Caution: do **not** appeal to nondeterminism anywhere in this assignment. Any problem whose solution relies on nondeterminism will recieve **no credit**.

Problem 1 (150 pts)**.**

Let $SPLIT_{DECISION} = \{\langle A, B, C\rangle : A, B,$ and C are deciders and $L(A) = L(B) \cup L(C)\}$.

Show that $SPLIT_{DECISION}$ is decidable.

Note: You may assume that decidable languages are closed under union and complement. If you wish to claim closure of other set operations, you should prove them. Also note that closure under union (I have two decidable languages and their union must be decidable) is not alone sufficient to prove this statement (I have three decidable languages, and I can decide if one is the union of the other two or not).

Problem 2 (150 pts)**.**

For a given constant c and TM M, show that the following problem is decidable:

M makes at most c moves for all $w \in \Sigma^*$.

Note: Assume that 'stay' is not a tape head option.

Problem 3 (200 pts)**.**

- Define $L_1 = \{\langle M\rangle : M$ is a TM where $|L(M)| = 1\}$. Show that L_1 is undecidable:
- Let a constant $k \geq 0$ be given. Define $L_k = \{\langle M, k\rangle : M$ is a TM where $|L(M)| \leq k\}$. Show that L_k is undecidable:

Problem 4 (Up to 600 bonus pts)**.**

- **(+450 pts)** First, complete the coding assignment below and then submit your completed code via the myCourses submission portal for HW 6 alongside the rest of the assignment.
- **(+150 pts)** Second, watch this video and then take the associated brief quiz to be provided on myCourses.

Synopsis:

In this project, starter code has been provided for you in myCourses, under the filename `hw-6-4.py`. As the name suggests, this is written in Python, and you are expected to flesh it out as per the subsequent specifications. Some functions have been provided to you. The first is `draw_square`. This function refers a square using the Python turtlegraphics package. You will be responsible for writing three functions:

- `generate_blank_grid`
  - **Input**: `dim`, a positive integer representing the length of the columns and rows
  - **Output**: a (`dim` x `dim`) 2-D matrix of boolean values all set to False
    - Make sure to generate a new list inside your function and return it. It needs to persist outside the function call.
- `modify_grid`
  - **Input**: `Grid_In`, a 2-D matrix of boolean values.
  - **Output**: `Grid_Out`, the 2-D matrix of boolean values subject to the following:
    - Define the neighbors of `Grid_In[i][j]` to be the (at most) eight elements of `Grid_In` immediately adjacent to but not including `Grid_In[i][j]`, including diagonally adjacent cells.
    - Then for each matrix element `Grid_In[i][j]`, we have
      - If `Grid_In[i][j] == True` and 2 **or** 3 neighbors of `Grid_In[i][j]` are `True`, then we set `Grid_Out[i][j] = True`
      - If `Grid_In[i][j] == False` and **exactly** 3 neighbors of `Grid_In[i][j]` are `True`, then we set `Grid_Out[i][j] = True`
      - Otherwise, we set `Grid_Out[i][j] = False`
- `draw_grid`
  - **Output**: screen rendering of Grid with entries filled only if corresponding boolean entry is true
  - **Input**:
    - `t`, a turtlegraphics variable.
    - `x_home`, an integer referencing the x-coordinate of the top-left corner of the bottom-left square to be drawn.
    - `y_home`, an integer referencing the y-coordinate of the top-left corner of the bottom-left square to be drawn.
      - Note the vertical inversion - we draw bottom up
    - `side_len`, a positive integer referencing the length of the sides of all (individual, small) squares.
    - `gap_len`, a positive integer referencing the length of the gap between (individual, small) squares.
    - `grid_dim`, a positive integer representing the length of the columns and rows (the number of squares to be drawn in each column and row).

        * `Grid`, a 2-D matrix of boolean values.
            - Note that Grid can also be conceived as a list of lists, with each sublist corresponding to a grid column
            - Note that each square (i,j) in the matrix should be colored (black) only if `Grid[i][j] == True`
        * Note that the provided `draw_square` can be called as a subroutine

The remaining functions provided to you are as follows, and the suggested input is preset in the provided code:

- `add_object_1`
- `add_object_2`
- `add_object_3`

After writing the functions listed above

- For each of the `add_object_*` functions, in the order they appear:
    - Call (uncomment) the `add_object_*` function with the input as given, to place an 'object' into the grid.
        * Make sure to only uncomment one `add_object_*` function call at a time. The preceding `generate_blank_grid` call needs to ocur immediately before embedding the object, and only one object at a time should be embedded.
    - Run the provided code to iterate `modify_grid` 512 times and observe the behavior.
    - Notice that the initial grid and embedded object(s) completely determine everything that happens under iteration of this function.
- Note: If the image does not fit on your screen, you can try using different values for `x_pos` and `y_pos` in order to move the image around. It may also be helpful to maximize the turtlegraphics screen once it pops up. If all else fails, try reducing `side_len` a little bit to shrink the output, but do not change `grid_side_len`.
- When you are sure everything is working correctly, submit your modified source code in the HW 5 submission portal in myCourses.

Example Output (static):
Note that a single instance of the output of `draw_grid()` may look something like: