**15295 Spring 2018 #9 Counting -- Problem Discussion**
March 21, 2018

This is where we collectively describe algorithms for these problems. To see the problem statements follow this link. To see the scoreboard, go to this page and select this contest.

**A. Number of Ways**

Since you will be splitting the the array into three sections with the same sum, this means that the sum of numbers in each section must be equal to one-third the total sum. You can proceed as follows:

First, create a prefix-sum array (like [(a1), (a1 + a2), (a1 + a2 + a3), … , (a1 + … + an)]) and keep track of the total sum. If the total sum is not divisible by 3, then there is no way to split its components into 3 parts of equal sum, and you can print “0” immediately. Otherwise, calculate what one-third and two-thirds of the total would be.

Iterate through your prefix sum array, keeping track of how many indices you have seen so far where the sum up to that point was equal to one-third the total. These indices will be all possible places where the first section can end. Whenever you encounter an index where the prefix sum at that point is equal to two-thirds the total, then the number of ways to split up the array where that index is the end of the second section is equal to the number of possible places to end the first section, given that you will end the second section here. That value will be however many times you have seen a prefix sum of one-third the total so far. Add this number to some accumulator that keeps track of the total number of ways, and keep going. Once you get to the end of the array, this accumulator will be the sum, for each possible place to end the second section, of the number of possible places to end the first section given that the second section ends there. This covers all possible valid splits, so just return that value.

If the total sum is 0, things are a little tricky because any time the prefix sum is 0, that could be the end of the first OR the second section, but if you just check in your loop whether that prefix sum is equal to two-thirds BEFORE you check if its equal to one-third, you don’t have to make a special case for this.

-- Jacqui F

**B. Shaass and Lights**

Probably there’s a simpler way to think about this, but here’s my solution.

Think of the lights as a zero-indexed length-n boolean array A, where A[i] = 1 if light i is on and A[i] = 0 if light i is off. Take some 1 in A and partition A into [X 1 Y] where X is non-empty. Say that there are *x* ways to turn on the lights in [X 1] and that there are *a* zeros in X. Say that there are *y* ways to turn on the lights in [1 Y] and that there are *b* zeros in Y. Then there are *x* * *y* * ((*a* + *b*) choose *a*) ways to turn on the lights in A.

Where does this expression "*x* * *y* * ((*a* + *b*) choose *a*) come from? The "*x*" factor comes from picking a particular way W of turning the lights on in [X 1]. The "*y*" factor comes from picking a particular way U of turning the lights on in [1 Y]. Then there are ((*a* + *b*) choose *a*) ways to interleave these two ways of turning on the lights W and U --- we take (*a* + *b*) time steps to turn on (*a* + *b*) lights, and we choose *a* time steps out of these (*a* + *b*) time steps at which to flip on the next light in sequence W instead of in U.

Finally, we need some base cases. An input like 000...001 or 100...00 only has one way to turn on the lights. An input like 1000...0001 has 2^(*k* - ept for the last one.

Now our algorithm is to scan A from left-to-right, stopping at the first 1 that's not A[0]. This is the point at which we partition A = [X 1 Y]. We know how many ways there are to turn on the lights in [X 1] by using the base cases described above, and we can compute how many ways there are to turn on the lights in [1 Y] by applying our algorithm recursively.1) ways to turn on the lights where *k* is the number of zeros --- at each time step, we pick either to toggle on the leftmost 0 or the rightmost 0, giving us two choices at each time step exc

-- Tom T

**C. Polo the Penguin and Houses**

Draw an edge from a node to the one it points to. After thinking about the requirements you realize that the nodes {1, 2, … , k} must form a tree (ignoring the one that 1 points to, which must be one of {1, 2, …, k}). The remaining n-k nodes are totally unrestricted as long as they only point to something in the set {n-k, n-k+1, …, n}.

The set {1, 2, … ,k} forms an arbitrary labeled tree. The number of these (by Calay's formula) is k^(k-2). (If you don't know about Calay's formula you can use brute force to count this because k is at most 8.) Then 1 points to any of these k nodes. Finally the other nodes each have n-k possible labels. Thus the answer is

$$k^{(k-2)} * k * (n-k)^{(n-k)} \mod p$$

Where p = 10^9 + 7. So we just use the modular power algorithm, which can compute a^b mod p in O(log b) modular multiplications. Of course 64 bit arithmetic must be used to avoid overflow.

**D. Pluses everywhere**

Consider a sequence of digits xxx...xxdxxx. We can isolate the impact of the digit d on the final answer. The impact of d depends on where the next "+" occurs after d. In case it's d+xxx, then d contributes d * ((n-2) choose (k-1)) because there are n-2 places in which to put the remaining k-1 "+" signs, and for each way of doing this there's a contribution of d to the total. If the pattern is dx+xx, then the contribution is d*10*((n-3) choose (k-1)). And if the pattern is dxx+x then the contribution is d*10^2 * ((n-4) choose (k-1)). Finally if there is no

“+” after the d, then its contribution is 10^3 * ((n-4) choose k).

