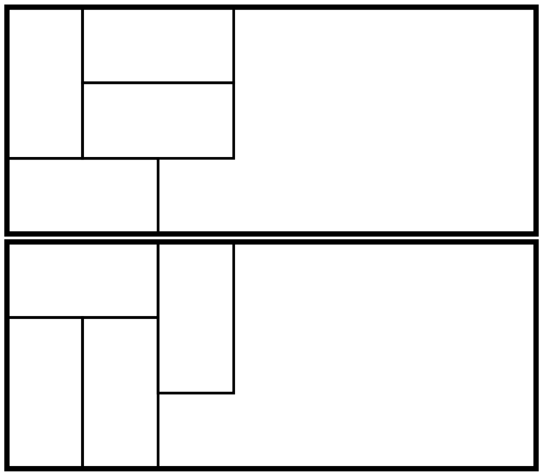
6.4.7 Grid Tiling

Consider the following problem:

> You have a pack of 1×2 tiles and a $3 \times n$ sized grid. How many ways are there to completely tile the grid?

Trying to recursively enumerate all possibilities is too slow for any non-trivial n, so where is the repeat work? Consider the following two parial tilings:



The aggregate space covered by the tiles is identical in shape and size. As such, the number of ways to tile the remaining, unfilled portion of the grid must also be the same. Therefore if we have seen a particular partially tiled "shape" more than once, we need not compute the rest of the grid a second time. The intuition in this case is very similar to TSP, whereby if we iterate through states in a clever order, we can encode the number of ways to reach that state and only traverse the remainder of the state space once.

This leaves the question: What is the clever order? In the case of TSP we added one node at a time from the set of nodes we hadn't visited. In grid tiling, we will add one column at a time. This isn't completely sufficient, however. Consider adding tiles to the n-th column, many of the possible ways to do so will occupy space in the $n+1$-th column. To cope with this, we augment our state to include both the column we have last filled and the rows which have currently spilled over into the next column. When considering that next column, we look at all possibilities of how the previous column could have spilled over, and for each, only consider ways to complete that column which do not conflict with the spillover.

Let's see a couple of examples:

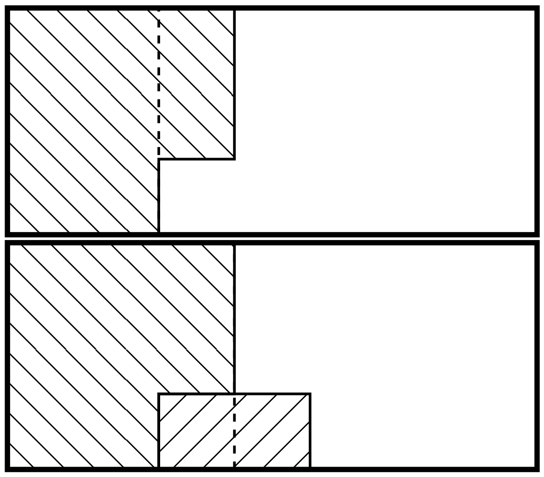


Figure 6.38: When processing the example from the beginning of the section, we have completely filled up through column 2, and have spillover in the top two rows. There is only a single way to complete the next column by placing a block in the lower row. This particular situation leaves column 3 completed with spillover in the bottom row.

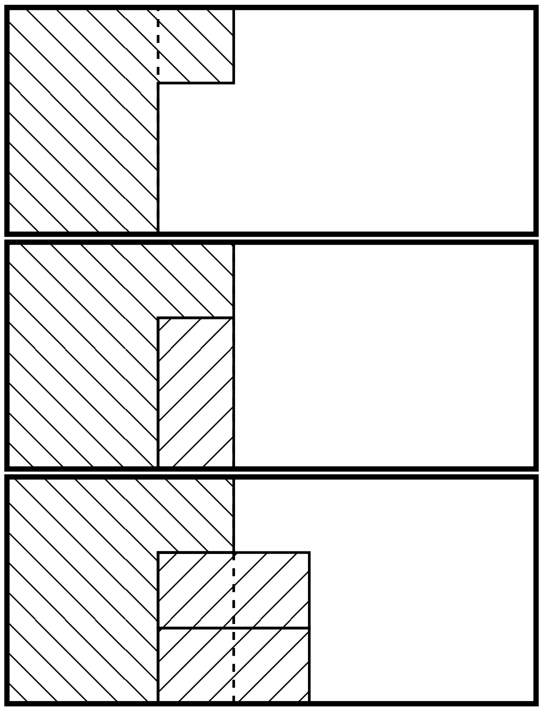


Figure 6.39: In a slightly different example where we have completed column 2 with spillover only in the top row, there are two ways to completed column 3. One with no spillover, and one with spillover in the bottom two rows.

