Code Jam 2014 - Round 1B

Analysis: New Lottery Game

In this problem, we are given three positive integers **A**, **B** and **K**. We are asked to calculate the number of pairs of integers (**a**, **b**) such that:

1. 0 â‰¤ a < A
2. 0 â‰¤ b < B
3. (a AND b) < K (denote AND means bitwise-AND of a and b).

We will refer to the set of valid pairs as **S(A, B, K)**, and the number of valid pairs as **f(A, B, K)**.

Solving the small dataset

For the small dataset, we can simply enumerate all pairs (a, b) that meet the first two constraints and count how many match the third constraint. Most programming languages have the built-in bitwise-AND operation, usually as the character â€œ&â€. This procedure will have a time complexity of O(**AB**), which is sufficient for the small dataset. Following is a sample implementation in Python 3:

```
def f(A, B, K):
  return len([(a, b) for a in range(A) for b in range(B) if (a & b) < K])
```

Solving the large dataset

We present two approaches for solving the large dataset in this problem. The first one is a standard Dynamic Programming solution (DP on digits/bits) that can be applied to different problems which are of a similar flavor. The second approach divides the search space into different sets, and computes the size of each set recursively.

First approach

Before delving into the actual solution, let us start thinking about a simpler problem: given an integer **M**, write a recursive function to count all non-negative integers **m** where 0 â‰¤ m < M. The answer is trivially M, but bear with us as our intent for this simpler problem is to build an intuition for the Dynamic Programming (DP) function which will be similar to the DP function we will write for the actual problem.

The core idea is based on counting the number of ways to generate the bits for the number m, such that they are always less than or equal to M but then we only count those that are strictly less than m. We can start by generating the bits of m from the most significant bit to the least significant bit, and for each bit position, generating only the feasible values. Note that the feasible values for the i-th bit can sometimes be both 0 and 1, or only 0.

Letâ€™s say M is 29 (which is 11101_2 in binary). The most significant bit is the bit at the fourth position and the least significant bit is at the zeroth position as shown here:

```
11101 (M = 29)
^^^^^
|||||
43210 (bit positions 4, 3, 2, 1, 0)
```

We will refer to the i-th bit as m_i. Letâ€™s say we have generated value 1 for m_4, value 0 for m_3, and we want to generate the feasible values for the bit at position i = 2. We can represent our current state as: 10cyy, where 1 and 0 are the values we have chosen, â€˜câ€™ is the current (i-th) bit we want to generate and â€˜yâ€™ denotes the bits we will try to generate in the future. We call

the bits to the left of position i as prefix. The prefix for the partially generated m_i = 10cyy is 10 (where i = 2). Now we explain the rules which help us decide the feasible values to use for the i-th bit c:

- Rule 1: we can always use value 0 for bit at position i.
- We can use value 1 for bit at position i if either:
  - Rule 2a: The prefix of the bits before position i in m (which is already generated) is less than the prefix of the bits before position i in M, or
  - Rule 2b: The i-th bit of M is 1.

Using the above rules, we can ensure that the prefix generated for m before position i will never be greater than the prefix of M before position i. Letâ€™s explore two scenarios which touch the above rules:

1. Letâ€™s say our current state is m = 10cyy as before. We want to determine candidate values for â€˜câ€™. In this case, the prefix is 10, and Mâ€™s prefix is 11 therefore both values 0 and 1 are feasible. Value 0 is always feasible (Rule 1) while from Rule 2a, value 1 is feasible.
2. If our current state is m = 11cyy, then both values 0 and 1 are feasible since value 0 is always feasible (Rule 1) while from Rule 2b, value 1 is feasible.
3. If our current state is m = 111cy, then only value 0 is feasible (Rule 1) but value 1 is not feasible since neither Rule 2a nor Rule 2b can be satisfied.

Note that as soon as the current prefix of m before position i is less than the prefix of M before position i, we can use both values 0 and 1 for the rest of the bit positions from i down to 0. Remember that we generate the bits from the most significant bit to the lowest (i.e., decreasing i).

We now proceed with describing the implementation. We define a recursive function count(i, lessM, M), where i is the i-th bit being generated and lessM is a boolean which denotes whether the prefix of m before position i is less than the prefix of M before position i.

