Main Solution

Attempt 3, algorithm ignoring closed form formula. Apparently, I would not easily give up, and with some extra hints, I was able to figure our an algorithm and test it on the recurrence relations (*), (**), and (***). The algorithm will take $\mathcal{O}(kn2^k)$ time and space, where $k < n$.

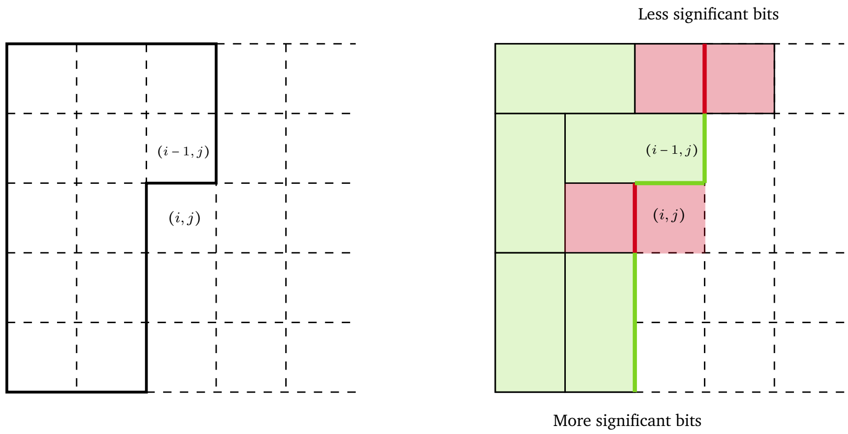
```python
def i_dont_know_how_to_count(k, n):
    if k * n % 2 == 1: return 0

    # create dp[][][], dimension k * n * 2^(n+1)
    # instead of dealing with 0-indexed arrays, I decide to simply pad it by making dp[][][] slightly larger. Index problems begone!
    # main idea: traverse through the hallway, and for each tile, keep track of all possible tilings of all previous tiles such that
        # (i) no 2x1 rectangles overlap
        # (ii) all previous tiles are covered, and
        # (iii) it does not matter if some 2x1 rectangles cover to-be-explored tiles, but they must lie within the hallway, i.e., not ouside the hallway's boundary
    # detailed explanation (e.g. choice of mask and update rules) below the algorithm

    max_mask = (1 << (k+1)) - 1
    dp = [[[0 for _ in range(1 << (k+2))] for _ in range(n+1)] for _ in range(k+1)] # slightly extra spaces
    dp[k][0][0] = 1 # base case: vacuously one way to tile a non-existent hallway

    for j in range(1, n+1):
        # the first for loop will execute every time we move on to a new column. In particular, it will carry all tiling information from tile[k][j-1] to tile[0][j] and prepare it for the first tile on column j, namely, tile[1][j] (recall 1-indexed arrays)
        for mask in range(max_mask+1):
            dp[0][j][mask << 1] += dp[k][j-1][mask]

        # this is how we traverse through most of the hallway, and the bulk of our DP algorithm. Main idea: decide whether and/or how to place the next rectangle given the current configuration, which is jointly decided by coordinates (i,j), and the mask.
        for i in range(1,k+1):
            for mask in range(max_mask+1):
                NE, SW = mask & (1 << (i-1)), mask & (1 << i)
                if NE and SW: continue
                if NE and not SW: dp[i][j][mask ^ (1 << (i-1))] += dp[i-1][j][mask]
                elif SW and not NE: dp[i][j][mask ^ (1 << i)] += dp[i-1][j][mask]
                else:
                    dp[i][j][mask ^ (1 << (i-1))] += dp[i-1][j][mask]
                    dp[i][j][mask ^ (1 << i)] += dp[i-1][j][mask]

    return dp[k][n][0] # done!
```

First, a graphical representation of how I traverse through each tile, what data I store, and how I define the mask. See below.

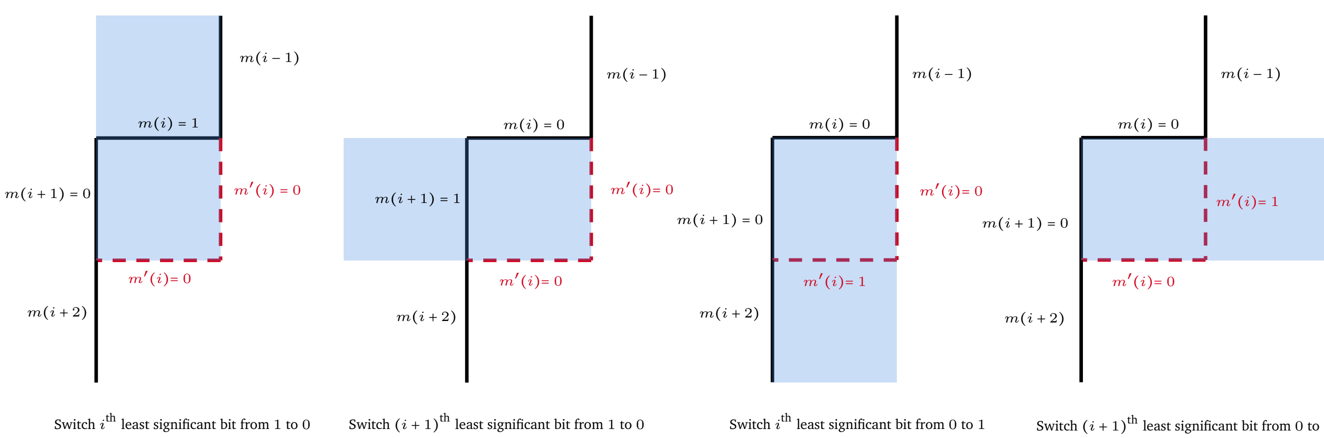
- At state (i, j), we will have traversed through the first $j-1$ full columns, along with the first $i-1$ tiles on the j^{th} column. Recall that everything is 1-indexed. On the figure below, $(i, j) = (3, 3)$.
- The particular boundary of interest is colored in green or red. In particular, it starts at the SE vertex of tile