Enumerating the ways to complete a column for all possible spillover situations, we can go column by column, look at each possible spillover, and sum up the ways we can complete the next column.

1. Indices: How many columns we have completed, and what rows have spillover.
2. Value: The count of distinct ways to reach that combination of column and spillover.
3. Relation: Given the spillover, each distinct way we complete the next column is summed into the appropriate index for the next column and spillover.[68]
4. Iteration Order: As each column contributes to the subsequent column, we iterate in column order.
5. Base Cases: We start with 0 ways to reach any possible state, except having completed 0 columns with no spillover, which has 1 possible way.

[68]Note this is a forward looking relation.

In order to actually build the code, we look at each possible spillover situation and enumerate all possibilities. In order to cleanly map spillovers we use a bitmask. 0 will represent no spillover in a given row, while 1 represents spillover. The LSB will be considered the top row.

- 000: No Spillover

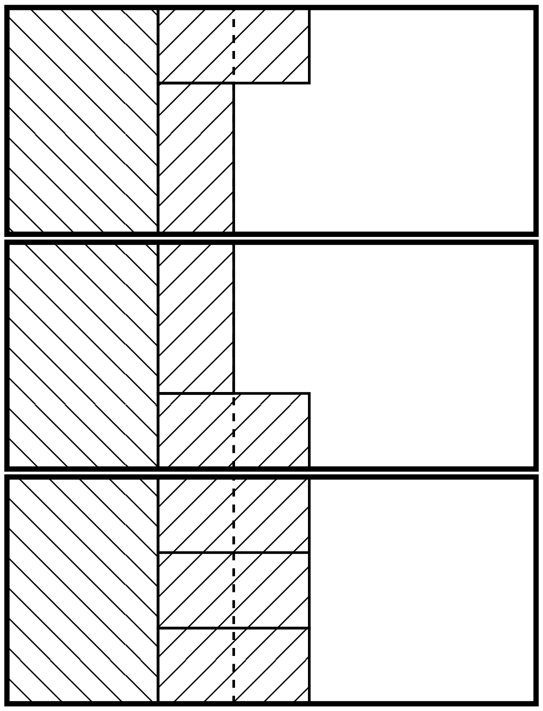


Figure 6.40: There are 3 ways to complete the next column with spillover 000, reaching column 3 with spillover 001, 100, and 111 respectively.

- 001: Spillover only in the top row. This is the earlier seen example. There are two ways to complete the column, either by placing a vertical tile leading to spillover of 000, or placing two horizontal tiles, leading to spillover in the other two rows (110).

- 010: Spillover only in the middle row[69]

[69]Astute readers will note this is not actually a reachable state.

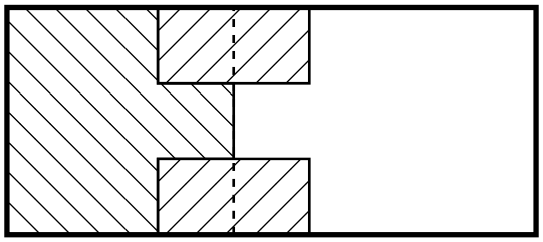


Figure 6.41: There is only 1 way to complete this column, leading to spillover in the top and bottom row (101).

- 011: Spillover in the top two rows. This is the other earlier seen example. There is one way to complete this column by placing a single horizontal tile in the bottom row, leading to spillover in that row (100).
- 100: Spillover in the bottom row. This mirrors the earlier seen example with two ways to complete leading to spillover of 000, or 011.
- 101: Spillover in the top and bottom row[70]

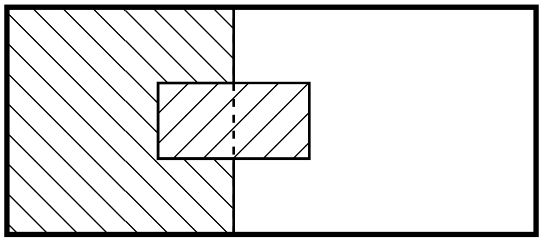


Figure 6.42: There is only 1 way to complete this column, leading to spillover in the middle row (010).

