6.4.9 Digit DP

Consider the following problem:

> How many numbers in the range $[a, b]$ where $1 \leq a \leq b \leq 10^{15} - 1$ contain exactly five 5's?[90]

While there may be combinatorial methods, we look at more flexible enumerative way. We can't directly check all numbers in the range, but techniques enable us to examine possible solutions one digit at a time. In order to understand how to do this, we will solve slightly easier problems:

1. How many numbers exist between 1 and $10^{15} - 1$?
2. How many numbers exist between 1 and any fixed $b < 10^{15}$?
3. How many numbers exist between 1 and some fixed b which have a specific number of 5's in their decimal representation?

Counting Integers We trust readers can successfully determine how many integers exist between 1 and $10^{15} - 1$,[91] however consider how we can compute this programmatically. Consider what possibilities we have for each digit: any of the 15 places in the number can take any value from 0-9. If we knew of no other way, we could build a basic DP to count the number of ways to construct a 15-digit number, one digit at a time.

1. Indices: The index of the digit we are examining (ones, tens, hundres, etc.).
2. Value: The count of unique numbers we can make using that many digits.
3. Relation: If we can construct dp_i unique numbers of length i, then for every possible digit in the $i+1$-th position, we add dp_i to dp_{i+1}.[92]
4. Iteration order: We increase the number of digits, from 0 to 15.
5. Base Case: There is 1 way to make a number of length 0.

```
ll dp[16]={};
dp[0]=1;
for(int i=0;i<16;i++){
   for(int next_digit=0;next_digit<=9;next_digit++){
      dp[i+1]+=dp[i];
   }
}

//there are dp[i] ways to make an integer of length i
```

[90] base 10

[91] If not, there are $10^{15} - 1$ integers between 1 and $10^{15} - 1$

[92] If one works it out, as there are 10 possible digits at each place, we end up with $dp_{i+1} = 10 * dp_i$

From an implementation standpoint, note the above code actually counts 000...0, which can be accounted for trivially if necessary by subtracting 1 from the solution if necessary.

Adding Upper Bounds The simplicity of the above method was possible since all numbers of length 15 are admissible. If we wish to only count up to some other target, say 6785, it doesn't apply, invalidating the DP setup. Instead, think about what digits are valid when attempting to enumerate all positive integers up to 6785 (inclusive):

- All digits up to 6 are valid at the first position. (e.g. 5XXX)
- All digits up to 7 are valid at the second position (e.g. 67XX), but so are 8 and 9 if the first digit was a 5 or lower (e.g. 59XX is valid, but 69XX is not).
- All digits up to 8 are valid at the third position (e.g. 678X), but so 9 if the first digit was a 5 or lower or the second digit was 6 or lower.
- All digits up to 5 are valid at the third position (e.g. 6784), but so are digits from 6-9, but only if one of the first three digits was lower than the target.

This ultimately leaves us with two general states:

1. We can always use any digit up to the target digit at a given place.
2. We can use digits above the target, but only if some earlier digit was less than its target.

We now have a clean extension to our DP by extending our state to include whether or not we have seen some digit that is less than its target.

1. Indices: The index of the digit we are examining (e.g. ones, tens, hundres, etc.) and whether we have seen some more-significant digit which is less than its target.
2. Value: The count of prefixes of valid numbers (i.e. numbers which are less than the target), up to so many places.[93]
3. Relation: We extend the valid prefixes by one digit. If we have seen some digit which was less than its target, all digits are valid, otherwise only those up to the target are. If the digit we select is less than the target at this place, we encode that information in the proper index.
4. Iteration order: We increase the number of digits.

[93] We must go most significant to least in order to track whether we have seen a more significant digit less than its target. Therefore the output is really the count of prefixes than the numbers themselves. For the example, after the first digit, we would have prefixes of 0XXX, 1XXX, 2XXX, 3XXX, 4XXX, 5XXX, and 6XXX, so 6 total prefixes of length 1.

