# Count numbers

For this type of problems, we need to count how many numbers or how many digits meet the specified requirement. In each problem, there may be some formula you can build up by dynamic programming, but if you do not know the formula, the common DFS solution should help. You iterate on every digit, using memory cache, and watch if the search path is at the end edge as the last number. For example, if last number is 12345, then any paths like 12, 123, 1234 are at the end edge. If the search path is not at end edge, the lower digits can be anything from 0 to 9, but if it is at the end edge it can only be the remaining range of the last number.

## 233. Number of Digit One

Hard

Given an integer n, count the total number of digit 1 appearing in all non-negative integers less than or equal to n.

**Example:**

**Input:** 13

**Output:** 6

**Explanation:** Digit 1 occurred in the following numbers: 1, 10, 11, 12, 13.

### Analysis:

You iterate from the first digit to the last digit. The search path can be at start edge which is all 0 in this case, or it can be at the end edge which is the number n in this case. If the search path is not at end edge you know the low digits can be anything from 0 to 9, but if it is at the end edge it must be the remaining numbers for n.

/// <summary>

/// Leet code #233. Number of Digit One

/// </summary>

int LeetCodeMath::countDigitOne(string& str\_n, int is\_last, int index, vector<int>& cache)

{

int result = 0;

if (index == str\_n.size()) return 0;

if (is\_last == 0 && cache[index] != -1)

{

return cache[index];

}

for (int d = 0; d <= (is\_last ? str\_n[index] - '0' : 9); d++)

{

int next\_last = (is\_last == 1 && d == str\_n[index] - '0') ? 1 : 0;

result += countDigitOne(str\_n, next\_last, index + 1, cache);

if (d == 1)

{

if (index == str\_n.size() - 1) result += 1;

else

{

if (next\_last == 1) result += atoi(str\_n.substr(index + 1).c\_str()) + 1;

else

{

result += (int)pow(10, str\_n.size() - index - 1);

}

}

}

}

if (is\_last == 0) cache[index] = result;

return result;

}

/// <summary>

/// Leet code #233. Number of Digit One

/// Given an integer n, count the total number of digit 1 appearing in all

/// non-negative integers less than or equal to n.

/// For example:

/// Given n = 13,

/// Return 6, because digit 1 occurred in the following numbers: 1, 10,

/// 11, 12, 13.

/// Hint:

/// 1.Beware of overflow.

/// </summary>

int LeetCodeMath::countDigitOne(int n)

{

if (n <= 0) return 0;

string str\_n = to\_string(n);

vector<int> cache(str\_n.size(), -1);

int result = countDigitOne(str\_n, 1, 0, cache);

return result;

}

## 1067. Digit Count in Range

Hard

Given an integer d between 0 and 9, and two positive integers low and high as lower and upper bounds, respectively. Return the number of times that d occurs as a digit in all integers between low and high, including the bounds low and high.

**Example 1:**

**Input:** d = 1, low = 1, high = 13

**Output:** 6

**Explanation:**

The digit d=1 occurs 6 times in 1,10,11,12,13. Note that the digit d=1 occurs twice in the number 11.

**Example 2:**

**Input:** d = 3, low = 100, high = 250

**Output:** 35

**Explanation:**

The digit d=3 occurs 35 times in 103,113,123,130,131,...,238,239,243.

**Note:**

1. 0 <= d <= 9
2. 1 <= low <= high <= 2×10^8

### Analysis:

This problem is almost same as what you count digit 1, the only thing different is that there is a range, and digit can also be 0, so you need to check if the search path is at start edge.

/// <summary>

/// Leet code #1067. Digit Count in Range

/// </summary>

int LeetCodeMath::digitsCount(string& str\_n, int d, int is\_first, int is\_last, int index, vector<int>& cache)

{

int result = 0;

if (index == str\_n.size()) return 0;

if (is\_first == 0 && is\_last == 0 && cache[index] != -1)

{

return cache[index];

}

for (int i = 0; i <= (is\_last ? str\_n[index] - '0' : 9); i++)

{

int next\_first = (is\_first == 1 && i == 0) ? 1 : 0;

int next\_last = (is\_last == 1 && i == str\_n[index] - '0') ? 1 : 0;

result += digitsCount(str\_n, d, next\_first, next\_last, index + 1, cache);

if (d == 0 && is\_first == 1) continue;

if (i == d)

{

if (index == str\_n.size() - 1) result += 1;

else

{

if (next\_last == 1) result += atoi(str\_n.substr(index + 1).c\_str()) + 1;

else

{

result += (int)pow(10, str\_n.size() - index - 1);

}

}

}

}

if (is\_first == 0 && is\_last == 0) cache[index] = result;

return result;

}

/// <summary>

/// Leet code #1067. Digit Count in Range

///

/// Given an integer d between 0 and 9, and two positive integers low and

/// high as lower and upper bounds, respectively. Return the number of

/// times that d occurs as a digit in all integers between low and high,

/// including the bounds low and high.

