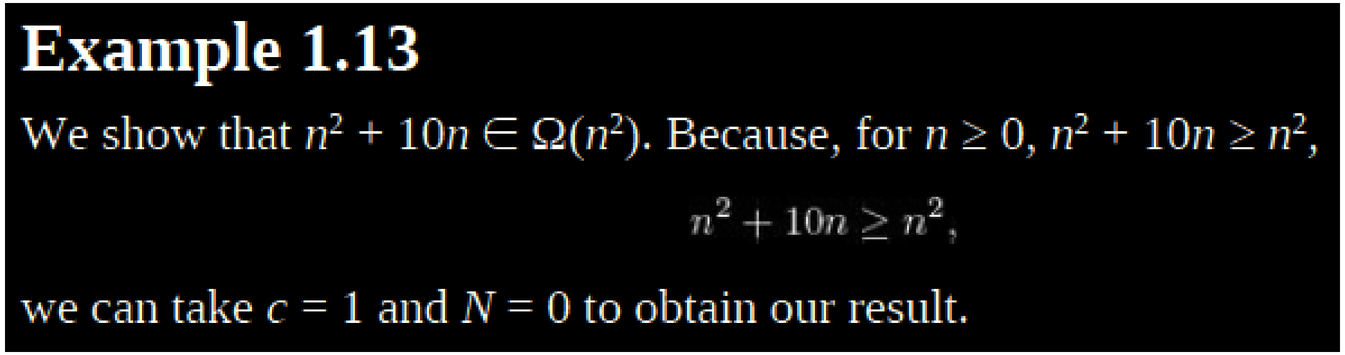
We show that $5n^2 \in \Omega(n^2)$. Because, for $n \geq 0$,

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

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



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

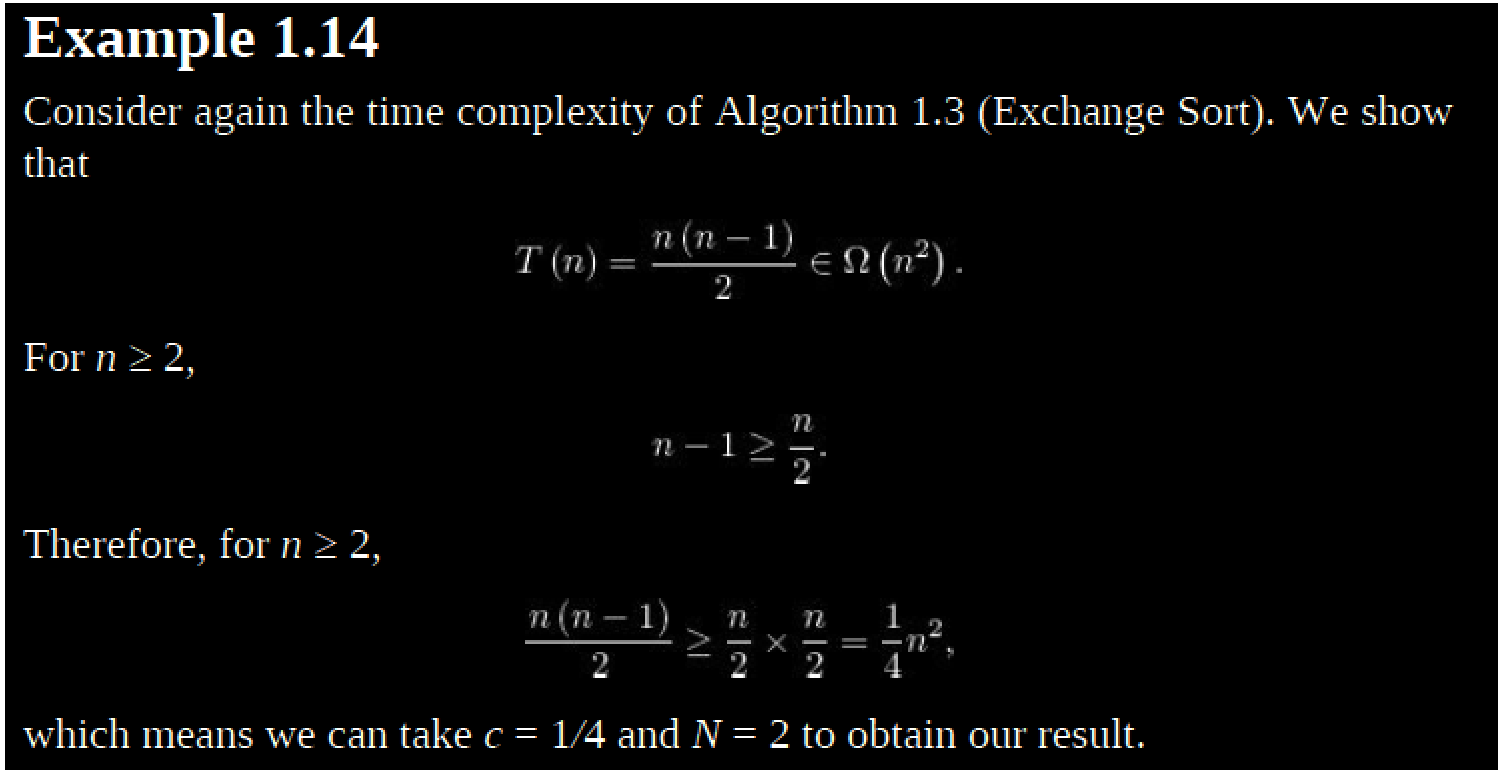


**Example 1.13**

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.



