

## How to count integers between large A and B with a certain property?



In programming contests, the following pattern occurs in a lot of tasks:

Given numbers A and B that are huge (maybe 20 decimal digits or more), determine the number of integers X with  $A \leq X \leq B$  that have a certain property P

SPOJ has lots of tasks like these for practice.

Where examples of interesting properties include:

- "the digit sum of X is 60"
- "X consists only of the digits 4 and 7"
- "X is palindromic", which means that the decimal representation of X equals its reverse (for example,  $X = 1234321$ )

I know that if we define  $f(Y)$  to be the number of such integers  $X \leq Y$ , then the answer to our question is  $f(B) - f(A - 1)$ . The reduced problem is how to compute the function  $f$  efficiently. In some cases we can make use of certain mathematical properties to come up with a formula, but often the properties are more complicated and we don't have enough time for that in a contest.

Is there a more general approach that works in a lot of cases? And can it also be used to enumerate the numbers with the given property or compute some aggregation on them?


A variation of this is to find the k-th number with a given property, which of course can be solved by using binary search together with a counting function.

algorithm dynamic-programming

edited Aug 12 '16 at 19:54

 **Stefan Pochmann**  
16.1k 3 18 51

asked Mar 14 '14 at 1:11

 **Niklas B.**  
64k 7 145 188

- 5 @JuanLopes: Yes. The idea is to document your ideas and results so that other people can profit from it, just like a blog :) I'm also trying to make the competitive programming community interested in Stack Overflow more, so maybe this will help set an example that questions like these are in fact welcome and on-topic here (and obviously I hope others will find this interesting as well) – [Niklas B.](#) Mar 14 '14 at 1:19
- 1 @JuanLopes: I'm also open to alternative answers of course – [Niklas B.](#) Mar 14 '14 at 1:21
- 2 @arunmoezhi You mean, self-answered questions? Because those occur all across SO (there is even a checkbox in the "Ask a question" dialog that lets you write an answer even before posting the question). Or do you mean competitive programming? I think I invented a tag for that already just now – [Niklas B.](#) Mar 14 '14 at 1:36
- 2 @Charles: Or to put it otherwise, this is pretty much a competitive-programming-specific question, because it describes a trick that is only useful in that setting and not in the "real world" – [Niklas B.](#) Mar 14 '14 at 4:55
- 1 @RobNeuhaus There's also programming-contest, so there is some confusion there, but both are meta-tags so I think it's probably best to get rid of them. Dukeling and others have a point – [Niklas B.](#) Mar 15 '14 at 18:00

### 1 Answer

Indeed, there is an approach to this pattern which turns out to work quite often. It can also be used to enumerate all the X with the given property, provided that their number is reasonably small. You can even use it to aggregate some associative operator over all the X with the given property, for example to find their sum.

To understand the general idea, let us try to formulate the condition  $X \leq Y$  in terms of the decimal representations of X and Y.

Say we have  $X = x_1 x_2 \dots x_{n-1} x_n$  and  $Y = y_1 y_2 \dots y_{n-1} y_n$ , where  $x_i$  and  $y_i$  are the decimal digits of X and Y. If the numbers have a different length, we can always add zero digits to the front of the shorter one.

Let us define `leftmost_lo` as the smallest  $i$  with  $x_i < y_i$ . We define `leftmost_lo` as  $n + 1$  if there is no such  $i$ . Analogously, we define `leftmost_hi` as the smallest  $i$  with  $x_i > y_i$  or  $n + 1$  otherwise.

Now  $X \leq Y$  is true if and exactly if `leftmost_lo`  $\leq$  `leftmost_hi`. With that observation it becomes possible to apply a [dynamic programming](#) approach to the problem, that "sets" the digits of X one after another. I will demonstrate this with your example problems:

Compute the number  $f(Y)$  of integers  $X$  with the property  $X \leq Y$  and  $X$  has the digit sum 60

Let  $n$  be the number of  $Y$ 's digits and  $y[i]$  be the  $i$ -th decimal digit of  $Y$  according to the definition above. The following recursive algorithm solves the problem:

```
count(i, sum_so_far, leftmost_lo, leftmost_hi):
    if i == n + 1:
        # base case of the recursion, we have recursed beyond the last digit
        # now we check whether the number X we built is a valid solution
        if sum_so_far == 60 and leftmost_lo <= leftmost_hi:
            return 1
        else:
            return 0
    result = 0
    # we need to decide which digit to use for x[i]
    for d := 0 to 9
        leftmost_lo' = leftmost_lo
        leftmost_hi' = leftmost_hi
        if d < y[i] and i < leftmost_lo': leftmost_lo' = i
        if d > y[i] and i < leftmost_hi': leftmost_hi' = i
        result += count(i + 1, sum_so_far + d, leftmost_lo', leftmost_hi')
    return result
```

Now we have  $f(Y) = \text{count}(1, 0, n + 1, n + 1)$  and we have solved the problem. We can add [memoization](#) to the function to make it fast. The runtime is  $O(n^4)$  for this particular implementation. In fact we can cleverly optimize the idea to make it  $O(n)$ . This is left as an exercise to the reader (Hint: You can compress the information stored in `leftmost_lo` and `leftmost_hi` into a single bit and you can prune if `sum_so_far > 60`). The solution can be found at the end of this post.