///

/// Example 1:

/// Input: d = 1, low = 1, high = 13

/// Output: 6

/// Explanation:

/// The digit d=1 occurs 6 times in 1,10,11,12,13. Note that the digit d=1

/// occurs twice in the number 11.

///

/// Example 2:

///

/// Input: d = 3, low = 100, high = 250

/// Output: 35

/// Explanation:

/// The digit d=3 occurs 35 times in 103,113,123,130,131,...,238,239,243.

///

///

/// Note:

///

/// 0 <= d <= 9

/// 1 <= low <= high <= 2×10^8

/// </summary>

int LeetCodeMath::digitsCount(int d, int low, int high)

{

string str\_low = to\_string(low - 1);

vector<int> cache(str\_low.size(), -1);

int low\_count = digitsCount(str\_low, d, 1, 1, 0, cache);

string str\_high = to\_string(high);

cache = vector<int>(str\_high.size(), -1);

int high\_count = digitsCount(str\_high, d, 1, 1, 0, cache);

return high\_count - low\_count;

}

## 1012. Numbers With Repeated Digits

Hard

Given a positive integer N, return the number of positive integers less than or equal to N that have at least 1 repeated digit.

**Example 1:**

**Input:** 20

**Output:** 1

**Explanation:** The only positive number (<= 20) with at least 1 repeated digit is 11.

**Example 2:**

**Input:** 100

**Output:** 10

**Explanation:** The positive numbers (<= 100) with atleast 1 repeated digit are 11, 22, 33, 44, 55, 66, 77, 88, 99, and 100.

**Example 3:**

**Input:** 1000

**Output:** 262

**Note:**

1. 1 <= N <= 10^9

### Analysis:

For this problem, we add one more trick, instead of tracking whether the number is at starting edge with all zeros, we say how many leading non-zero digits in each position during original scan for the number.

/// <summary>

/// Leet code #1012. Numbers With Repeated Digits

/// </summary>

int LeetCodeMath::numDupDigitsAtMostN(string& str\_n, int index, int leading, int is\_last, int bit\_mask, vector<vector<int>>& cache)

{

if (index >= str\_n.size()) return 0;

if (is\_last == 0 && cache[index][leading] != -1) return cache[index][leading];

int result = 0;

for (size\_t i = 0; i <= (is\_last == 1 ? str\_n[index] - '0' : 9); i++)

{

int next\_leading = (leading == 0 && i == 0) ? 0 : leading + 1;

int next\_last = (is\_last == 1 && i == str\_n[index] - '0') ? 1 : 0;

if (next\_leading == 0)

{

result += numDupDigitsAtMostN(str\_n, index + 1, next\_leading, next\_last, bit\_mask, cache);

}

else

{

int bit = 1 << i;

if ((bit & bit\_mask) != 0)

{

if (next\_last == 1)

{

result += atoi(str\_n.substr(index + 1).c\_str()) + 1;

}

else

{

result += pow(10, str\_n.size() - 1 - index);

}

}

else

{

result += numDupDigitsAtMostN(str\_n, index + 1, next\_leading, next\_last, (bit | bit\_mask), cache);

}

}

}

if (is\_last == 0)

{

cache[index][leading] = result;

}

return result;

}

/// <summary>

/// Leet code #1012. Numbers With Repeated Digits

///

/// Given a positive integer N, return the number of positive integers less

/// than or equal to N that have at least 1 repeated digit.

///

/// Example 1:

/// Input: 20

/// Output: 1

/// Explanation: The only positive number (<= 20) with at least 1 repeated

/// digit is 11.

///

/// Example 2:

/// Input: 100

/// Output: 10

/// Explanation: The positive numbers (<= 100) with atleast 1 repeated digit

/// are 11, 22, 33, 44, 55, 66, 77, 88, 99, and 100.

///

/// Example 3:

/// Input: 1000

/// Output: 262

///

///

/// Note:

///

/// 1 <= N <= 10^9

/// </summary>

int LeetCodeMath::numDupDigitsAtMostN(int N)

{

string str\_n = to\_string(N);

int index = 0;

vector<vector<int>> cache(str\_n.size(),vector<int>(str\_n.size(), -1));

return numDupDigitsAtMostN(str\_n, 0, 0, 1, 0, cache);

}

## 1088. Confusing Number II

Hard

We can rotate digits by 180 degrees to form new digits. When 0, 1, 6, 8, 9 are rotated 180 degrees, they become 0, 1, 9, 8, 6 respectively. When 2, 3, 4, 5 and 7 are rotated 180 degrees, they become invalid.