- 110: Spillover in the bottom two rows. This mirrors the earlier seen example, with a single way to complete leading to spillover of 001.
- 111: Spillover in all rows. This is a bit of a special case, as the next column is already completed. There are no tiles which need to be added and we simply need to add the count into the subsequent column with no spillover.

With all transitions determined, we can execute the code directly.

```
long long dp[c+1][8]={}; //columns and shapes
dp[0][0]=1; //base case
for(int i=0;i<c;i++){
    //3 ways to do 000
```

[70] Astute readers will note this is not actually a reachable state.

```
    dp[i+1][0b100]+=dp[i][0b000]; //horizontal over a vertical
    dp[i+1][0b111]+=dp[i][0b000]; //3 horizontal
    dp[i+1][0b001]+=dp[i][0b000]; //vertical over a horizontal
    //2 ways to do 001
    dp[i+1][0b000]+=dp[i][0b001]; //single vertical
    dp[i+1][0b110]+=dp[i][0b001]; //2 horizontal
    //1 way to do 010
    dp[i+1][0b101]+=dp[i][0b010];
    //1 way to do 011
    dp[i+1][0b100]+=dp[i][0b011];
    //2 ways to do 100
    dp[i+1][0b000]+=dp[i][0b100]; //single vertical
    dp[i+1][0b011]+=dp[i][0b100]; //2 horizontal
    //1 way to do 101
    dp[i+1][0b010]+=dp[i][0b101];
    //1 way to do 110
    dp[i+1][0b001]+=dp[i][0b110];
    //1 way to do 111 (placing no tiles)
    dp[i+1][0b000]+=dp[i][0b111];
}

//answer is dp[c][0]
//completed c columns with 0 spillover
```

The number of states include c columns and 2^r spillover situations. There are a constant number of transitions, leading to an overall runtime of $O(c * 2^r)$. While it may seem like this allows us to solve for a large number of columns (so long as the number of rows is < 20), the limiting factor is not the runtime but our ability to manually encode all possible ways to complete a given column. The number of ways to do so is exponential in r, so while this amortizes to a constant number per given DP state, it becomes unweildy to code for any non-trivial r. We have shown $r = 3$, can definitely do $r = 2$, likely can do $r = 4$ in a pinch, and $r = 5$ would probably be painful. The application of this technique is therefore limited to a very small number of rows.

This technique can be extended is slightly different tile shapes. So long as the available tile shapes and orientations can only ever create spillover of a single column, we can compute the appropriate transitions.[71]

6.4.7.1 Broken Front DP

The main cause of the difficulty in the above solution is we considered how to complete an entire column at once. If we could reduce this to looking at how to complete only a single cell at a time, it might be more weildy.

Suppose we consider a single cell, where does the previous DP design fail? Our DP state assumes we have completed columns up to some i, and had

[71]This opens up L-shaped tiles and others, so long as they are oriented to only span two columns.

spillover of at most 1 in each row. Suppose we had some intermediate state with no spillover after column i and were attempting to fill just the top cell. There would be two ways to do this: placing a tile horizontally or vertically. Placing vertically yields no issue, as we end up with spillover in the top two rows over the previously completed column 2. However, attempting to place the tile horizontally leads to a situation where we have spilled two columns past the last completed column, a state we have no ability to encode.

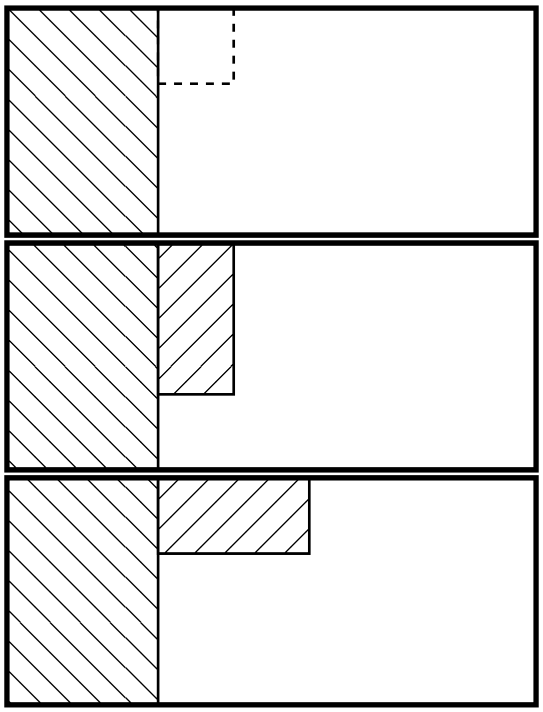


Figure 6.43: With 2 columns completed, we try to complete the first cell of column 3. Covering this cell with a vertical tile maps to 2 completed columns with overflow in the top to rows, but placing a horizontal tile has no valid encoding.