## 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 $n \geq 2$,

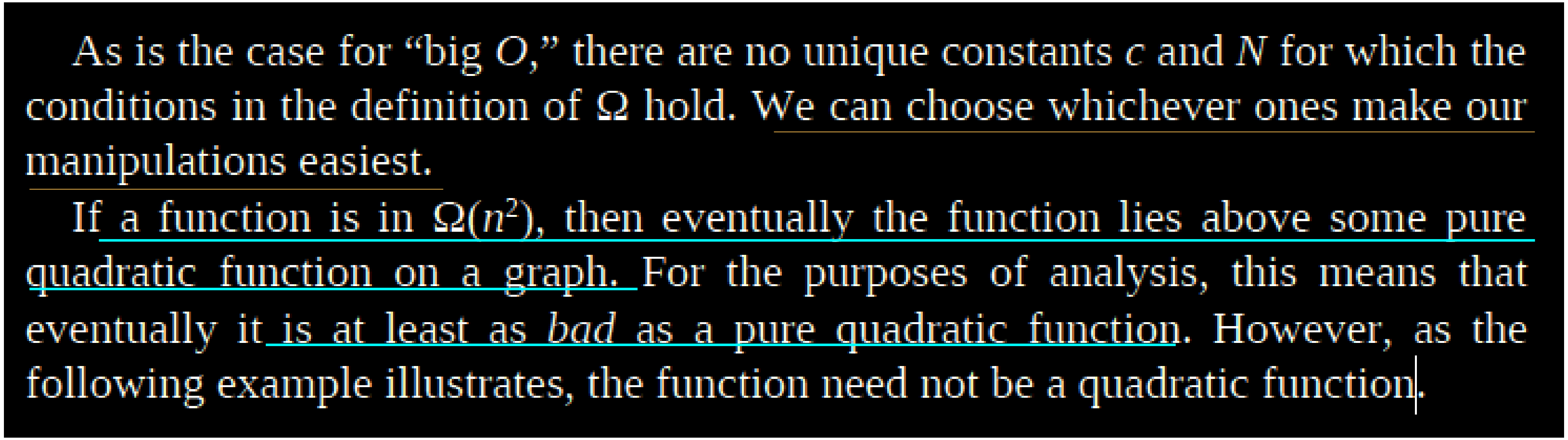
$$n - 1 \geq \frac{n}{2}.$$

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.