If you watch closely, `sum_so_far` here is just an example of an arbitrary function computing a value from the digit sequence of  $X$ . It could be *any* function that can be computed digit by digit and outputs a small enough result. It could be the product of digits, a bitmask of the set of digits that fulfill a certain property or a lot of other things.

It could also just be a function that returns 1 or 0, depending on whether the number consists only of digits 4 and 7, which solves the second example trivially. We have to be a bit careful here because we *are* allowed to have leading zeroes at the beginning, so we need to carry an additional bit through the recursive function calls telling us whether we are still allowed to use zero as a digit.

Compute the number  $f(Y)$  of integers  $X$  with the property  $X \leq Y$  and  $X$  is palindromic

This one is slightly tougher. We need to be careful with leading zeroes: The mirror point of a palindromic number depends on how many leading zeroes we have, so we would need to keep track of the number of leading zeroes.

There is a trick to simplify it a bit though: If we can count the  $f(Y)$  with the additional restriction that all numbers  $X$  must have the same digit count as  $Y$ , then we can solve the original problem as well, by iterating over all possible digit counts and adding up the results.

So we can just assume that we don't have leading zeroes at all:

```
count(i, leftmost_lo, leftmost_hi):
    if i == ceil(n/2) + 1: # we stop after we have placed one half of the number
        if leftmost_lo <= leftmost_hi:
            return 1
        else:
            return 0
    result = 0
    start = (i == 1) ? 1 : 0    # no leading zero, remember?
    for d := start to 9
        leftmost_lo' = leftmost_lo
        leftmost_hi' = leftmost_hi
        # digit n - i + 1 is the mirrored place of index i, so we place both at
        # the same time here
        if d < y[i] and i < leftmost_lo': leftmost_lo' = i
        if d < y[n-i+1] and n-i+1 < leftmost_lo': leftmost_lo' = n-i+1
        if d > y[i] and i < leftmost_hi': leftmost_hi' = i
        if d > y[n-i+1] and n-i+1 < leftmost_hi': leftmost_hi' = n-i+1
        result += count(i + 1, leftmost_lo', leftmost_hi')
    return result
```

The result will again be  $f(Y) = \text{count}(1, n + 1, n + 1)$ .

**UPDATE:** If we don't only want to count the numbers, but maybe enumerate them or compute some aggregate function from them which does not expose group structure, we need to enforce the lower bound on  $X$  as well during the recursion. This adds a few more parameters.

**UPDATE 2:**  $O(n)$  Solution for the "digit sum 60" example:

In this application we place the digits from left to right. Since we are only interested in whether `leftmost_lo < leftmost_hi` holds true, let us add a new parameter `lo`. `lo` is true iff

`leftmost_lo' < leftmost_hi'` can use any digit for the position `i`. If `lo` is false, then the digit would cause `leftmost_lo' < leftmost_hi'` to be true.

```
def f(i, sum_so_far, lo):
    if i == n + 1: return sum_so_far == 60
    if sum_so_far > 60: return 0
```

```

res = 0
for d := 0 to (lo ? 9 : y[i]):
    res += f(i + 1, sum + d, lo || d < y[i])
return res

```

Arguably, this way of looking at it is somewhat simpler, but also a bit less explicit than the `leftmost_lo / leftmost_hi` approach. It also doesn't work immediately for somewhat more complicated scenarios like the palindrome problem (although it can be used there as well).

edited Aug 29 '16 at 7:12



[rishabh\\_dev](#)

53 1 5

answered Mar 14 '14 at 1:11



[Niklas B.](#)

64k 7 145 188

1 Your algorithm for Compute the number  $f(Y)$  of integers  $X$  with the property  $X \leq Y$  and  $X$  has the digit sum 60 looks like it has a complexity of  $O(10^n)$ . I can't see how it is  $O(n^4)$ . The recursion level can go upto  $n$  and in each recursion there's a loop from 0 to 9. Am I missing something here? – [Rushil](#) May 19 '14 at 19:11

1 @Rushil As I said, you need to add memoization to make it fast. There are only  $O(n^4)$  different assignments of the parameters, so the function body is executed at most  $O(n^4)$  times, each time doing only constant work apart from the recursive calls (the loop from 0 to 9 can be considered constant time). Or in other words: There are  $10^n$  branches, but only  $O(n^4)$  of those are actually distinct subproblems. Since we don't solve a branch more than once, there are only  $O(n^4)$  nodes in the recursion tree – [Niklas B.](#) May 19 '14 at 19:57

So the  $O(10^n)$  complexity can be reduced to  $O(n^4)$  with memoization. How do you make it  $O(n^2)$  or better? – [Rushil](#) May 20 '14 at 5:46

@Rushil First realize that if `sum_so_far > 60`, you can return 0 immediately. This gets the runtime down to  $O(n^3)$ . Also realize that we're only interested in `leftmost_lo < leftmost_hi`, so if we set the digits from left to right like in this case, we just need one state bit to remember whether we have already placed a digit smaller than the upper bound. Example: [pastie.org/9199021](http://pastie.org/9199021) with memoization, you get  $O(n)$  runtime – [Niklas B.](#) May 22 '14 at 13:06

2 @NiklasB. didn't you have to memoize on both  $i$  and `sum_so_far` so it becomes  $O(n^2)$ ? correct me if I am wrong – [harish.venkat](#) May 23 '14 at 19:34