$(k, j-1)$ and ends at the NE vertex of tile $(1, j)$. It is immedaite that this path has length $k+1$. It includes the west and north edges of tile (i, j).

- To define a mask corresponding to this particular tiling, we note that some part of this path has a 2×1 rectangular laid over it, whereas the other part does not. The former is colored in red and the latter in green.
- Naturally, we define our mask to be a $(k+1)$-bit binary number, with 1's corresponding to red and 0's to green, with the north endpoint being least significant and south most. It is clear that different masks correspond to different tilings. We use `dp[i][j][mask]` to store the number of computed tilings up to this point. For a more formal definition, this is the number of tilings such that at least at least half of each 2×1 rectangle is inside the "explored area."
- For example, the path and the tiling in the figure below correspond to the mask $(001001)_2$.



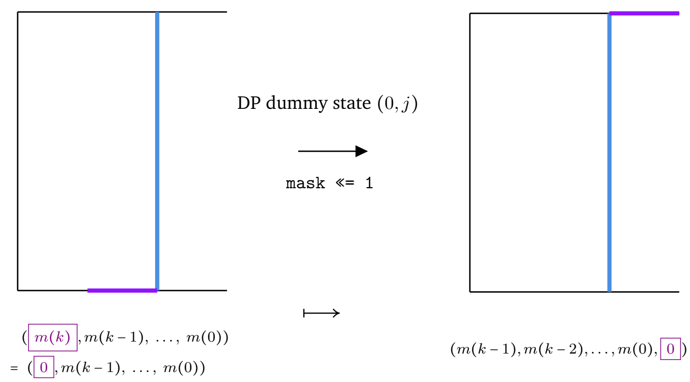
Now, the update rules for (i, j) with $i \neq k$ (i.e., not changing columns). For convenience, we denote the digits of $(k+1)$-bit mask by $m(k), m(k-1), \dots, m(0)$, with $m(0)$ the least significant.



- We use variables `NW` and `SE` to indicate whether $m(i), m(i+1) = 1$, respectively. To do so, we need to check if the i^{th} (resp. $(i+1)^{\text{th}}$) least significant bits of `mask` is nonzero, as shown in line 24 using bitwise `AND` and left shift.

- It is impossible to have $m(i) = m(i+1) = 1$, as this implies tile (i, j) is both covered by a horitonzal rectangular tile (with $(i, j-1)$) and a vertical one (with $(i-1, j)$ assuming $j \geqslant 2$).
- If $m(i) = 1$ and $m(i+1) = 0$, this means tiles (i, j) is covered by a retangular tile along with $(i-1, j)$. Updating the boundary, replacing black $m(i), m(i+1)$ by the red dashed ones, we simply set $m'(i) = 0$, using bitwise `XOR` and left shift. We copy `dp[i-1][j][mask]` into `dp[i][j][new_mask]`, as the number of tilings corresponding to the new mask does not change.
- The case $m(i) = 0$ and $m(i+1) = 1$ is analogous.
- If $m(i) = m(i+1) = 0$, then tile (i, j) is not yet occupied. We can either place a horizontal rectangle with (i, j) being the left one, or a vertical one with (i, j) on top. Correspondingly, we either change modify $m'(i)$ or $m'(i+1)$, using almost identical techniques as before. We copy `dp[i-1][j][mask]` into both `dp[i][j][mask_new_horizontal]` and `dp[i][j][mask_new_vertical]`.

Finally, to switch to a new row, we note that the new mask is simply old mask with a right shift, as the two purple edges always corresponde to 0 — we allowed tilings to hit unexploreded areas, but *not* outside our hallway!



It should be clear that this algorithm has runtime $\mathcal{O}(n+1)(k+1)2^{(k+2)} = \mathcal{O}(nk2^k)$. I am, obviously, hoping that it is correct, so I plugged in value pairs $(k, n) \in \{2, 3, 4\} \times \{1, 2, \ldots, 10\}$ and confirmed that they agree with (*), (**), and (***) in solution attempt #1.

	$n = 1$	2	3	4	5	6	7	8	9	10
$k = 2$	1	2	3	5	8	13	21	34	55	89
$k = 3$	0	3	0	11	0	41	0	153	0	571
$k = 4$	1	5	11	36	95	281	781	2245	6336	18061

So yes, either I screwed up the entire thing, or with probability $1 - \epsilon$ this algorithm is valid.