5. Base Case: There is 1 way to make a number of length 0 without having seen a digit less than its target.

In order to implement this without worrying about the limits of integer size, we assume our target is stored in a string.

```
//assume upper bounds stored in string "high"
//this is our "target"

//place index and whether we’ve seen a digit lower than
//the target at any more significant place
ll dp[high.length()+1][2]={};
dp[0][0]=1; //base case
for(int i=0;i<high.length();i++){
   for(int under=0;under<2;under++){
      int low_digit=0;
      //the highest digit at this place is 9 if we have seen some
      //number lower than its target already, or the digit
      //at the right place in the target itself otherwise
      int high_digit=under?9:(high[i]-’0’);
      for(int next_digit=low_digit;next_digit<=high_digit;next_digit++){
         //the under value stays the same, unless our new digit is
         //under its target, in which case we set the under flag
         bool next_under=under||next_digit<(high[i]-’0’);
         dp[i+1][next_under?1:0]+=dp[i][under];
      }
   }
}

//there are dp[i][0]+d[i][1] ways to make a prefix with length i,
//including numbers with leading 0’s (e.g. 000XX)
```

As earlier the code actually counts 000...0, which can be accounted for trivially if necessary by subtracting 1 from the solution.

Counting Digits With our ability to enumerate all numbers up to a given maximum by going digit by digit, it is a small leap to add the number of 5’s we have seen to our state.

1. Indices: The index of the digit we are examining (e.g. ones, tens, hundres, etc.), whether we have seen some more-significant digit which is less than its target, and the count of 5’s we have included.

2. Value: The count of prefixes to valid numbers (i.e. numbers which are less than the target) up to so many places, which include exactly some count of 5’s.

3. Relation: We extend the valid prefixes by one digit. If we have seen some digit which was less than its target, all digits are valid, otherwise only

those up to the target are. If the digit we select is less than the target at this place, we encode that information in the proper index. If the digit we select is a 5, we also encode the proper index.

4. Iteration order: We increase the number of digits.

5. Base Case: There is 1 way to make a number of length 0 without having seen a digit less than its target or any 5’s.

In this implementation, due to the tracking of all the variations of 5’s, there will be a significant number of invalid states (such as having 3 5’s when we have only seen 2 digits). To avoid doing unnecessary work, we add an extra check to skip processing states by skipping processing if their count is 0.

```
//assume upper bounds stored in string "high"

//place index and whether we’ve seen a digit lower than
//the target at any more significant place...and the
//number of 5’s we’ve seen at any previous place
ll dp[high.length()+1][2][high.length()+1]={};
dp[0][0][0]=1; //base case
for(int i=0;i<high.length();i++){
  for(int under=0;under<2;under++){
    for(int num_fives=0;num_fives<=high.length();num_fives++){
      //skip any entries which are 0
      if(!dp[i][under][num_fives])continue;
      //find the bounds of the digit at this place
      int low_digit=0;
      int high_digit=under?9:(high[i]-’0’);
      for(int next_digit=low_digit;next_digit<=high_digit;next_digit++){
        bool next_under=under||next_digit<(high[i]-’0’);
        //if this digit is a 5 increase the number of 5’s we’ve seen
        int next_num_fives=num_fives+(next_digit==5?1:0);
        dp[i+1][next_under?1:0][next_num_fives]+=dp[i][under][num_fives];
      }
    }
  }
}

//there are dp[i][0][5]+d[i][1][5]
//ways to make a number no greater than "high" with exactly 5 5’s.
```

Lower Bounds Given we have the ability to compute the count of numbers which have five 5’s and are below some target, we can also compute the count within some range. A straightforward way could be to run the DP twice, once at the high end of the range, and once at one less than the low end of the range, and subtract the two.[94] In order to avoid unintentionally excluding the lower endpoint, we could either write a routine to subtract 1 from the lower endpoint (a subtraction which is difficult if the bounds exceeds common integer types), or otherwise writing a routine to explicitly check if the lower endpoint matches the criteria we are looking for (which is at best, extra code). Another option is to extend the DP by yet *another* dimension, encoding whether we have seen a digit higher than the low target, mirroring the one that tracks the upper target.

We note the following about the implementation:

