Digit DP

Digit DP is a technique to count or compute something about numbers by processing
their digits one-by-one (most often from most-significant to least-significant).

Instead of iterating every number up to N (too slow),
you treat the digits of N as a boundary and use dynamic programming to count all valid digit sequences
that form numbers ≤ N.

You build a DP over positions (digits), plus extra state(s) that
capture constraints (like whether you already matched the prefix of N,
whether you have started the real number or are still in leading zeros,
current sum of digits, a "mod" state, previous digit, etc.).

Core ideas / state variables (the classic ones)

-> pos – current digit index (0 .. len-1). We usually process digits from left (most significant) to right.

-> tight (or isBound) – whether the prefix you've chosen so far is exactly equal to the prefix of N.
If tight == 1, next digit can't exceed digits[pos].
If tight == 0, next digit can be 0..9 freely.

-> leadingZero (or started) – whether we still have only leading zeros (we haven't placed any non-zero digit yet).
Useful for problems where leading zeros shouldn't count toward sum or previous-digit constraints.

Additional states as needed: sumSoFar, modSoFar, prevDigit, countOfSomething, etc.

Memoization rule:
if your state includes pos, tight, leadingZero, and other bounded states
(like sum modulo some value, or small sum), you can memoize results for states to avoid recomputation.
Usually pos max is ~20 for 64-bit numbers, and other states are bounded to reasonable sizes.

Template (recursive + memoization)

This is the generic recursive structure you'll use:

```
long long dfs(int pos, int tight, int leadingZero, /* other states */) {
    if (pos == len) {
        // base case: evaluate if current number (constructed so far) is valid
        return /* 1 or 0 or sum value etc. */;
    }
    if (memoized) return memo;

    long long ans = 0;
    int maxDigit = tight ? digits[pos] : 9;
    for (int d = 0; d <= maxDigit; ++d) {
        int nextTight = tight && (d == maxDigit);
        int nextLeadingZero = leadingZero && (d == 0);
        // compute other next states
        ans += dfs(pos + 1, nextTight, nextLeadingZero, /* next states */);
    }
    store in memo if safe;
    return ans;
}
```

Important: In many solutions people only memoize when tight == 0.
But it's safe to include tight in the memoization keys so long as the
digits array (the bound) is the same across calls (which it is).

Example 1 — Count numbers 0..N that do not contain digit 4

Problem statement: Given non-negative integer N, count how many integers x
where 0 <= x <= N and decimal representation of x does not contain the digit 4.

Let's take a tiny N = 57.

Valid numbers:
0, 1, 2, 3, skip 4, 5, 6, 7, 8, 9,
10..13, skip 14, 15..19,
20..23, skip 24, 25..29
...
50..53, skip 54, 55, 56, 57

We want a program that counts these without iterating all numbers.

Digit DP works because:

You do not need to build the entire number to know whether future numbers are possible. You only need:

- which digit position you are at
- whether you are still tight (must obey the upper bound)
- whether you are still in leading zeros
- and any other constraints (like digit sum, previous digit, etc.)

These few states tell you how many valid completions exist from this point.

This is EXACTLY the same idea as DP in other domains:
→ "Given the current state, how many ways can the rest be completed?"

Digit DP is just DP on the digits.

Our States

`dfs(pos, tight, leadingZero)`

Meaning:

→ pos = which digit we are filling (0 = tens place, 1 = ones place)
→ tight = 1 = we cannot exceed the corresponding digit of N
→ leadingZero = 1 = we have not placed any non-zero digit yet

Digits of 57:
digits = [5, 7]
positions:

- pos=0 → digit '5' (tens)
- pos=1 → digit '7' (ones)

```
vector<int> digits;
ll dp[20][2][2]; // pos, tight, leadingZero
int len;

ll dfs(int pos, int tight, int leadingZero) {
    if (pos == len) {
        // reached end => one valid number constructed
        return 1;
    }

    ll &res = dp[pos][tight][leadingZero];

    if (res != -1) return res;
    res = 0;

    int limit = tight ? digits[pos] : 9;

    for (int d = 0; d <= limit; ++d) {
        if (d == 4) continue; // forbid digit 4

        int nextTight = tight && (d == limit);
        int nextLeadingZero = leadingZero && (d == 0);

        res += dfs(pos + 1, nextTight, nextLeadingZero);
    }

    return res;
}
```

Complexity summary

If your state size is S (product of all discrete state dimensions) and you try 0..9 at each pos,
complexity ≈ O(len * S * 10). For typical problems len ≤ 20, and S might be 2*2*180 = 720 or similar — easily fast.

O(number_of_states × transitions_per_state)