

Digit DP

Daryll Ko



Consider the following problem:



問題 1.

Given two integers ℓ and r , count the number of integers between ℓ and r inclusive.

There exist straightforward approaches to this problem, but here we consider a dynamic programming solution. But...where do we even start? To simplify things, let's first consider the following simpler problem:



問題 2.

Given an integer r , count the number of integers between 1 and r inclusive.

If we denote the problem above as a function **ans** of the input r , then the answer to the first problem is simply $\text{ans}(r) - \text{ans}(\ell - 1)$.



質問 3.

Why do we subtract $\text{ans}(\ell - 1)$ instead of $\text{ans}(\ell)$?

Okay, so dynamic programming. What kind of “state” do we consider? Well, the only thing we have to work with in the second problem is the number r , so let's use that.

Any number is a sequence of *digits*; we can write r in the form $d_1 d_2 \dots d_n$. Our problem now involves counting *sequences* that satisfy a certain condition. Let's reframe the problem like this:



問題 4.

Count the number of ways to assign a digit to each of n slots $a_1, a_2, \dots, a_n$ such that the integer formed by gluing these digits together is at most $r = d_1 d_2 \dots d_n$.

As with many DP problems involving sequences, what if we try to use our position in the sequence as our state?

Well, that gives us the following definition:



定義 5.

ans(i) is the number of ways to assign digits to the slots $a_i, a_{i+1}, \dots, a_n$.

If the “such that ... is at most r ” part wasn't there, the recurrence would be simple:

$$\begin{aligned}\text{ans}(i) &= \underbrace{\text{ans}(i+1)}_{d_i=0} + \underbrace{\text{ans}(i+1)}_{d_i=1} + \dots + \underbrace{\text{ans}(i+1)}_{d_i=9} \\ &= 10 \times \text{ans}(i+1).\end{aligned}$$

Note that the base case is $\text{ans}(n+1) = 1$, to represent the empty assignment.

But alas, we have a condition to deal with. Well, what distinguishes whether our current sequence is “at most r ”? Recall that if we have two sequences $a := (a_1, a_2, \dots, a_n)$ and $b := (b_1, b_2, \dots, b_n)$, then $a < b$ (in the Python sense) iff there exists an index i such that:

- $a_j = b_j$ for $1 \leq j < i$,
- $a_i < b_i$, and
- a_j can be anything for $i+1 \leq j \leq n$.

So in English, we keep seeing the same digit until we don’t. Whether we’ve seen a smaller digit or not will dictate if we can set subsequent digits freely, so we might want to consider including it in our state:



定義 6.

$\text{ans}(i, \text{less_seen})$ is the number of ways to assign digits to the slots $a_i, a_{i+1}, \dots, a_n$, where less_seen is `True` iff we have encountered some index j for which $a_j < d_j$.

Now, what are our transitions? We have

$$\begin{aligned}\text{ans}(i, \text{True}) &= 10 \times \text{ans}(i+1, \text{True}) \\ \text{ans}(i, \text{False}) &= \underbrace{\text{ans}(i+1, \text{True})}_{a_i=0} + \underbrace{\text{ans}(i+1, \text{True})}_{a_i=1} + \dots + \underbrace{\text{ans}(i+1, \text{True})}_{a_i=d_i-1} + \underbrace{\text{ans}(i+1, \text{False})}_{a_i=d_i} \\ &= d_i \times \text{ans}(i+1, \text{True}) + \text{ans}(i+1, \text{False}).\end{aligned}$$



質問 7.

Why are the terms $\underbrace{\text{ans}(i+1, \text{False})}_{a_i=d_i+1}, \dots, \underbrace{\text{ans}(i+1, \text{False})}_{a_i=9}$ not included in the second case?



質問 8.

What should the base case(s) be?

Thus, we have the following implementation:

Code Block 9.

```
from functools import cache

def solve_one(r: int) -> int:
    digits = [*map(int, list(str(r)))]
    n = len(digits)

    @cache
    def ans(i: int, less_seen: bool) -> int:
        if i == n:
            return 1

        if less_seen:
            return 10 * ans(i+1, True)
        else:
            return digits[i] * ans(i+1, True) + ans(i+1, False)
```

```

    return ans(0, False)

def solve(l: int, r: int) -> int:
    return solve_one(r) - solve_one(l - 1)

```

There are $\mathcal{O}(\log r)$ states (r has $\mathcal{O}(\log r)$ digits) and each state takes $\mathcal{O}(1)$ time, so each call to `solve_one` (and thus `solve`) takes $\mathcal{O}(\log r) \times \mathcal{O}(1) = \mathcal{O}(\log r)$ time.

This is, of course, a pretty silly solution (especially if you compare it to the standard $\mathcal{O}(1)$ one), but it highlights the power of treating numbers as *sequences of digits* then forming a notion of *state* based on that idea.



質問 10.

Implement the $\mathcal{O}(1)$ solution. Don't overthink it!

How were we able to compress $\mathcal{O}(r)$ numbers into $\mathcal{O}(\log r)$ states? What we did here is that we *partitioned* all numbers up to r into different *classes*. For example, `ans(i, ...)` represents two disjoint classes where all numbers in each class have $n - i + 1$ digits.

This technique, where we think in terms of the digits of the number, is called *digit DP*.

Now that you've seen an example, try the following problem!



問題 11.

Given two integers ℓ and r , count the number of integers between ℓ and r inclusive where no two adjacent digits in the number are the same.



質問 12.

The following code is meant to be part of a solution to the problem above. Can you identify what's wrong with it?

Code Block 13.

```

@cache
def ans(i: int, less_seen: bool, last_seen: int) -> int:
    if i == n:
        return 1

    res = 0

    for d in range(10):
        if d != last_seen:
            if less_seen:
                res += ans(i + 1, True, d)
            elif d <= digits[i]:
                res += ans(i + 1, d < digits[i], d)

    return res

```