- We pad the ”low” target with zeros to avoid dealing with the differing string lengths.
- Tracking which dimension is which in the array becomes critical as the number of dimensions grow.
- If memory consumption is an issue, note the relation only depends on the immediately preceding digit, and thus can be done with only two ”rows” rather than storing one for each index.

```
//assume bounds stored in strings "low" and "high"

//pad low to be at equal length to high
while(low.length()<high.length())low="0"+low;

//place index, seen a digit above the lower target,
//seen a digit below the upper target, and count of 5’s
ll dp[high.length()+1][2][2][high.length()+1]={};
dp[0][0][0][0]=1; //base case
for(int i=0;i<high.length();i++){
 for(int over=0;over<2;over++){
  for(int under=0;under<2;under++){
   for(int num_fives=0;num_fives<=high.length();num_fives++){
    if(!dp[i][over][under][num_fives])continue;
    //find the bounds of the digit at this place
    int low_digit=over?0:(low[i]-’0’);
    int high_digit=under?9:(high[i]-’0’);
    for(int next_digit=low_digit;
      next_digit<=high_digit;
      next_digit++){
     bool next_over=over||next_digit>(low[i]-’0’);
     bool next_under=under||next_digit<(high[i]-’0’);
     int next_num_fives=num_fives+(next_digit==5?1:0);
     dp[i+1][next_over?1:0][next_under?1:0][next_num_fives]+=
      dp[i][over][under][num_fives];
    }
   }
```

[94] As the DP computes up to the target inclusive, if we subtracted the lower bounds exactly, we would not have included the lower bound in our target inveral.

```
    }
  }
}

//sum up all dp[high.length()][X][X][5] to get the result
```

Identifying Digit DP The technique described here can be applied not just to counting numbers with a certain number of a certain digit, but to any metric which can be computed one digit at a time. This technique of computing such counts one digit at a time is called *Digit DP*.

Typical hints that the technique applies are:

- Counting numbers which have a specific property over a range.
- The endpoints of the range are large, generally in the 10^{10} to 10^{20} range, though this can vary depending on how complex the rest of the dimensions are.
- The property can be encoded in a small number of states which can be determined one digit at a time.

6.4.9.1 Expanded Techniques

While techniques such as computing the sum of the digits, or the number of a specific digit are straightforward, there are other techniques which are more involved. We cannot cover every possible application here, but illustrate some common patterns.

Computing Overall Divisibility If a number n is divisible by some d, we know $n \pmod{d} = 0$. How can we compute the modulus digit by digit? Fortunately, the modulus operator distributes across addition, so $6785 \pmod{d} = (6*1000) \pmod{d} + (7*100) \pmod{d} + (8*10) \pmod{d} + (5*q) \pmod{d}$. This results in a computation which is performed one digit at a time so long as we the value of the partial sum, $\pmod{d}$ to our DP state. This adds a dimension of size d, but once the DP is complete, the count of numbers which are divisible by d will be stored in the element with a mod of 0.

When implementing, computing the powers of 10 under some modulus (10^i (d))for each entry can be time consuming, and thus should be precomputed. Some problems require computing the divisibility of the number under multiple divisors. If possible, it may be faster to compute the results independently with separate DP runs rather than adding both to the state.[95]

```
//assume bounds stored in strings "high"

//place index, seen a digit below target, and mod
```

[95] Though this may not always be possible

```
ll dp[high.length()+1][2][MOD]={};
dp[0][0][0]=1; //base case

//precompute (10^i)%MOD
ll pow10mod[high.length()];
for(int i=0;i<high.length();i++)
   pow10mod[i]=(ll)pow(10,high.length()-i-1)%MOD;

for(int i=0;i<high.length();i++){
   for(int under=0;under<2;under++){
      for(int mod=0;mod<MOD;mod++){
         if(!dp[i][under][mod])continue;
         int high_digit=under?9:(high[i]-'0');
         for(int next_digit=0;next_digit<=high_digit;next_digit++){
            bool next_under=under||next_digit<(high[i]-'0');
            dp[i+1][next_under?1:0][(mod+next_digit*pow10mod[i])%MOD]+=
               dp[i][under][mod];
         }
      }
   }
}

//sum up all dp[high.length()][X][0] to get the result
```

Divisibility of Product of Digits In order to determine if the product of digits is divisble by some number, we could use the same method above (tracking mods), but this lessens the magnitude of the divisor (low thousands, perhaps). To go further, consider there are other ways to determine divisibility. At its core, a number is divisble by another if the multiplicity of any prime factor is greater in the dividend than the divisor. For example:

$$\frac{360}{12} = \frac{2^3 * 3^2 * 5}{2^2 * 3}$$

Since there are more 2's in the numerator than the denominator (3 vs 2) and more 3's (2 vs 1), we know for certain 360 is divisible by 12. This means in order to track the divisibility of the product of the digits going one digit at a time, we simply need to track the multiplicity of prime factors rather than the overall modulus. In the end, we can check if the multiplicity is high enough for all numbers in that class to be divisible by the intended divisor. As every factor is a single digit,[96] the highest prime factor is 7, and we can add all four potential factors[97] as dimensions in the DP. When we have completed the DP, we can examine states which have seen enough of each of the appropriate factors to determine the count of numbers which are devisible by the target divisor.

