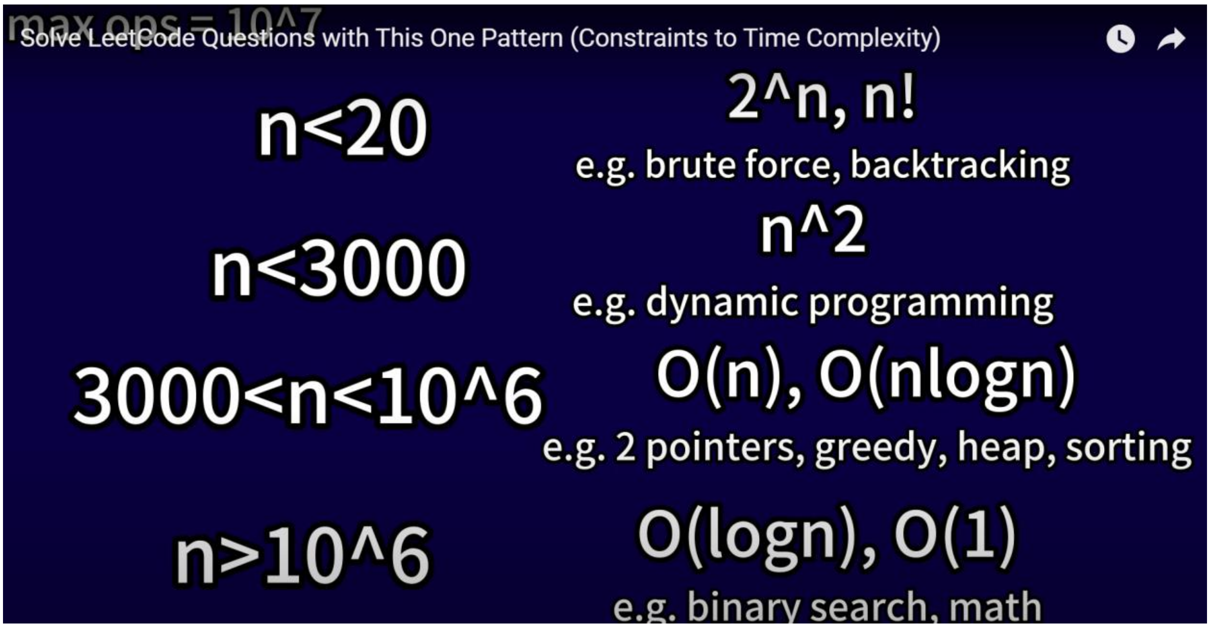
Constraints	**Worst time complexity**	**Algorithmic solution**	**Examples**
***n* ≤ 12**	O(n!)	Recursion & Backtracking	Permutation 1….n
***n* ≤ 25**	O(2n)	Recursion , Backtracking & Bit Manipulation	All subsets of an array of size n
***n* ≤ 100**	O(n4)	Dynamic Programming	4Sum
***n* ≤ 500**	O(n3)	Dynamic Programming	All triangles with side length less than n
***n* ≤ 104**	O(n2)	Dynamic Programming, Graphs, Trees	Bubble Sort (Slow comparison-based sorting)
***n* ≤ 106**	O(n log n)	Sorting, Binary Search, Divide and Conquer	Merge Sort (Fast comparison-based sorting)
***n* ≤ 108**	O(n)	Mathematical, Greedy	Min and max of element
***n* > 108**	O(log n) or O(1)	Mathematical, Greedy	Binary Search



Common time complexities

Let *n* be the main variable in the problem.

- If *n* ≤ 12, the time complexity can be O(n!).
- If *n* ≤ 25, the time complexity can be O(2^n).
- If *n* ≤ 100, the time complexity can be O(n^4).
- If *n* ≤ 500, the time complexity can be O(n^3).
- If *n* ≤ 10^4, the time complexity can be O(n^2).
- If *n* ≤ 10^6, the time complexity can be O(n log n).
- If *n* ≤ 10^8, the time complexity can be O(n).
- If *n* > 10^8, the time complexity can be O(log n) or O(1).

Examples of each common time complexity

- O(n!) [Factorial time]: Permutations of 1 ... *n*
- O(2^n) [Exponential time]: Exhaust all subsets of an array of size *n*
- O(n^3) [Cubic time]: Exhaust all triangles with side length less than *n*
- O(n^2) [Quadratic time]: Slow comparison-based sorting (eg. Bubble Sort, Insertion Sort, Selection Sort)
- O(n log n) [Linearithmic time]: Fast comparison-based sorting (eg. Merge Sort)
- O(n) [Linear time]: Linear Search (Finding maximum/minimum element in a 1D array), Counting Sort
- O(log n) [Logarithmic time]: Binary Search, finding GCD (Greatest Common Divisor) using Euclidean Algorithm

- O(1) [Constant time]: Calculation (eg. Solving linear equations in one unknown)

you get a clearer picture of how quickly each type of complexity increases with larger inputs. This is useful for understanding and comparing the efficiency of different algorithms.

log10 (n!)   = 8.68 **for** n = 12

log10(2^n)   = 7.52 **for** n = 25

log10(n^4)   = 8.00 **for** n = 100

log10(n^3)   = 8.09 **for** n = 500

log10(n^2)   = 8.00 **for** n = 10^4

log10(n log n) = 7.29 **for** n = 10^6

log10(n)   = 8.00 **for** n = 10^8