As noted earlier, we start generating the number from the most significant bit (the leftmost bit) to the least significant bit (the rightmost bit). Therefore, the base case is when i is -1 which implies we have successfully constructed a whole number m. For the base case we return 1 if the generated number m is less than M otherwise we return 0 (since we only want to count such m that is strictly less than M).

We can do Dynamic Programming by caching (memoizing) on the parameters i and lessM and the result. Here is a sample implementation in Python 3 (note that lru_cache provides the required memoization):

```
from functools import lru_cache

def getBit(num, i):
  return (num >> i) & 1  # Returns the i-th bit value of num.

@lru_cache(maxsize = None)
def count(i, lessM, M):
  if i == -1:  # The base case.
    return lessM  # only count if it is strictly less than M.

  maxM = lessM or getBit(M, i) == 1

  res = count(i - 1, maxM, M)  # Value 0 is always feasible. See (1) below.

  if maxM: # Value 1 is feasible if maxM is true. See (2) below.
    res += count(i - 1, lessM, M)  # See (3) below.

  return res

# Prints how many non-negative numbers that are less than 123456789
print(count(31, False, 123456789))
```

Notes:
(1): To compute the boolean value of lessM for the next bit of m in the recurrence, we look at the value of the current lessM. If the current lessM is already true, then lessM for the next bit in the recurrence will also be true. Another case when lessM for the next bit is true is when the i-th bit of M is equal to 1. Since we pick value 0 for the current (i-th) bit in m and it is less than the i-th bit of M (which is 1), it means that lessM is true for the next bit. maxM captures what we described just now, therefore the next value for lessM for the next bit in the recursion is set to maxM.
(2): Value 1 is feasible if lessM is true (which means we are free to use both values 0 and 1) or the i-th bit of M is 1 (which means we are still generating feasible partial number m that is less than or equal to M).
(3): The value for lessM in the next bit can only be true if lessM is previously true. If the current lessM is false, then we know that the i-th bit of M is 1. Since we picked value 1 for the current bit, the next value for lessM will not change (since 1 is not less than 1).

Now, with the above intuition for generating non-negative numbers that are less than M, we can generalize it to count all possible pairs (a,b) that are less than A and B respectively and where the bitwise ANDing of the pair (a,b) is less than K. We enumerate all possible values for current bit in a and b (i.e. the 4 possible values (0, 0), (0, 1), (1, 0), (1, 1)) and add new constraints to ensure that the bitwise ANDing of the pair (a,b) < K.

The code for the original problem is of a similar style as in the simpler problem. The code is presented below. The purpose of the variable lessA is equivalent to lessM, similar for lessB and lessK. We try to generate all feasible values for a and b and keep k in check (see the following notes).

```
@lru_cache(maxsize = None)
def countPairs(i, lessA, lessB, lessK, A, B, K):
  if i == -1:  # The base case.
    return lessA and lessB and lessK  # Count those that are strictly less.

  maxA = lessA or getBit(A, i) == 1
  maxB = lessB or getBit(B, i) == 1
  maxK = lessK or getBit(K, i) == 1

  # Use value 0 for a, b, and k which is always possible. See (1).
  count = countPairs(i - 1, maxA, maxB, maxK, A, B, K)

  if maxA:  # Use value 1 for a, and 0 for b and k. See (2).
    count += countPairs(i - 1, lessA, maxB, maxK, A, B, K)

  if maxB:  # Use value 1 for b, and 0 for a and k. See (3)
    count += countPairs(i - 1, maxA, lessB, maxK, A, B, K)

  if maxA and maxB and maxK:  # Use value 1 for a, b, and k. See (4)
    count += countPairs(i - 1, lessA, lessB, lessK, A, B, K)

  return count
```

Notes:
(1): If we choose 0 for a and 0 for b, the value for k should be 0 since 0 & 0 = 0
(2): If we choose 1 for a and 0 for b, the value for k should be 0 since 0 & 1 = 0
(3): If we choose 0 for a and 1 for b, the value for k should be 0 since 1 & 0 = 0
(4): If we choose 1 for a and 1 for b, the value for k should be 1 since 1 & 1 = 1

To avoid overflows, you should take care to use 64-bit integers. Also, this solution pattern is a standard way to solve these kinds of problems and can be generalized to any number system (and not just base 2 as was the case in our problem).