This leads to the following efficient algorithm. Define a recurrence for s(0), s(1), … as follows:

s(0) = 0
s(i+1) = s(i) + 10^i * ((n-2-i) choose (k-1))

With this in hand define a recurrence for c() as follows:

c(0) = 0
c(i+1) = c(i) + d(i) * [s(i) + 10^i * ((n-1-i) choose k)]

Where d(i) is the ith digit from the right (d(0) is the rightmost one). The desired answer is c(n).

But one more trick is needed. We need to be able to compute (a choose b) mod p for a and b up to about 10^6 efficiently. We do it based on the factorial formula for (a choose b).

(a choose b) = a! / ( (a-b)! * b!)

So we first build a table of factorials 1!, 2!, …, (10^6)! all modulo p. (Where p = 10^9 + 7). To divide by (a-b)! and b! We have to be able to compute inverses modulo p. This can be done either by making use of the Euclid’s extended GCD algorithm, or Fermat’s theorem. In this case:

a^(-1) = a^(p-2) modulo p

Using the efficient powering algorithm mentioned in problem c above this runs in time O(log p). Thus our algorithm is O(n log p).

--DS

**E. Gerald and Giant Chess**

The easiest way to solve this problem is to compute the number of paths from the starting point (1, 1) to the ending point (w, h), and then subtract all paths which cross over a black cell. This is efficient because the number of black cells is capped at 2000.

For starters, we need to know how to calculate the number of paths over a given x distance and y distance. Let xdist, ydist be the distances over x and y respectively. Then, the number of paths is (assuming xdist and ydist are both non-negative):

(xdist + ydist) choose (ydist)

Because we have (xdist + ydist) total moves to make, and we pick a certain amount to be down moves. If either variable is negative, the number of paths is 0 because there is no

possible way we can move up or left. Using this formula, we now know the total moves from (1, 1) to (w, h) as ((w + h - 2) choose (h - 1)).

All we need now is to know the number of moves which cross a black square. Our strategy for computing the number of moves is seeing how many moves lead us to the given black cell, times how many moves take us from the black square to the exit. We can use our choose formula for both of these tasks.

However, if we subtract this calculation for all black cells, we will be overcounting the paths which hit two or more black cells. To combat this, we sort the black squares by x value, then y value. We can then look at the ones which are closer to the end first and move backwards. At each step of the way, we solve for the number of ways to get to the exit without crossing any of the prior black squares that we have already processed.

More formally, for each black cell ‘i’ moving backwards from the exit, we compute the number of paths from this cell to the exit:

paths[i] = (w - blacks[i].x + h - blacks[i].y) choose (h - blacks[i].y)

And then subtract from this value the number of paths to each other black cell ‘j’ which we have already computed, times the number of paths from that black cell to the exit. This way, we remove paths which cross any further black cell than black cell ‘j’:

paths[i] = paths[i] - paths[j] * ((blacks[j].x - blacks[i].x + blacks[j].y - blacks[i].i) choose (blacks[j].y - blacks[i].y))

Once we have built up our table for the number of paths from each black cell to the exit, without crossing another black cell, we just do the same process but this time for the start, subtracting the value from the paths from the start to the end.

-- Cal L.

**F. Jigsaw Puzzle**

Regard each column as a "digit", where the digit can take on 64 values corresponding to each way that column can contain holes. (Assuming n=6 here.)

Now, given a sequence of digits d1, d2, ... di, what can we know about what is happening in column i+1? The answer is that there could be one of several "protrusion patterns" from column i into column i+1. Call such a set of protrusion patterns a pset. It can be represented as a 64 bit number.

So to count the number of feasible digit patterns, the dynamic programming state keeps a collection of psets, and for each one we keep a count of the number of digit patterns that leads to it.

Say a pset contains the 6 bit pattern 110011. Consider the second pattern 111111. Now I claim that if the pset contains the first, it must also contain the second. More generally, whenever the pattern contains two 0s in a row, it must also contain the same pattern with those two 0s replaced by 1s. This corresponds to the fact that you can put a vertical domino in there replacing the two 0s by two 1s. This is what I call normalization of the pset. Normalization will reduce the number of possible psets, which is good because there are 2^64 of them.

The transition from one state to the next is done as follows. We loop over all possible digits and all possible psets in column i. We then loop over each pattern in the pset. If a pattern is consistent with the digit, we take what is left after removing the digit positions and the pattern, and that is where we will put horizontal dominos to create a protrusion pattern into column i+1. In this way we construct a pset for column i+1, and (after normalizing it) we add the count for the current pset into the count for the new pset.

I don't know how to bound the number of psets that will actually occur. But this did not stop me from implementing and testing the algorithm. It turns out that the number of psets that actually get used in the n=6 case is only 60. This is counterintuitive. Richard said "Wow, I don't believe you. My guess was that it's between 10^3~10^5. Do you pass the data?". The answer is "yes" it does pass. Although the time limit is pretty tough for ocaml. So I built a pre-computed table for the n=6 case.

--DS