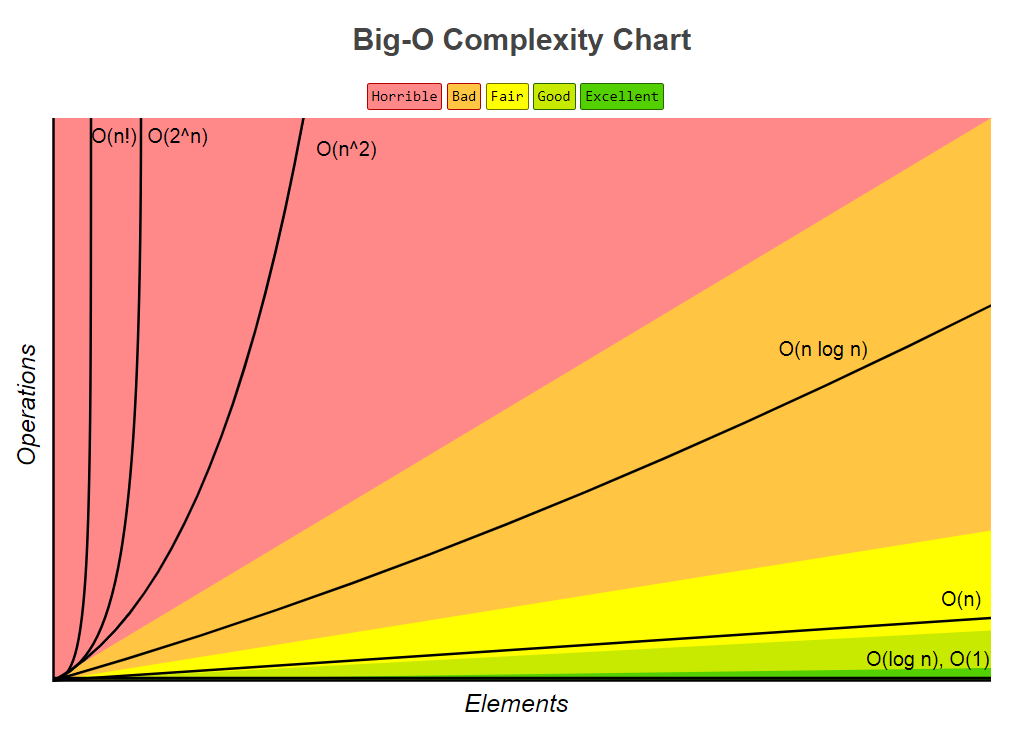
Big-O Analysis

*Big-O analysis* is a form of metric used to describe the ***efficiency/performance***of an ***algorithm*** in terms of ***time & space/memory*** it takes to run for a given input size.

* *Big-O analysis* doesn’t provide formal benchmark i.e. actual time & space required to execute the algorithm and execution speed of the algorithm.
* *algorithm* examples – insert(), get(), delete() API’s of Java ArrayList class
* *Big-O analysis* compares the predicted relative performance of different algorithms
* *Big-O analysis* provides a simple way to classify algorithms by relative efficiency when dealing with large input sizes.
* *Performance* of *any* algorithm is depends upon various factors like data size, data type (sorted/unsorted etc.) etc. hence it’s not possible define the performance of an algorithm with just one case/scenario*. Big-O analysis* describes/provides the efficiency/performance of an algorithm in [possibly three conditions/scenarios](https://en.wikipedia.org/wiki/Best,_worst_and_average_case),
  + **Best Case** – *at least*
  + **Expected/Average Case** -*on average*
  + **Worst Case** – *at most*
* In most cases**, Average Case and Worst-Case** scenarios considered; though sometime **Best-Case** scenario also considered for measuring performance
* For example, any car manufacturer provides the fuel efficiency/performance of the given car in three scenarios based on driving condition, condition of the tire, drivers experience etc.
  + **Best Case** (Very good driving condition – road, weather, tire; driver; no. of stops/brakes) – *at least 30 miles/gallon*
  + **Expected/Average Case** -*on average 25 miles/gallon*
  + **Worst Case** – *at most 18 miles/gallon*

Which Algorithm Is Better?



*Big-O analysis* classifies the “Runtime” of algorithm as follows from best-to-worse performance:

* + **Constant Time** **O (1)** – Runtime is*constant* for any given input size. The fastest possible running time for any runtime analysis is constant O (1). This is the ideal runtime for an algorithm, but it’s rarely achievable.
  + **Logarithmic Time** **O (log n)** – Runtimeincreases/decrease *logarithmically* in proportion to the input size
    - Any operation reduces by half repeatedly. E.g. Binary Search
  + **Linear Time** **O(n)** – Runtimeincreases/decrease *linearly*  in proportion to the input size
  + **Super Linear Time O(n log n)** – Midway between a *linear time* and *polynomial time*
  + **Polynomial Time O(n power of c)** – Runtime grows *quickly* in proportion to the input size
  + **Amortized Time** - <https://stackoverflow.com/questions/15079327/amortized-complexity-in-laymans-terms> – Combination of constant time and linear time.
    - E.g. ArrayList->insert() operation takes O(1) in most cases and when array is full it takes O(n) for copying elements to new array so average of O(1) + O(n) = Amortized Time
  + **Exponential Time O(c power of n)** – Runtime grows even faster than a *Polynomial time*
  + **Factorial Time O (n!)** – Run time grows fastest and becomes quickly unusable for even small values of *n*
  + Examples for input size n = 10,
    - 1
    - Log 10 = 1 🡪 Is always best but difficult to achieve
    - 10 = 10
    - 10 Log 10 = 10
    - 10 power of 2 = 100
    - 2 power of 10 = 1024
    - 10! = 3,628,800
* Few notes,
  + Recursive function calls always takes O(n)
  + If input size *n* is small then runtime for any algorithms will be same

How to Do *Big-O Analysis*?

1. Figure out what the *input* is and what *n* represents
2. Identify the number of operations algorithm performs on each item of *input* in terms of *n*
3. Eliminate all but the *highest-order* terms
4. Remove all the *constant* factors like O (n+2), O (n/2) etc.
5. Identify the *Best Case, Worst Case & Average Case* based on *arrangement* of input values for any input size *n*

Memory Footprint Analysis or Space Complexity

*Memory footprint* provides analysis about usage of memory by algorithm for the given input and input size.

1. Identify the amount of memory required by the algorithm for each item of *input* in terms of *n*
2. There is usually tradeoff between optimal memory use and runtime performance

Big-O Analysis (Time & Space Complexities) on Common Data Structures and Algorithms