Due to the 2×1 size of the tiles, when we complete an entire column with the naive method we have a strong guarantee there is only ever be a difference of 1 in the most filled column in every row. When we move to a cell-first method, this assumption is broken. The key point is after we fill a given cell in column i, we might reach all the way into column $i+2$. The yet-unprocessed rows in that column might still only be filled up until column i, leading to the difference of 2.

We can address this by modifying our definition of spillover. Instead of computing the spillover as whether a block extends further right of the last completed column, we will compute it based on the last processed cell in its row. This means the column used as the reference for spillover changes on a row by row basis. We'll refer to the reference column for each row as its baseline. Let's see some examples on a 4-row grid where we have processed 2 cells in the fourth column so far.

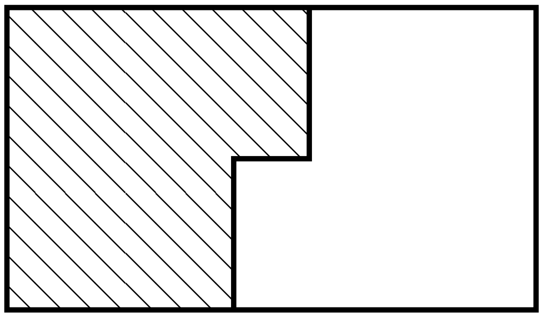


Figure 6.44: We have already processed the top two cells in the fourth column, but not the bottom two. No blocks spillover past the processed cells, so the spillover is 0000.

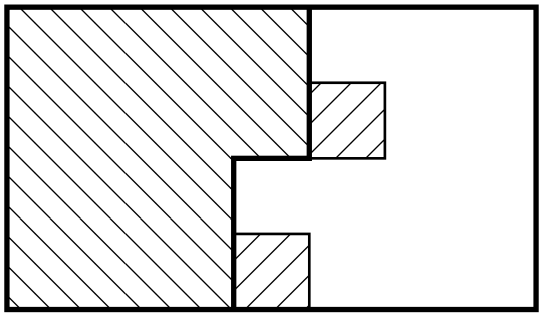


Figure 6.45: One cell sticks out past the processed cells in the bottom row, and one in the second from the top. This leads to a spillover of 1010. Note that the number of cells processed in each row differs, so the baseline for the top two rows is one column further than the baseline for the bottom two rows. This explains why the spilt-over blocks are in two different columns.

When we were processing an entire column at a time, we had to enumerate all possible ways to fill a given column given the spillover. When we are processing a single cell at a time, we must only enumerate the ways to fill that given cell. There are exactly 3:

1. The cell is already filled due to spillover, in which case we do not need to place a tile. The resulting spillover is the same in all rows but the one containing the just-processed cell. As that cell becomes processed, the baseline used to compute the spillover shifted one column to the right. As

no tile was placed and the base changed, there is no longer spillover in that column.

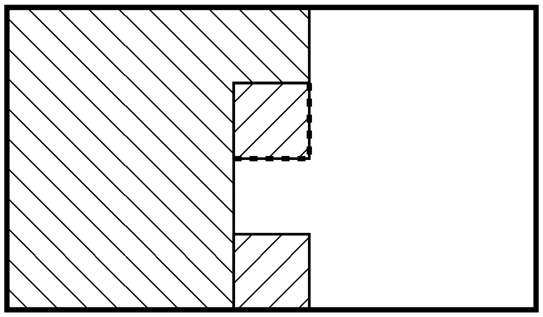


Figure 6.46: We are trying to fill column 3 row 1 (dashed box) with spillover 1010 (boxes with lines going in the alternate direction). As there was spillover into the box we are trying to fill, there is no tile we can place. As the baseline in this row shifts by one column after we process it, it results in spillover of 1000.

2. The cell was not previously filled and we placed a horizontal tile. The spillover does not change in any row but the one we just processed. The baseline moved right by 1, but we placed a tile of width two, meaning there is now spillover in this row.