A *confusing number* is a number that when rotated 180 degrees becomes a **different** number with each digit valid.(Note that the rotated number can be greater than the original number.)

Given a positive integer N, return the number of confusing numbers between 1 and N inclusive.

**Example 1:**

**Input:** 20

**Output:** 6

**Explanation:**

The confusing numbers are [6,9,10,16,18,19].

6 converts to 9.

9 converts to 6.

10 converts to 01 which is just 1.

16 converts to 91.

18 converts to 81.

19 converts to 61.

**Example 2:**

**Input:** 100

**Output:** 19

**Explanation:**

The confusing numbers are [6,9,10,16,18,19,60,61,66,68,80,81,86,89,90,91,98,99,100].

**Note:**

1. 1 <= N <= 10^9

### Analysis:

We also use leading digits here, one thing we need to watch is that at the middle point we need to deduct strobogrammatic numbers.

/// <summary>

/// Leet code #1088. Confusing Number II

/// </summary>

int LeetCodeMath::confusingNumberII(string& str\_n, int index, int leading, int is\_last, vector<vector<int>>& cache)

{

if (index >= str\_n.size()) return 1;

if (is\_last == 0 && cache[index][leading] != -1) return cache[index][leading];

int result = 0;

for (size\_t i = 0; i <= (is\_last == 1 ? str\_n[index] - '0' : 9); i++)

{

if (i != 0 && i != 1 && i != 6 && i != 8 && i != 9) continue;

int next\_leading = (leading == 0 && i == 0) ? 0 : leading + 1;

int next\_last = (is\_last == 1 && i == str\_n[index] - '0') ? 1 : 0;

int remaining = str\_n.size() - 1 - index;

result += confusingNumberII(str\_n, index + 1, next\_leading, next\_last, cache);

if ((remaining == next\_leading) ||

((remaining == next\_leading - 1) && (i == 0 || i == 1 || i == 8)))

{

if (next\_last == 0) result--;

else

{

int half\_size = str\_n.size() / 2;

string left\_str = str\_n.substr(0, half\_size);

string right\_str = str\_n.substr(str\_n.size() - half\_size);

vector<int> mapping = { 0, 1, 0, 0, 0, 0, 9, 0, 8, 6 };

for (size\_t i = 0; i < left\_str.size(); i++)

{

left\_str[i] = '0' + mapping[left\_str[i] - '0'];

}

std::reverse(left\_str.begin(), left\_str.end());

if (atoi(left\_str.c\_str()) < atoi(right\_str.c\_str())) result--;

}

}

}

if (is\_last == 0)

{

cache[index][leading] = result;

}

return result;

}

/// <summary>

/// Leet code #1088. Confusing Number II

///

/// We can rotate digits by 180 degrees to form new digits. When

/// 0, 1, 6, 8, 9 are rotated 180 degrees, they become 0, 1, 9, 8, 6

/// respectively. When 2, 3, 4, 5 and 7 are rotated 180 degrees, they

/// become invalid.

///

/// A confusing number is a number that when rotated 180 degrees

/// becomes a different number with each digit valid.(Note that the

/// rotated number can be greater than the original number.)

///

/// Given a positive integer N, return the number of confusing numbers

/// between 1 and N inclusive.

///

/// Example 1:

///

/// Input: 20

/// Output: 6

/// Explanation:

/// The confusing numbers are [6,9,10,16,18,19].

/// 6 converts to 9.

/// 9 converts to 6.

/// 10 converts to 01 which is just 1.

/// 16 converts to 91.

/// 18 converts to 81.

/// 19 converts to 61.

///

/// Example 2:

///

/// Input: 100

/// Output: 19

/// Explanation:

/// The confusing numbers are [6,9,10,16,18,19,60,61,66,68,80,81,86,89,

/// 90,91,98,99,100].

///

///

/// Note:

///

/// 1. 1 <= N <= 10^9

/// </summary>

int LeetCodeMath::confusingNumberII(int N)

{

string str\_n = to\_string(N);

vector<vector<int>> cache(str\_n.size(), vector<int>(str\_n.size(), -1));

return confusingNumberII(str\_n, 0, 0, 1, cache);

}

## 1397. Find All Good Strings

Hard

Given the strings s1 and s2 of size n, and the string evil. *Return the number of****good****strings*.

A **good** string has size n, it is alphabetically greater than or equal to s1, it is alphabetically smaller than or equal to s2, and it does not contain the string evil as a substring. Since the answer can be a huge number, return this modulo 10^9 + 7.

