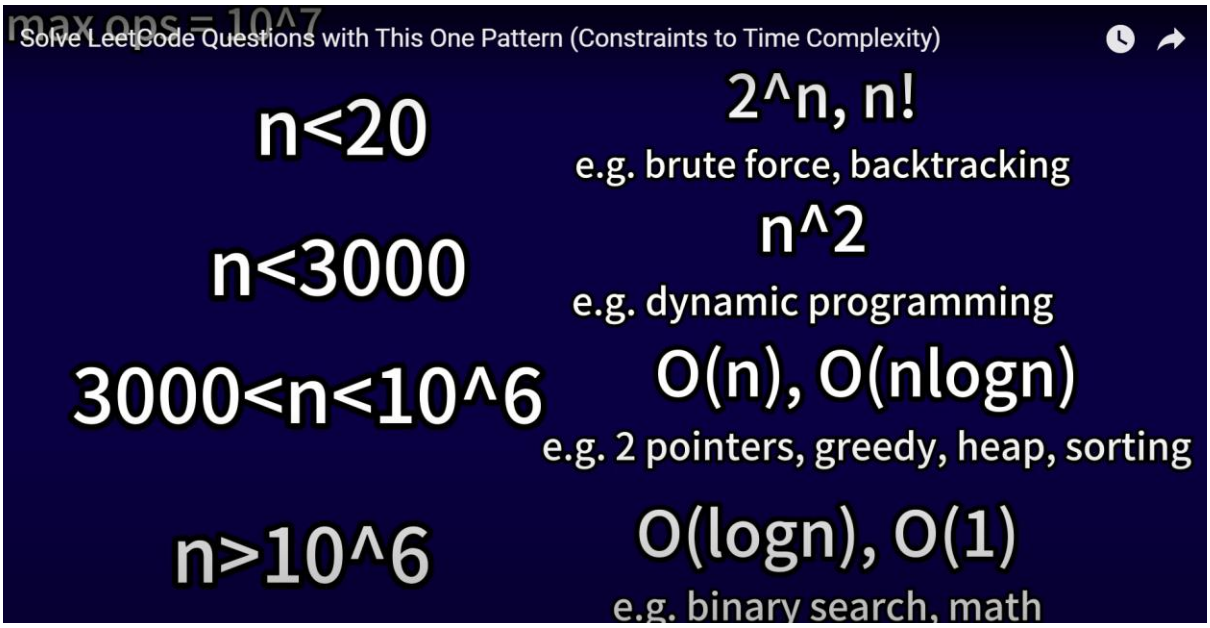
Constraints	**Worst time complexity**	**Algorithmic solution**	**Examples**
n **≤ 12**	O(n!)	Recursion & Backtracking	Permutation 1….n
n **≤ 25**	O(2n)	Recursion , Backtracking & Bit Manipulation	All subsets of an array of size n
n **≤ 100**	O(n4)	Dynamic Programming	4Sum
n **≤ 500**	O(n3)	Dynamic Programming	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

Data Types and their ranges

Data Type	**Actual Range**	**Approximate Range (10^)**
short int	-32,768 to 32,767	±3.2 × 10^4
unsigned short int	0 to 65,535	0 to 6.5 × 10^4
int	-2,147,483,648 to 2,147,483,647	±2.1 × 10^9
unsigned int	0 to 4,294,967,295	0 to 4.3 × 10^9
long int	-2,147,483,648 to 2,147,483,647	±2.1 × 10^9
unsigned long int	0 to 4,294,967,295	0 to 4.3 × 10^9
long long int	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	±9.2 × 10^18
unsigned long long int	0 to 18,446,744,073,709,551,615	0 to 1.8 × 10^19
float	~1.2E-38 to ~3.4E+38	~10^-38 to ~10^38
double	~2.3E-308 to ~1.7E+308	~10^-308 to ~10^308
long double	Greater than double (varies by implementation, e.g., 1.2E-4932 to 1.2E+4932)	~10^-4932 to ~10^4932 (varies)

Data Type	Actual Range	Approximate Range (10^)
char	-128 to 127 (signed) or 0 to 255 (unsigned)	±1.2 × 10^2 (signed) or 0 to 2.5 × 10^2 (unsigned)
unsigned char	0 to 255	0 to 2.5 × 10^2
signed char	-128 to 127	±1.2 × 10^2
bool	true or false	N/A
wchar_t	Implementation-defined (e.g., 0 to 4,294,967,295 for 4-byte wchar_t)	Varies

Types of Solutions Feasible for Upper Limits

Given the constraints, the following table suggests feasible solutions for different input size constraints and time complexity:

Input Size Constraint	Time Complexity	Feasible Solutions
N <= 10	O(N!)	Brute force permutations, backtracking.
N <= 20	O(2^N)	Dynamic programming with bitmasks, exhaustive search.
N <= 100	O(N^3)	Floyd-Warshall for shortest paths, cubic algorithms.
N <= 1,000	O(N^2)	Dynamic programming, graph algorithms (Floyd-Warshall).
N <= 10,000	O(N log N)	Sorting algorithms, binary search, segment trees.
N <= 100,000	O(N)	Linear time algorithms, counting sort, prefix sums.
N <= 1,000,000	O(N log N)	Efficient sorting, divide and conquer, balanced trees.
N <= 10,000,000	O(N)	Linear algorithms with efficient data structures (e.g., counting sort).