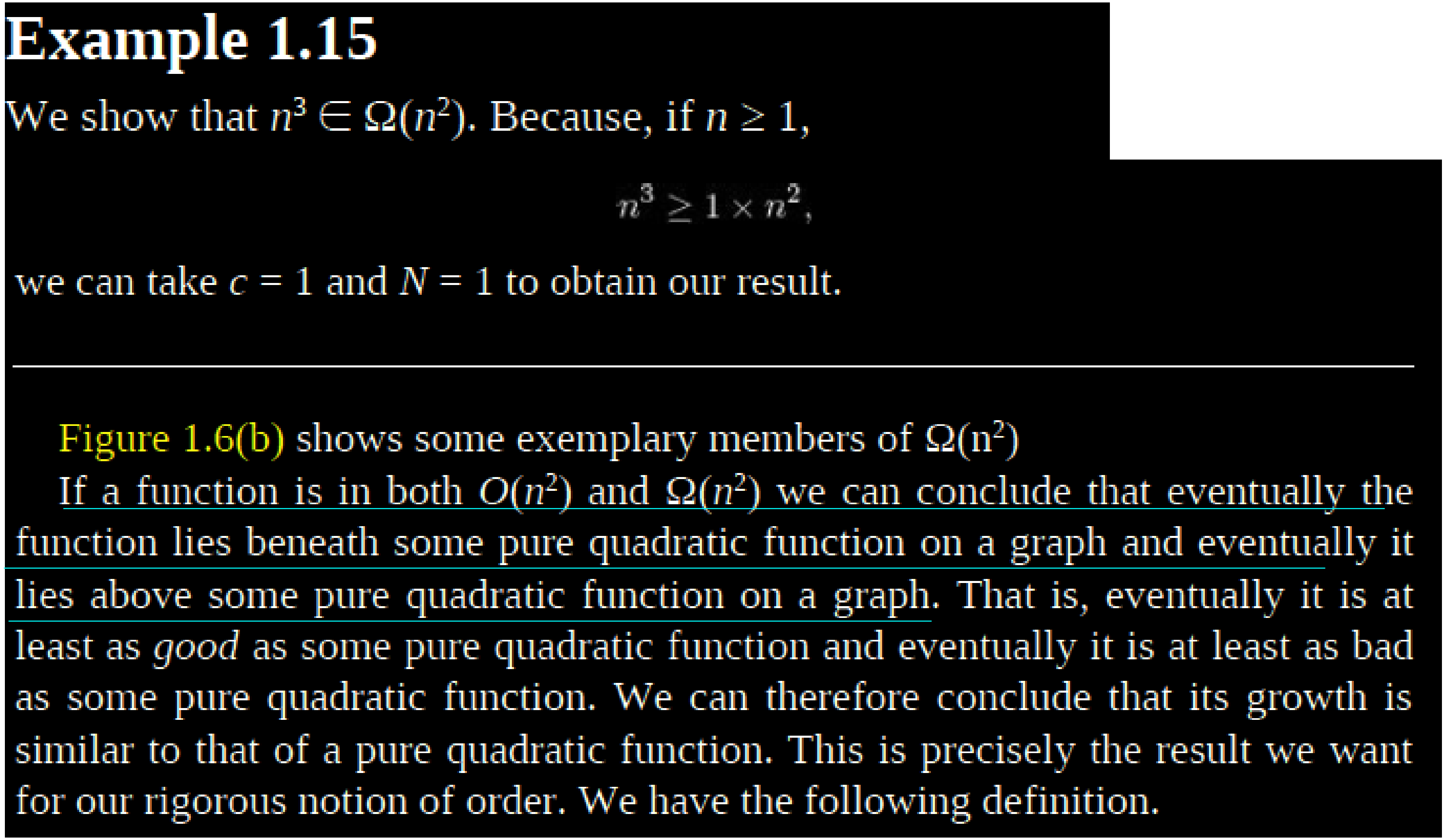
Ω



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



## Example 1.15

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

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

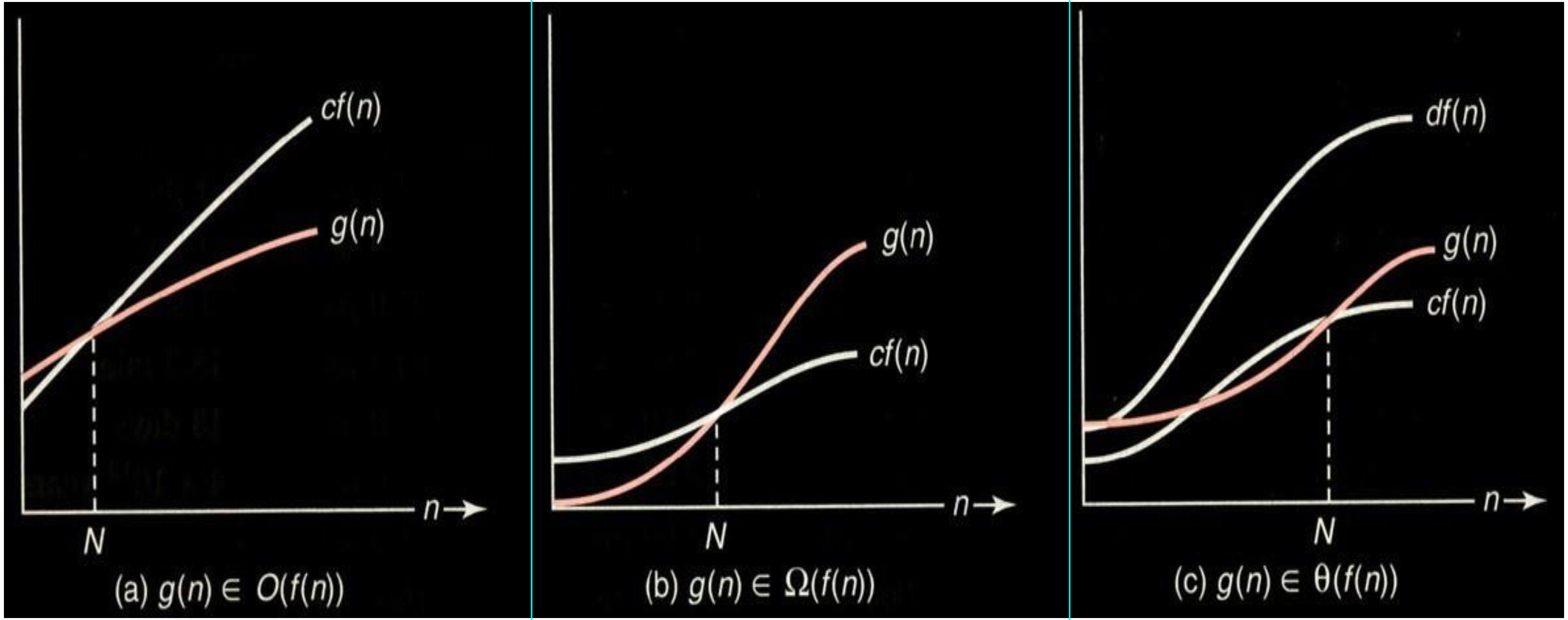
we can take $c = 1$ and $N = 1$ to obtain our result.