**Example 1:**

**Input:** n = 2, s1 = "aa", s2 = "da", evil = "b"

**Output:** 51

**Explanation:** There are 25 good strings starting with 'a': "aa","ac","ad",...,"az". Then there are 25 good strings starting with 'c': "ca","cc","cd",...,"cz" and finally there is one good string starting with 'd': "da".

**Example 2:**

**Input:** n = 8, s1 = "leetcode", s2 = "leetgoes", evil = "leet"

**Output:** 0

**Explanation:** All strings greater than or equal to s1 and smaller than or equal to s2 start with the prefix "leet", therefore, there is not any good string.

**Example 3:**

**Input:** n = 2, s1 = "gx", s2 = "gz", evil = "x"

**Output:** 2

**Constraints:**

* s1.length == n
* s2.length == n
* s1 <= s2
* 1 <= n <= 500
* 1 <= evil.length <= 50
* All strings consist of lowercase English letters.

### Analysis:

This problem is too difficult for interview, but still the same pattern. You need to not only track the s1 and s2 but also the evil string.

/// <summary>

/// Leet code #1397. Find All Good Strings

/// </summary>

int LeetCodeDFS::findGoodStrings(int index, string& s1, string& s2, string& evil,

int pre\_s1, int pre\_s2, int pre\_evil, vector<int> &kmp,

vector<vector<vector<vector<int>>>> &cache)

{

int M = 1000000007;

if (pre\_evil == evil.size()) return 0;

if (index == s1.size()) return 1;

if (cache[index][pre\_evil][pre\_s1][pre\_s2] != -1)

{

return cache[index][pre\_evil][pre\_s1][pre\_s2];

}

char first = (pre\_s1 == 1) ? s1[index] : 'a';

char last = (pre\_s2 == 1) ? s2[index] : 'z';

int result = 0;

for (char c = first; c <= last; c++)

{

int next\_evil = pre\_evil;

while (next\_evil > 0 && c != evil[next\_evil])

{

next\_evil = kmp[next\_evil - 1];

}

if (c == evil[next\_evil]) next\_evil++;

int next\_s1 = (pre\_s1 == 1 && c == first) ? 1 : 0;

int next\_s2 = (pre\_s2 == 1 && c == last) ? 1 : 0;

result += findGoodStrings(index + 1, s1, s2, evil, next\_s1,

next\_s2, next\_evil, kmp, cache);

result %= M;

}

cache[index][pre\_evil][pre\_s1][pre\_s2] = result;

return cache[index][pre\_evil][pre\_s1][pre\_s2];

}

/// <summary>

/// Leet code #1397. Find All Good Strings

///

/// Hard

///

/// Given the strings s1 and s2 of size n, and the string evil. Return

/// the number of good strings.

///

/// A good string has size n, it is alphabetically greater than or equal

/// to s1, it is alphabetically smaller than or equal to s2, and it does

/// not contain the string evil as a substring. Since the answer can be

/// a huge number, return this modulo 10^9 + 7.

///

/// Example 1:

/// Input: n = 2, s1 = "aa", s2 = "da", evil = "b"

/// Output: 51

/// Explanation: There are 25 good strings starting with 'a': "aa","ac",

/// "ad",...,"az". Then there are 25 good strings starting with 'c':

/// "ca","cc","cd",...,"cz" and finally there is one good string starting

/// with 'd': "da".

///

/// Example 2:

/// Input: n = 8, s1 = "leetcode", s2 = "leetgoes", evil = "leet"

/// Output: 0

/// Explanation: All strings greater than or equal to s1 and smaller than

/// or equal to s2 start with the prefix "leet", therefore, there is not

/// any good string.

///

/// Example 3:

/// Input: n = 2, s1 = "gx", s2 = "gz", evil = "x"

/// Output: 2

///

/// Constraints:

/// 1. s1.length == n

/// 2. s2.length == n

/// 3. 1 <= n <= 500

/// 4. 1 <= evil.length <= 50

/// 5. All strings consist of lowercase English letters.

/// </summary>

int LeetCodeDFS::findGoodStrings(int n, string s1, string s2, string evil)

{

vector<int> kmp(evil.size());

int i = 1;

int j = 0;

while (i < (int)evil.size())

{

if (evil[i] == evil[j])

{

j++;

kmp[i] = j;

i++;

}

else if (j == 0)

{

kmp[i] = 0;

i++;

}

else

{

j = kmp[j - 1];

}

}

vector<vector<vector<vector<int>>>>

cache(n, vector<vector<vector<int>>>(evil.size(),

vector<vector<int>>(2, vector<int>(2, -1))));

return findGoodStrings(0, s1, s2, evil, 1, 1, 0, kmp, cache);

}