Work with your neighbor. (This will be graded for participation only.)

1. We have written the recursive function version of `sumlist(L)` that returns the sum of the elements in `L`. Re-write the recursive part in two different ways:
   a. Recurse on the first through second to the last elements and add the last element

   b. Recurse on each half and add them together

2. Write a recursive function `bin_search(alist, item)` that searches for `item` in `alist` and returns `True` if found and `False` otherwise.

3. On the last ICA, you wrote `sum_cols(grid, n)` that sums column `n` in a grid:

```
def sum_cols(grid, n):
    if len(grid) == 0:
        return 0
    else:
        return grid[0][n] + sum_cols(grid[1:], n)
```

Now consider summing along the diagonal. Write a recursive function `sum_diag(grid)` that returns the sum of the diagonal from upper left to bottom right in a grid, i.e., it sums `grid[0][0]`, `grid[1][1]`, and so on. You may assume the grid is square.

Question: You can slice the grid (list of lists) in each round of recursion as usual. That means that `grid[0]` is the next row in each recursive call. But how will you know which column you need to index into for each recursive step?

Hint: Have `sum_diag(grid)` call a "helper" function called `sum_diag_helper` that is recursive. It has a **new** argument, `col`, that keeps track of the current column: `sum_diag_helper(grid, col)`. Call the helper function with 0 as the column number to start with.

```
def sum_diag(grid):

    return sum_diag_helper(grid, 0) # call the helper function

# sum_diag: a helper function
# the helper function has an additional argument, col
# col will keep track of the current column
# in the diagonal
def sum_diag_helper(grid, col):

    # your code goes here
```

4. Write a recursive function `zip(a,b)` where `a` and `b` are lists of any type. The function `zip(a,b)` returns a list of 2-tuples where the first tuple is `(a[0],b[0]),` the second is `(a[1],b[1]),` etc. Zipping stops when the shorter list runs out. For example, if `len(a)` is `3` and `len(b)` is `2`, then `len(zip(a,b))` is `2`. Don't use a helper function.

   Examples:

   ```
   zip([1,2,3],[4,5,6])          →   [ (1,4), (2,5), (3, 6) ]

   zip([1,2,3,4,5],['a','b','c','d','e']) →
             [(1, 'a'), (2, 'b'), (3, 'c'), (4, 'd'), (5, 'e')]

   zip([2,4,6], ['fall','leaves'] → [(2, 'fall'), (4, 'leaves')]

   zip([], [4,5,6])               → []
   ```

5. Now let's rewrite `zip` using a helper function. We will call `zip(a,b)` as before, but the new version will call a helper function `zip_helper(a,b,result)` that takes an additional list argument called `result`.

   Before each recursive call, `zip_helper(a,b,result)` will modify `result` by concatenating the new tuple of `(a[0], b[0])` to `result` and then pass the modified `result` list to the recursive call.

   ```
   def zip(a,b):

       # call the helper function

       # with an empty list

       return zip_helper(a,b,[])
   ```

```
def zip_helper(a,b,result):
    # consider carefully what should be returned
    # in the base case
```