---

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**”, **Ω**, and **Θ**



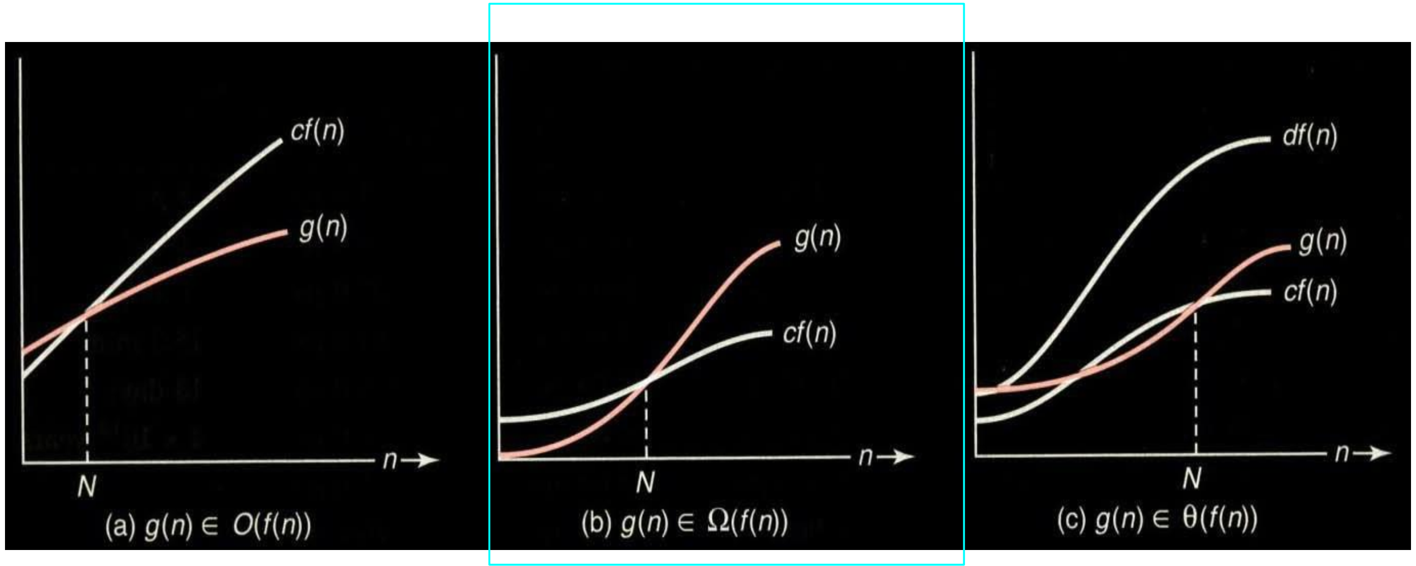
(a) $g(n) \in O(f(n))$

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

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

# Ω 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)



# $\Omega$ 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 Notation: More Examples

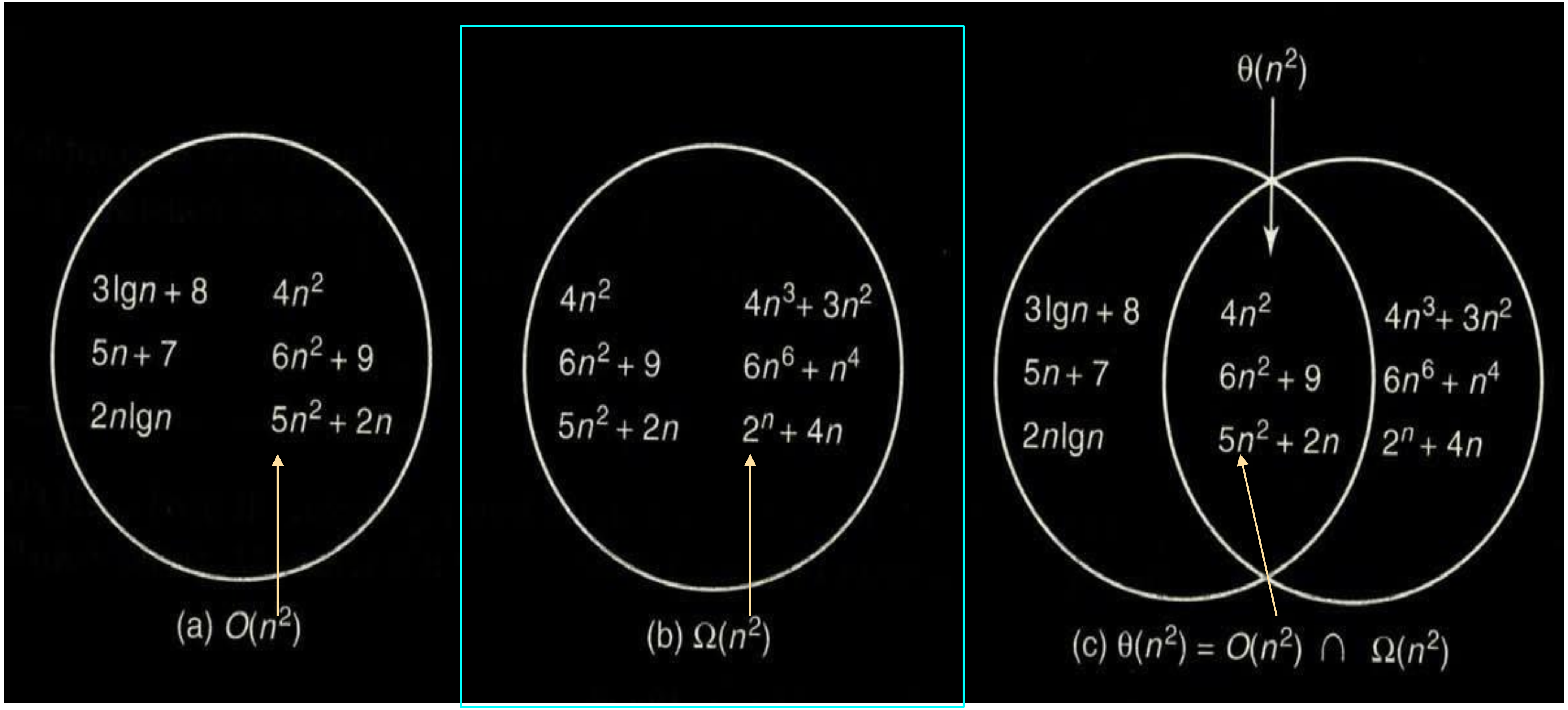
- $T(n) = \frac{n(n-1)}{2}$
  - Because, **for** $\boldsymbol{n \geq 2}$**,** $\boldsymbol{n - 1 \geq n/2}$, so it holds that $\frac{n(n-1)}{2} \geq \frac{n}{2} \times \frac{n}{2} = \frac{1}{4} n^2$
  - Therefore, we can take *c = 1/4* and *N = 2*, to conclude that $T(n) \in \Omega(n^2)$.
- $n^3 \in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