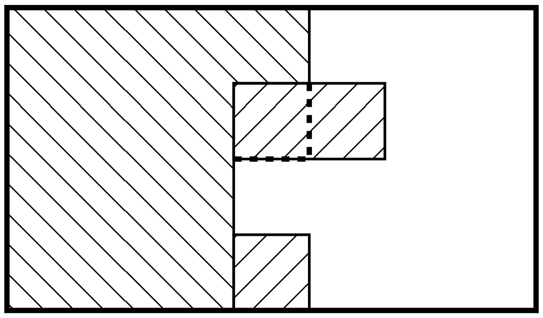


Figure 6.47: We are trying to fill column 3 row 1 (dashed box) with spillover 1000. When we place the new horizontal tile, it spills over past the dashed line, leading to a new spillover of 1010.

3. The cell was not previously filled and we placed a vertical tile. For this to be possible, we must not have had spillover in the row below the processed cell. The baseline in the row we just processed moved by 1, and we added 1 cell worth of tile, so its spillover does not change. On the other hand, we covered a cell in the row below. Its baseline did not change but now has another cell filled, and therefore there is spillover in the row below.

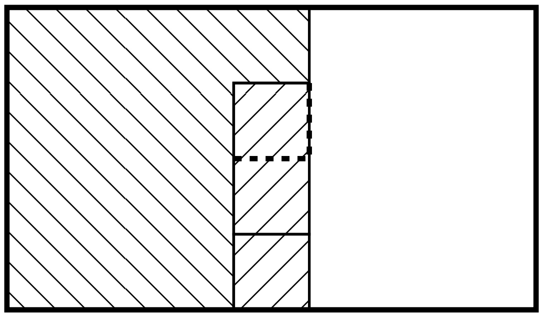


Figure 6.48: We are trying to fill column 3 row 1 (dashed box) with spillover 1000. When we place the new vertical tile, it does not spill over in row 1, but does create spillover in row 2, leading to a new spillover of 1100. Note that we would not have taken this option if there was already spillover in the row below the one we're examining or if we are the bottom row.

With this understanding, we can define the components of the DP, which due to the fact that the baseline differs between rows as we process cells, is known as DP on a *broken front*:

1. Indices: Which cells (column and row) have been completed, and what is the spillover relative to the last completed cell in each row.[72]
2. Value: The count of distinct ways to reach that combination of row,column and spillover.
3. Relation: Given the spillover, each distinct way we complete the next cell is summed into the appropriate index for the next cell and spillover.
4. Iteration Order: As each cell is contributing only contributes to the spillover in its own row or the below row, we iterate through cells by row then column.
5. Base Cases: We start with 0 ways to reach any possible state, excpet having completed no cells with no spillover, which has 1 possible way.

The number of DP states is $r * c * 2^r$, and the number of transitions per row is both constant, and exactly 3 regardless of the number of rows.[73] The runtime is therefore $O(r * c * 2^r)$. This enables a number of rows of about 20, as opposed to the 3 or 4 we could handle with the naive method.

There are some implementation complexities which must be noted before the code is shown.

[72] i.e. the baseline

[73] unlike the naive method, which incurred a growing set of transitions as rows grew

- It may not be obvious from the relation, but the processing of each cell only ever updates the next cell (with potentially a different spillover). As such, we only store two rows of the DP table,[74] one for the current cell and one for the next, with each possible spillover. This is critical as with so many DP states, trying to store the whole table might exceed available memory.

- We can process such large grids, so it is likely to overflow most integer widths. As such, the result is almost universally computed under some modulus.

- While the forward-looking implementation is supplied here for comprehensibility, the backward-looking implementation is quite a bit more compact. It is provided in the appendix.

```
int dp[2][1<<r];//only depend on previous state, so compress the DP and
    store spillover
for(int i=0;i<2;i++)for(int j=0;j<(1<<r);j++)dp[i][j]=0;
int next=1,current=0;
dp[current][0]=1;
for(int col=0;col<c;col++)for(int row=0;row<r;row++){
   fill(dp[next],dp[next]+(1<<r),0);
   for(int spill=0;spill<(1<<r);spill++){
      if(spill&(1<<row)){
         dp[next][spill&~(1<<row)]+=dp[current][spill];//place no tile,
             already spilt
         dp[next][spill&~(1<<row)]%=MOD;
      }else{
         dp[next][spill|(1<<row)]+=dp[current][spill];//place horizontal
             tile
         dp[next][spill|(1<<row)]%=MOD;
         if(row<r-1&&!(spill&(1<<(row+1)))){
            dp[next][spill|(1<<(row+1))]+=dp[current][spill];//place
                vertical tile
            dp[next][spill|(1<<(row+1))]%=MOD;
         }
      }
   }
   swap(next,current);
}

//result in dp[current][0]
```

[74] as we did with knapsack DP