n <= 10

The expected time complexity likely has a factorial or an exponential with a base larger than `2` - $O(n^2 \cdot n!)$ or $O(4^n)$ for example.

You should think about backtracking or any brute-force-esque recursive algorithm `n <= 10` is extremely small and usually **any** algorithm that correctly finds the answer will be fast enough.

10 < n <= 20

The expected time complexity likely involves $O(2^n)$. Any higher base or a factorial will be too slow (3^{20} = ~3.5 billion, and $20!$ is much larger). A 2^n usually implies that given a collection of elements, you are considering all subsets/subsequences - for each element, there are two choices: take it or don't take it.

Again, this bound is very small, so most algorithms that are correct will probably be fast enough. Consider **backtracking and recursion.**

20 < n <= 100

At this point, exponentials will be too slow. The upper bound will likely involve $O(n^3)$.

Problems marked as "easy" on LeetCode usually have this bound, which can be deceiving. There may be solutions that run in $O(n)$, but the small bound allows brute force solutions to pass (finding the linear time solution might not be considered as "easy").

Consider brute force solutions that involve nested loops. If you come up with a brute force solution, try analyzing the algorithm to find what steps are "slow", and try to improve on those steps using tools like hash maps or heaps.

100 < n <= 1,000

In this range, a quadratic time complexity $O(n^2)$ should be sufficient, as long as the constant factor isn't too large.

Similar to the previous range, you should consider nested loops. The difference between this range and the previous one is that $O(n^2)$ is usually the expected/optimal time complexity in this range, and it might not be possible to improve.

1,000 < n < 100,000

n<=105 is the most common constraint you will see on LeetCode. In this range, the slowest acceptable **common** time complexity is $(n \cdot \log n)$, although a linear time approach $O(n)$ is commonly the goal.

In this range, ask yourself if sorting the input or using a heap can be helpful. If not, then aim for an $O(n)$ algorithm. Nested loops that run in $O(n^2)$ are unacceptable - you will probably need to make use of a technique learned in this course to simulate a nested loop's behavior in $O(1)$ or $O(\log n)$:

- Hash map
- A two pointers implementation like sliding window.
- Monotonic stack
- Binary search
- Heap
- A combination of any of the above

If you have an $O(n)$ algorithm, the constant factor can be reasonably large (around 40). One common theme for string problems involves looping over the characters of the alphabet at each iteration resulting in a time complexity of $O(26n)$.

100,000 < n < 1,000,000

$n<=10^6$ is a rare constraint, and will likely require a time complexity of $O(n)$. In this range, $O(n \cdot \log n)$ is usually safe as long as it has a small constant factor. You will very likely need to incorporate a hash map in some way.

1,000,000 < n

With huge inputs, typically in the range of 10^9 or more, the most common acceptable time complexity will be logarithmic $(\log n)$ or constant $)O(1)$. In these problems, you must either significantly reduce your search space at each iteration (usually binary search) or use clever tricks to find information in constant time (like with math or a clever use of hash maps).

Other time complexities are possible like $O(n)$, but this is very rare and will usually only be seen in very advanced problems.