# Ω Notation: Last 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's divide both sides by $cn$.
  - Then, we will get **$1/c \geq n$**, which is **impossible**.
  - Therefore, $n$ does not belong to $\Omega(n^2)$ .

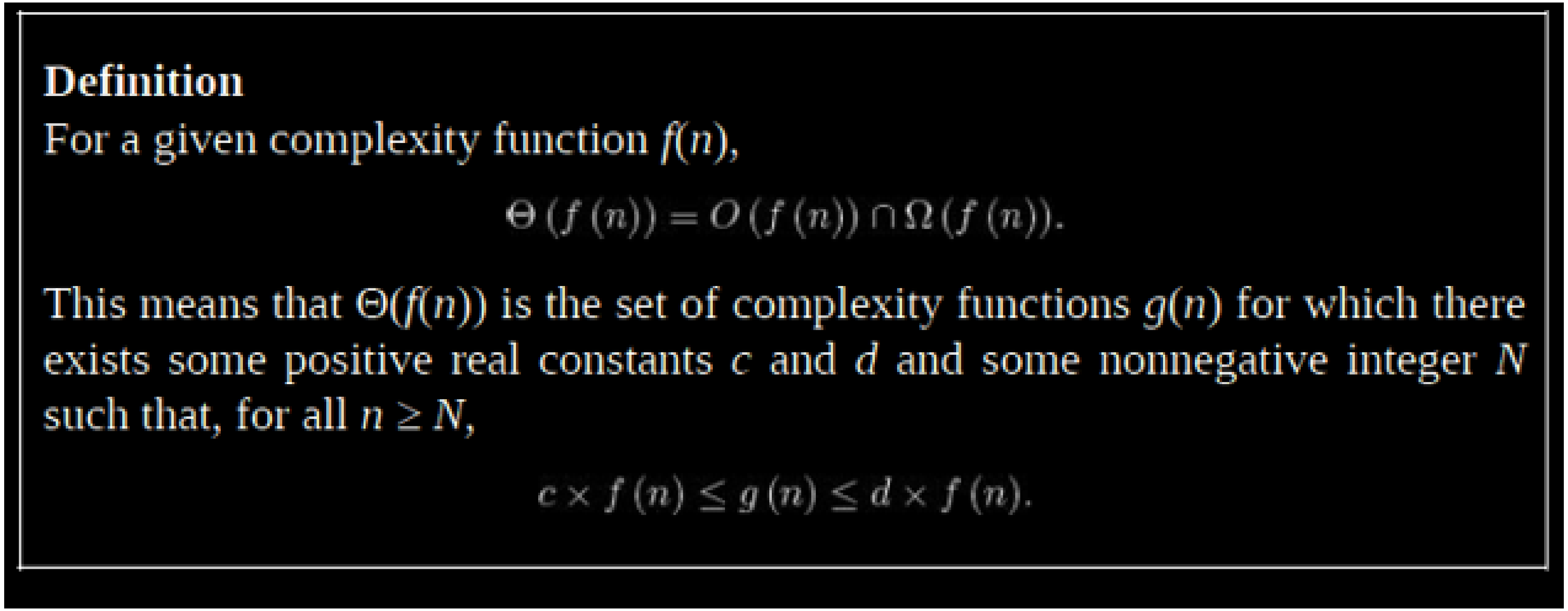
# Figure 1.6 The sets $O(n^2)$, $\Omega(n^2)$, $\Theta(n^2)$. Some exemplary members are shown.

