CS 4820, Spring 2018 **Homework 3, Problem 3**
Name: Rongguang Wang
NetID: rw564
Collaborators: Siyuao Liu (sl2928); Yihao Chen (yc2288)

(3) *(10 points)*
In the TROMINO TILING problem one is given a subset of a square grid, and one must decide whether the given subset can be tiled (i.e., covered without overlaps) by L-shaped trominoes.

The input to the problem can be given in the form of a $k \times n$ array of 0's and 1's. Denoting this array by A, the entry $A[i, j]$ equals 1 if the grid cell in location (i, j) belongs to the subset to be tiled, and if not then $A[i, j] = 0$.

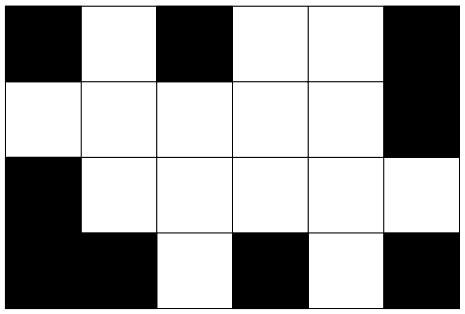
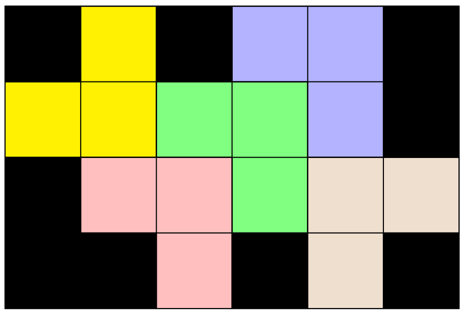
Design an algorithm to solve TROMINO TILING in time $O(n) \cdot 2^{O(k)}$. Your algorithm only needs to output a simple "yes" or "no" answer indicating whether or not the given subset of the grid can be tiled by L-shaped trominoes. In describing your algorithm, you may assume you have access to a subroutine called WIDTHTWO that solves the case when n (the width of the grid) is equal to 2, and k (the grid's height) is an arbitrary positive integer. The running time of the WIDTHTWO subroutine is $O(k)$. You are *not* responsible for designing or analyzing the WIDTHTWO subroutine; you are welcome to just assume it exists and treat it as a "black box."

Note that the running time of the algorithm you are being asked to design is **not polynomial in the input size**. It is linear in n but exponential in k. Partial credit will be awarded for algorithms whose running time depends super-exponentially on k, or has super-linear but still polynomial dependence on n. No credit will be awarded for algorithms whose running time is exponential in n.

Example. In the following example $k = 4$ and $n = 6$, and the matrix A is given by

$$A = \begin{pmatrix} 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 \end{pmatrix}.$$

The region to be tiled is the set of white squares in the left figure below. The right figure illustrates a tromino tiling of the region.

Solution:

The problem is aiming to find an algorithm which can tile L-shaped Tronimos in to the blank spaces of a $k \times n$ matrix represented using 0s and 1s where 0 means filled (black) and 1 means blank space (white). The algorithm should output true or false corresponding to the square grid can be tiled or not. The time complexity is required to be bounded by $O(n) \cdot 2^{O(k)}$ where n is column number and k is the row number of the square grid (matrix). The dynamic programming algorithm can be used to solve this problem. Since every L-shaped Tromino occupied 2 columns, the basic idea to solve this problem is trying out every possible tiling on 2 adjacent columns. Record the result (true or false) of tiling for each row in array, which can be called dynamic programming loop-up table (DPT). As the situation of each small square are represented using 0 and 1, which is binary number, and there are k rows for each column, the total number of combination of filled state and blank state for each column is 2^k. The index of the DPT array should be the decimal representation of the binary digits which represents the states of each small square in the column and the value stored in the array should be 0 or 1 where 0 represents false which means that this situation can not be tiled and 1 represents true which means that this situation can be tiled. To build the DPT for the $k \times n$ matrix, $2^{k \cdot n}$ combinations should be stored which leads to the time complexity of $2^{O(k \cdot n)}$. This exceed the upper bound requirement of the problem. Therefore, the DPT should not build for the whole grid at the same time. A refinement is that only create a DPT for 2 columns since each L-shaped Tronimo can only occupy 2 column at a time. Then, the number of the combination is reduced to 2^{2k} and the time complexity is reduced to $2^{O(k)}$ which meets the requirement.

Since this problem can be solved by using dynamic programming algorithm, there should be a recursive function used to tile the grid. The grid table can be filled starting from the last 2 columns c_{n-1} and c_n. Each time only the last column c_n of the restricted 2 columns should be fully tiled with no blank space left. Then, move leftward for one column step to columns c_{n-2} and c_{n-1}. Now, these 2 columns are treated as the restricted columns block (RCB). The DPT should be created for each restricted columns block ($k \times 2$ grid). The indexes, which is the different combinations of k digits 0 and 1, represent the occupation situation of each small square of one column and the values stored in the array is Boolean value (true or false), which represents whether the 2 columns can be tiled. The DPT for the second column c_{n-1} is built based on the situation of the last column c_n. As the RCB moves leftward with step size of one column, a recursive function Tronimo Tiling (TT) should be called each time. The input of the TT function is the decimal representation of the occupation situation (k digits binary number) of one column to be tested. In the function, the input should be checked by the DPT. Once the output is false, the grid cannot be tiled. If the DPT returns true, the array M which stores the occupation situation of the matrix should be updated according to the tiled result in this RCB. Then, a new DPT should be created for next RCB and the function TT should be called again. Once the recursive function TT runs $n-1$ times and all outputs are true, then, the grid can be tiled.

Algorithm:
1) Initialize an array $M[k] (k = 1, \dots, n)$ and assign the decimal representation of each column binary digits to the array
1) Create DPT for the last 2 columns c_n and c_{n-1} according to $M[k]$

2) Call $TT(k)$
$TT(k)$:

```
if k = 0, return true;
if k > 0
    if DPT(M[k]) = true
        Update M[k];
        Create DPT for next RCB c_{k-1} and c_k;
        Call TT(k-1);
     else
        return false;
end if
```

Algorithm's Correctness:

An exhaustive case analysis justifying correctness was already presented above.

Running Time Analysis:

The time cost in this algorithm is mainly caused by creating the DPT recursively. The time spend on creating a DPT for 2 columns is $2^{O(k)}$ and there are n iterations in the TT function. Therefore, the total time spend in this algorithm is $O(n) \cdot 2^{O(k)}$.