The complexity of this solution is based on the size of the DP table which here is 31 * 2 * 2 * 2.

Second approach

We can group the pairs (a, b) in **S(A, B, K)** based on whether a and b are odd or even, i.e. have the least significant bit set. Each such pair will be accounted for in one of the four sets below (written using set-builder notation), so summing up the sizes of the four sets will give us the value **f(A, B, K)**.

1. {(a/2, b/2) | (a, b) âˆˆ S(A, B, K) && a even && b even}
   = S(ceil(A/2), ceil(B/2), ceil(K/2))
2. {(a/2, (b-1)/2) | (a, b) âˆˆ S(A, B, K) && a even && b odd}
   = S(ceil(A/2), floor(B/2), ceil(K/2))
3. {((a-1)/2, b/2) | (a, b) âˆˆ S(A, B, K) && a odd && b even}
   = S(floor(A/2), ceil(B/2), ceil(K/2))
4. {((a-1)/2, (b-1)/2) | (a, b) âˆˆ S(A, B, K) && a odd && b odd}
   = S(floor(A/2), floor(B/2), floor(K/2))

Note that in the 4 sets, the values for â€˜kâ€™ are forced. If â€˜aâ€™ or â€˜bâ€™ is even, â€˜kâ€™ will also be even while if both â€˜aâ€™ and â€˜bâ€™ are odd, then â€˜kâ€™ is also odd (because of the bitwise ANDing).

Let us provide more intuition on the 4 sets. We show it with an example where we list even numbers. If A is odd, e.g. 7 then the possible â€˜aâ€™ values that are even are 0, 2, 4, 6. If A is even, e.g. 8 then the possible â€˜aâ€™ values that are even are also 0, 2, 4, 6. The numbers in their binary representation are 000, 010, 100, 110. The 4 sets is akin to fixing (or eliminating) the least significant bit (i.e. by shifting right by 1, i.e. a>>1) which results in the set: 00,01,10,11 i.e. values 0, 1, 2, 3. The new value for A here is 4 (i.e. first integer greater than 0, 1, 2, 3). We generalize and say the following: If we are considering even values for â€˜aâ€™, then the new A is given by (A+1)>>1 (which is also ceil(A/2)). By going through a similar exercise for when â€˜aâ€™ is odd and A is odd or even (which we leave to the reader), we find the new A is given by A>>1 (which is also floor(A/2)).

The 4 sets give us a simple recursive procedure to compute **f(A, B, K)**, as the sizes of the sets on the right hand side of the equations are simply calls to f. The appropriate base cases here are **f(1, 1, K)** = 1 and **f(A, B, K)** = 0 if any of A, B, K are equal to zero. To turn this recursive procedure into an efficient algorithm, we can memoize computed values. This means that after computing some **f(A, B, K)**, we cache the result so future computations with the same values will take constant time.

Sample implementation in Python 3:

```
@lru_cache(maxsize = None)
def f(A, B, K):
    if A == 0 or B == 0 or K == 0:
        return 0
    if A == B == 1:
        return 1
    return f((A+1)>>1, (B+1)>>1, (K+1)>>1) + \
           f((A+1)>>1, B>>1, (K+1)>>1) + \
           f(A>>1, (B+1)>>1, (K+1)>>1) + \
           f(A>>1, B>>1, K>>1)
```

Complexity Analysis

It can be shown by induction that a recursive call **f(Aâ€™, Bâ€™, Kâ€™)** of recursive depth **n** has Aâ€™ equal to either floor(A / 2^n) or 1+floor(A / 2^n), and that Bâ€™ and Kâ€™ satisfy similar relations. From this, we can see that at a given recursive depth there are at most 8 different calls to f. Due to our chosen base cases, we can also see that the maximum recursion depth is O(**log(max(A, B))**). We conclude that the algorithm is O(**log(max(A, B))**).