Special Constraints

- **Definition:** Unique conditions specific to the problem, like distinct values or specific properties of data (e.g., sorted array, connected graph).
- **Impact:** Adds complexity and requires consideration of additional factors in the solution.

Special Constraints and Their Solutions

1. **Distinct Elements**
   - **Example:** All array elements are distinct.

- **Feasible Solutions:** Sorting algorithms, unique combinations, or permutations can be leveraged effectively.

2. **Graph Connectivity**
   - **Example:** The graph is connected.
   - **Feasible Solutions:** DFS, BFS, and union-find algorithms for efficient traversal and connectivity checks.
3. **Sorted Input**
   - **Example:** The input array is already sorted.
   - **Feasible Solutions:** Binary search, two-pointer techniques, and efficient merge operations.

a comprehensive table that suggests feasible solutions based on different input size constraints and their corresponding time complexities:

Detailed Feasible Solutions for Different Input Sizes

Input Size Constraint	Time Complexity	Space Complexity	Feasible Solutions
N <= 10	**O(N!)**	**O(N)**	**Brute force permutations, backtracking, exhaustive search.**
N <= 20	**O(2^N)**	**O(2^N)**	**Dynamic programming with bitmasks, exhaustive search, subset generation.**
N <= 100	**O(N^3)**	**O(N^2)**	**Floyd-Warshall for shortest paths, cubic algorithms, dynamic programming with three nested loops.**
N <= 10^3	**O(N^2)**	**O(N^2)**	**Dynamic programming (e.g., longest common subsequence), graph algorithms (e.g., Floyd-Warshall, matrix multiplication).**
N <= 10^4	**O(N log N)**	**O(N)**	**Efficient sorting algorithms (e.g., merge sort, quicksort), binary search, line sweeping algorithms, segment trees.**
N <= 10^5	**O(N)**	**O(N)**	**Linear time algorithms (e.g., counting sort, radix sort, prefix sums, two-pointer techniques, linear scan).**
N <= 10^6	**O(N log N)**	**O(N)**	**Balanced tree algorithms, divide and conquer strategies, efficient data structures (e.g., Fenwick trees, AVL trees).**

Input Size Constraint	Time Complexity	Space Complexity	Feasible Solutions
N <= 10^7	O(N)	O(N)	Linear algorithms with efficient implementation (e.g., counting sort, radix sort, linear scan for specific problems).
N <= 10^8	O(N log log N)	O(N)	Sieve of Eratosthenes for prime number generation, certain amortized linear algorithms.
N <= 10^9	O(N)	O(N)	Specific linear algorithms, data streaming algorithms, counting frequencies.
N <= 10^{18}	O(log N)	O(1)	Binary search on ranges, logarithmic algorithms, number theory algorithms (e.g., modular arithmetic, logarithmic exponentiation).

Small Input Size (N <= 10)

- **O(N!)**
  - Brute force solutions.
  - Permutations and combinations.
  - Backtracking algorithms.

Moderate Input Size (N <= 20)

- **O(2^N)**
  - Subset generation.
  - Dynamic programming with bitmasks.
  - Recursive algorithms with memoization.

Larger Input Size (N <= 100)

- **O(N^3)**
  - Floyd-Warshall algorithm for all pairs shortest paths.
  - Matrix multiplication.
  - Dynamic programming for problems like the traveling salesman problem with small N.

Significant Input Size (N <= 1,000)

- **O(N^2)**
    - Dynamic programming for problems like the longest common subsequence.
    - Graph algorithms like Floyd-Warshall.
    - Matrix chain multiplication.

Large Input Size (N <= 10,000)

- **O(N log N)**
    - Efficient sorting algorithms like merge sort and quicksort.
    - Binary search algorithms.
    - Data structures like segment trees for range queries.

Very Large Input Size (N <= 100,000)

- **O(N)**
    - Linear algorithms like counting sort, radix sort.
    - Prefix sums for range sum queries.
    - Two-pointer techniques for problems involving subarrays or subsequences.

Extremely Large Input Size (N <= 1,000,000)

- **O(N log N)**
    - Balanced tree algorithms like AVL trees.
    - Efficient divide and conquer strategies.
    - Data structures like Fenwick trees for range queries.

Ultra Large Input Size (N <= 10,000,000)

- **O(N)**
    - Highly optimized linear algorithms.
    - Counting sort for specific ranges.
    - Linear scan algorithms for specific types of problems.

Massive Input Size (N <= 100,000,000)

- **O(N log log N)**
    - Sieve of Eratosthenes for prime number generation.
    - Amortized linear time algorithms for specific problem domains.

Giga Input Size (N <= 1,000,000,000)

- **O(N)**
    - Data streaming algorithms.

    - Counting frequencies in large data sets.
    - Highly specialized linear algorithms.

N <= 10^{18}

- **Time Complexity: O(log N)**
- **Space Complexity: O(1)**
    - Binary search on ranges, logarithmic algorithms like modular exponentiation, and number theory algorithms.

#note

Standard CPU operation 10^8 ops/s