Approx equal



Upper bound

Lower bound



**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 $n \geq N$,

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

## 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

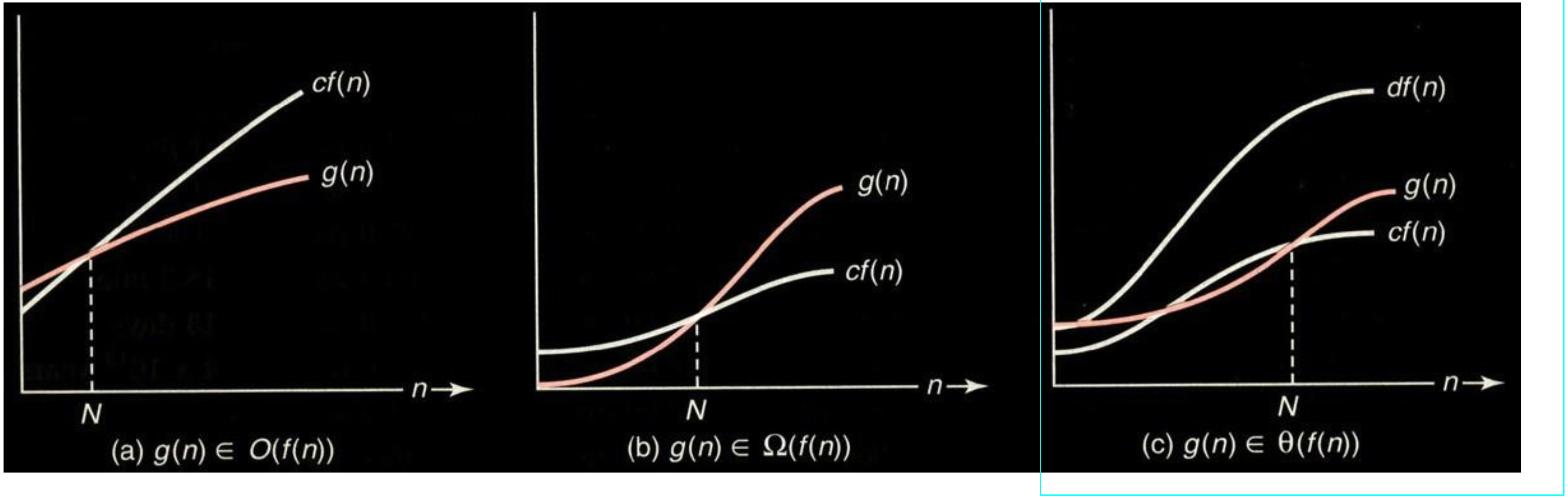
- Definition: (Asymptotic Tight Bound)
  - For a given complexity function $f(n)$, $\Theta(f(n))$ is the set of complexity functions $g(n)$ for which there exists some positive real constants $c$ and $d$ and some non-negative integer $N$ such that for all $n \geq N$,