[96] since we're taking the product of the digits themselves

[97] 2,3,4, and 7

For each digit, we compute how many of each prime factor we would contribute to the overall product and use that information to update the appropriate entry in the DP array. For efficiency, we only store the multiplicity of each factor up to the max required for the divisor[98] and therefore must take care not to overflow any particular dimension once we have hit that maximum.

```
//twos->sevens store the multiplicity of that prime factor in the divisor

//cache the prime factors of each possible digit
vector<int> pf[10]={{},{},{2},{3},{2,2},{5},{2,3},{7},{2,2,2},{3,3}};

//index, under the high target, factor multiplicity
ll dp[high.size()+1][2][twos+1][threes+1][fives+1][sevens+1]={};
dp[0][0][0][0][0][0]=1; //base case
for(int i=0;i<high.length();i++){
  for(int under=0;under<2;under++){
    for(int tw=0;tw<=twos;tw++){
      for(int th=0;th<=threes;th++){
        for(int fi=0;fi<=fives;fi++){
          for(int se=0;se<=sevens;se++){
            if(!dp[i][under][tw][th][fi][se])continue;
            int high_digit=under?9:(high[i]-’0’);
            for(int next_digit=0;next_digit<=high_digit;next_digit++){
              bool next_under=under||next_digit<(high[i]-’0’);
              int next_tw=tw,next_th=th,next_fi=fi,next_se=se;
              //walk through factors of next digit, and increase
              //the appropriate variable for each one
              for(int factor:pf[next_digit]){
                if(factor==2)next_tw++;
                if(factor==3)next_th++;
                if(factor==5)next_fi++;
                if(factor==7)next_se++;
              }
              //never set to higher than what is needed for the
              //divisor we’re looking for
              next_tw=min(next_tw,twos);
              next_th=min(next_th,threes);
              next_fi=min(next_fi,fives);
              next_se=min(next_se,sevens);
              dp[i+1][next_under][next_tw][next_th][next_fi][next_se]+=
                dp[i][under][tw][th][fi][se];
            }
          }
        }
      }
    }
  }
}
```

[98] since we don’t care about extra factors, only those required to hit the minimum

```
//answer is the sum of over the two entries that have all the factors set
```

It turns out the above code is not entirely correct. Conisder the number 10, with a divisor of 3. The above code finds no factors of 3, however $1 * 0 = 0$ is trivially divisible by 3. The issue is we have not handled the corner case were a factor of 0 makes the overall product divisble by any arbitrary divisor. While it is tempting to simply special case handling of zeros by maxing out all the factors, this also doesn't work. Digit DP always places some digit at every position, including potential leading zeros. If we simply special cased zeros, we would improperly track ”0011” as divisble by 3 as well, when 11 is not so divisible.

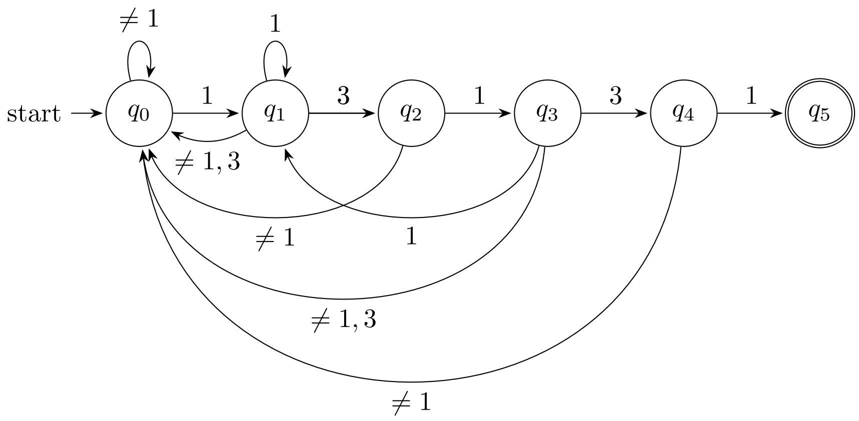
What we actually need to track is whether the zero appears after some other non-zero digit. This incurs the introduction of yet another dimension to the DP, whether we have seen some nonzero digit alreayd. Thus we only max out the factors on a zero if our state indicates we have already seen such a digit.