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

  - $g(n) \in \Theta(f(n))$, 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

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



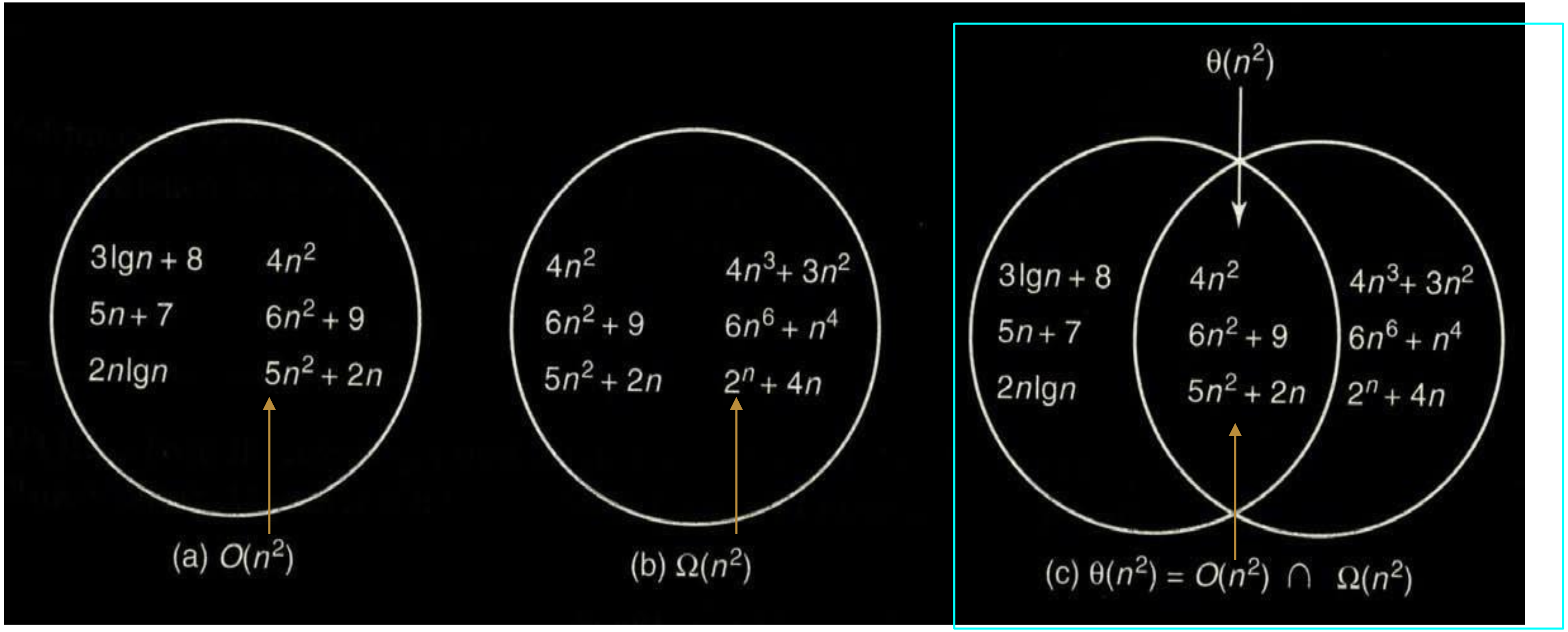
(a) $g(n) \in O(f(n))$

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

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

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

Approx equal



Upper bound

Lower bound

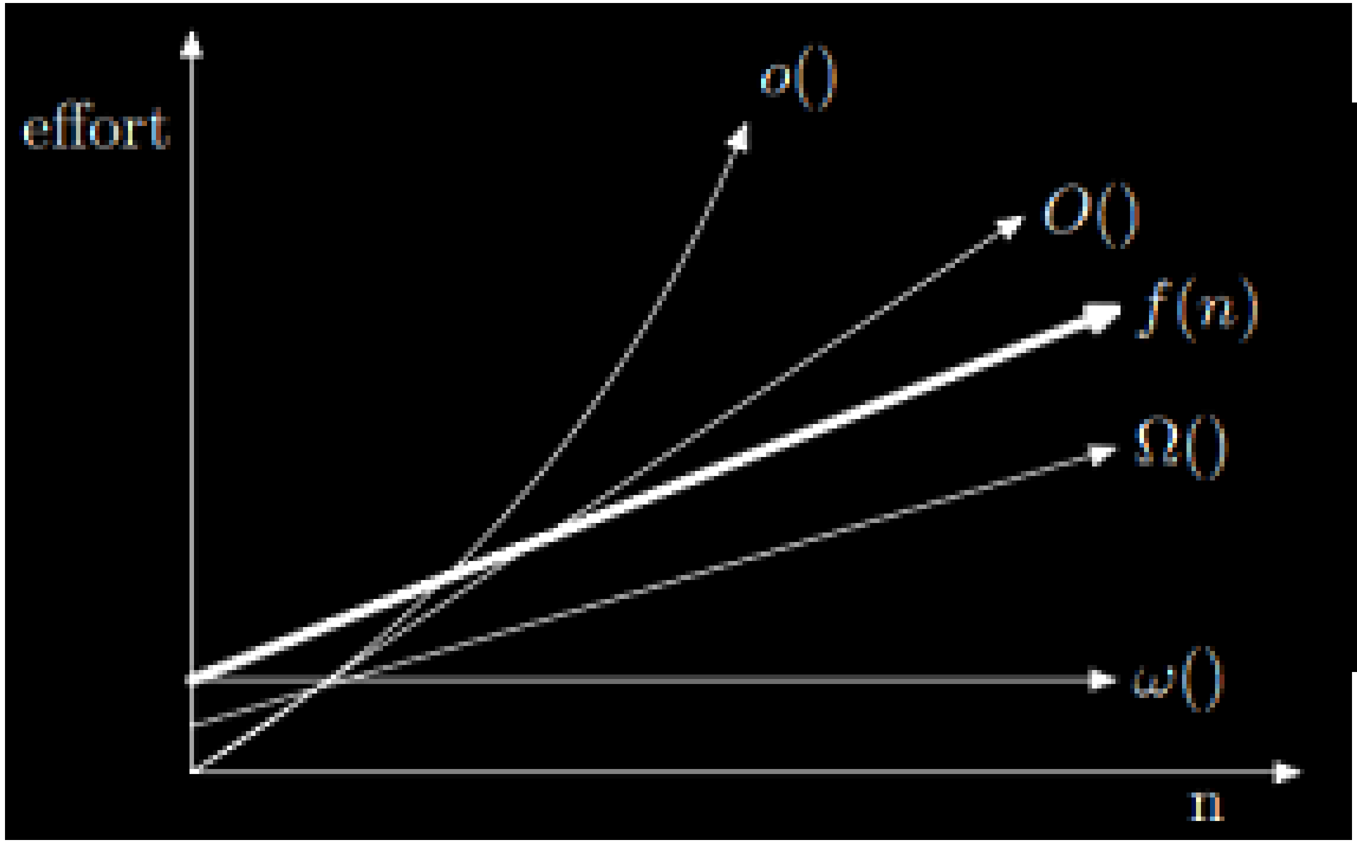
# Rigorous Definition to Order: Small *o*

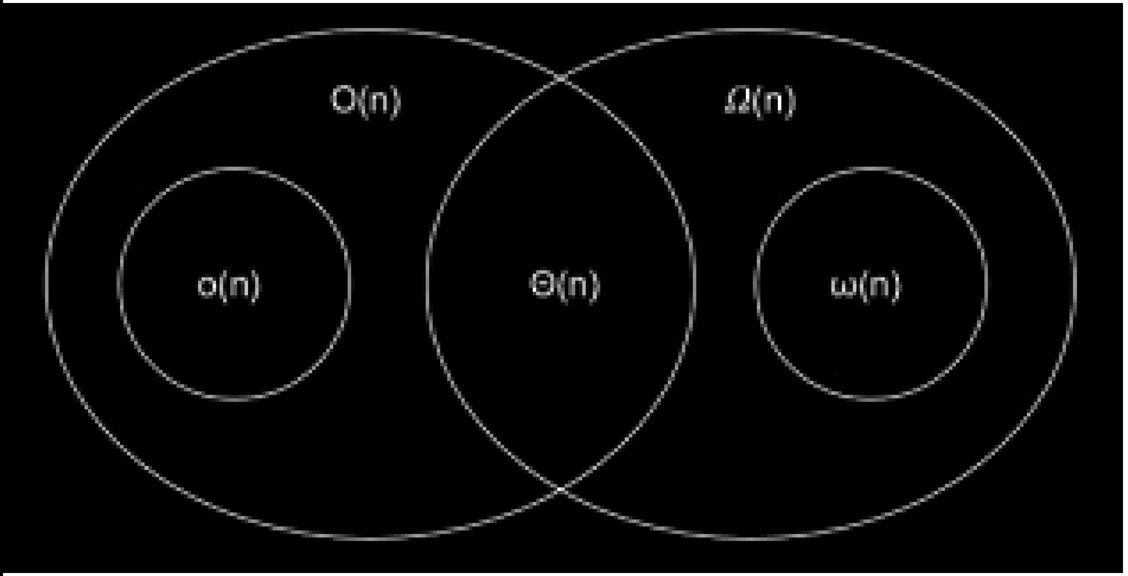
- Definition:
  - For a given complexity function $f(n)$, $o(f(n))$ is the set of complexity functions $g(n)$ satisfying the following: For ***every*** positive real constant $c$ there exists a non-negative integer $N$ such that for all $n \geq N$,

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

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

# Big O vs Small o





# 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 isn't asymptotically tight

# Small *o* Notation: Example

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

# 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 $n \geq N$, $$n \leq \frac{1}{6} \times 5n = \frac{5}{6}n$$
  - 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 look at 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-linea**r: 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/

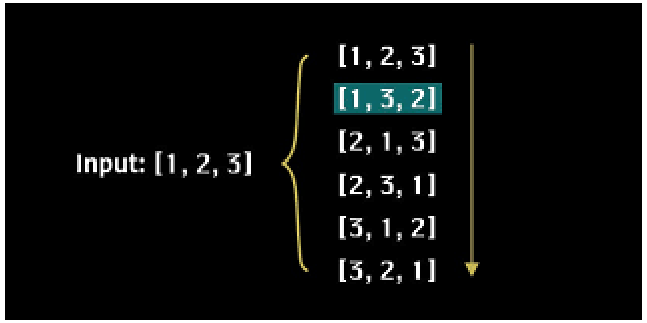
# Summary ctd..

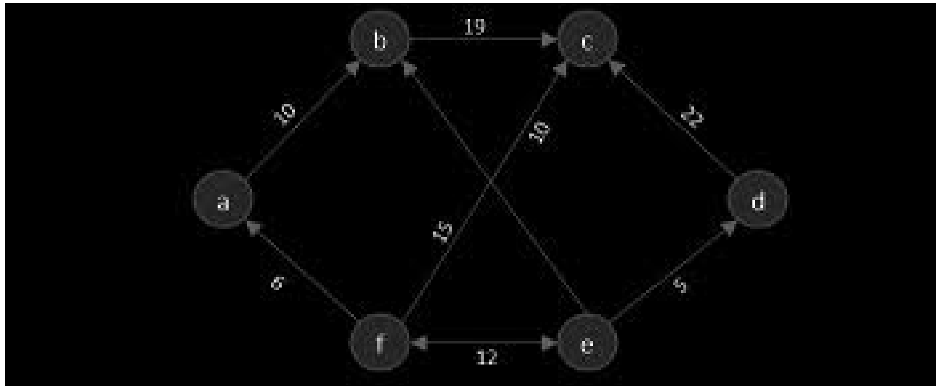
8. $O(N^2)$ – **Quadratic**: The Bubble Sort, Insertion Sort, and Select Sort are some of the algorithms that have the quadratic complexity. If there are two nested loops traversing the whole array each time, then we have $O(n^2)$ of time complexity.
9. $O(N^3)$ – **Cubic**: When we have 3 nested loops 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's a password that supports only numbers 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?