Fortunately, this monstrosity is the end of our state explosion and is mostly correct.

```
//twos->sevens store the multiplicity of that prime factor in the divisor

//cache the prime factors of each possible digit
vector<int> pf[10]={{},{},{2},{3},{2,2},{5},{2,3},{7},{2,2,2},{3,3}};

//index, under the high target, factor multiplicity, seen non-zero digit
ll dp[high.size()+1][2][twos+1][threes+1][fives+1][sevens+1][2]={};
dp[0][0][0][0][0][0][0]=1; //base case
for(int i=0;i<high.length();i++){
  for(int under=0;under<2;under++){
    for(int tw=0;tw<=twos;tw++){
      for(int th=0;th<=threes;th++){
        for(int fi=0;fi<=fives;fi++){
          for(int se=0;se<=sevens;se++){
            for(int nz_seen=0;nz_seen<2;nz_seen++){
              if(!dp[i][under][tw][th][fi][se][nz_seen])continue;
              int high_digit=under?9:(high[i]-’0’);
              for(int next_digit=0;next_digit<=high_digit;next_digit++){
                bool next_under=under||next_digit<(high[i]-’0’);
                //set non-zero seen if we had already seen non-zero or
                //if this character is not a zero
                bool next_nz_seen=nz_seen||next_digit;
                int next_tw=tw,next_th=th,next_fi=fi,next_se=se;
                for(int factor:pf[next_digit]){
                  if(factor==2)next_tw++;
                  if(factor==3)next_th++;
                  if(factor==5)next_fi++;
                  if(factor==7)next_se++;
                }
                //if non-zero digit seen, and this digit is a zero
                //then the number is divisible by anything, so
                //max out all the factors
                if(nz_seen&&!next_digit){
                  next_tw=twos;
                  next_th=threes;
                  next_fi=fives;
                  next_se=sevens;
                }
                next_tw=min(next_tw,twos);
                next_th=min(next_th,threes);
                next_fi=min(next_fi,fives);
                next_se=min(next_se,sevens);
                dp[i+1][next_under][next_tw][next_th]
                    [next_fi][next_se][next_nz_seen]+=
                    dp[i][under][tw][th][fi][se][nz_seen];
              }
            }
          }
        }
      }
    }
  }
}

//answer is the sum of over the two entries that have all the
//factors set, with some non-zero element seen
```

The unhandled case when this solution fails is when the divisor has some prime factor greater than 7. In that case, the product of digits is only divisible if it has some non-leading zero regardless of other prime factors. The implementation of this corner case can be done as an separate DP rather than trying to build it on top of the existing, factor-based one.

Pattern Tracking In our final pattern, we'll examine how we can use DP to find not just single digits, but also patterns of digits. Consider attempting to find all numbers which contain the pattern "13131." We'll first look at the automata we would use to match this pattern on an arbitrary string:



We can simply take each of those six automata states as a DP dimension to track how many prefixes are in each state, using the automata itself to compute the next state:

```
//assume upper bounds stored in string "high"
ll dp[high.length()+1][2][6]={}; //indices, under, automata state
dp[0][0][0]=1; //base case
for(int i=0;i<high.length();i++){
  for(int under=0;under<2;under++){
    for(int state=0;state<6;state++){
      if(!dp[i][under][state])continue;
      int low_digit=0;
      int high_digit=under?9:(high[i]-’0’);
      for(int next_digit=low_digit;next_digit<=high_digit;next_digit++){
        bool next_under=under||next_digit<(high[i]-’0’);
        //execute the automata transition for this state and digit
        int next_state=0;
        switch(state){
          case 0:
            if(next_digit==1)next_state=1;
            break;
          case 1:
            if(next_digit==1)next_state=1;
            if(next_digit==3)next_state=2;
            break;
          case 2:
            if(next_digit==1)next_state=3;
            break;
          case 3:
            if(next_digit==1)next_state=1;
            if(next_digit==3)next_state=4;
            break;
          case 4:
            if(next_digit==1)next_state=5;
            break;
          case 5:
            next_state=5;
        }
        dp[i+1][next_under][next_state]+=dp[i][under][state];
      }
    }
  }
}

//answer is anything in state 5 at the end
```

From this perspective, we can consider any digit DP as an automata which processes all numbers on the interval, one digit at a time. All information after the index dimension are simply states in the automata. We compute the proper next state transition based on the current state and the next digit.