EECE 230X – Introduction to Computation and Programming
Programming Assignment 10

- This programming assignment consists of 4 problems
- Prerequisites: Topic 12
- Related material: Module II: Recursion and elementary data structures

Problem 1. Generate all binary strings of a given weight

a) Write a recursive function `genBinStr2(n,w)`, which given integers n and w such that $w \leq n$, returns a list `L` consisting of all length-n binary strings with exactly w ones. If $n < 0$ or $w < 0$, the function should return the empty list.

Examples:

```
genBinStr2(3,0) should return ['000']
genBinStr2(3,1) should return ['001', '010', '100']
genBinStr2(3,2) should return ['011', '101', '110']
genBinStr2(3,3) should return ['111']
genBinStr2(4,2) should return ['0011', '0101', '0110', '1001', '1010', '1100']
genBinStr2(6,1) should return ['000001', '000010', '000100', '001000', '010000', '100000']
```

You are not supposed to solve this problem by first calling the function `genBinStr` we implemented in Topic 12 and then excluding all strings of weights not equal to w. Do it efficiently. You are not asked to analyze the time of your algorithm.

Don't try to read the following hint unless you are stuck (to read it you need to magnify the PDF file and properly read it in reverse order); try first to solve it on your own without help.

(*Hint*:)1-w,1-n(dna)w,1-n(:smelborpbus owT)

b) **Memoize recursion using a dictionary.** If you have solved Part (a) according to the hint, there will be repeated subproblems. Memoize the recursion to avoid resolving the same problems. Note that the problems are indexed with pairs (n, w). Memoizing using a 2-dimensional list is costly as not all pairs (n, w) will be solved. Use instead a dictionary whose keys are tuples (n, w).

Here again, you are not asked to analyze the time of your algorithm.

Problem 2. Generate pattern matrix

a) **Concatenating matrices horizontally.** In this part, we are given two matrices A and B represented as lists of lists. Assume that the number of columns of A is equal to that B and that the number of rows of A is equal to that B.

We can concatenate A and B vertically using the list + operators. For instance, consider the 2×3 matrices `A = [[1, 2, 3],[4, 5, 6]]` and `B = [[7, 8, 9 ], [10, 11, 12]]`. Then `A+B` is the 4×3 matrix `[[1, 2, 3],[4, 5, 6], [7, 8, 9 ], [10, 11, 12]]`.

You are asked in this part to write a function to concatenate them horizontally.

For instance, the horizontal concatenation of the above matrices A and B is the 2×6 matrix `[[1, 2, 3, 7, 8, 9 ],[4, 5, 6, 10, 11, 12]]`.

Implement the function `concatenateHorizontal(A,B)`, which given two matrices A and B represented as lists of lists, returns their horizontal concatenation. Assume that the number of rows of A is equal to the number of rows of B. Your function is not supposed to check if this condition holds.

Test program/output:

```
import numpy
A = [[1, 2, 3],[4, 5, 6]]
B = [[7, 8, 9 ], [10, 11, 12]]
C = A+B
D = concatenateHorizontal(A,B)
print("A:")
print(numpy.matrix(A))
print("\nB:")
print(numpy.matrix(B))
print("\nC:")
print(numpy.matrix(C))
print("\nD:")
print(numpy.matrix(D))
```

```
A:
[[1 2 3]
 [4 5 6]]

B:
[[ 7  8  9]
 [10 11 12]]

C:
[[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]]

D:
[[ 1  2  3  7  8  9]
 [ 4  5  6 10 11 12]]
```

b) **Generate pattern.** Implement the function `generatePatern(k)`, which given an integer $k \geq 0$, returns a $2^k \times 2^k$ matrix as shown in the examples below. Guess the pattern.

(*Hint:* Use recursion and Part (a)).

Test program:

```
import numpy
for k in range(5):
    print("k=",k,":")
    print(numpy.matrix(generatePattern(k)),"\n")

# For fun: show matrix as a figure with 1 and  0 in different colors
import matplotlib.pyplot as plt
plt.imshow(generatePattern(10))
plt.show()
```

Output:

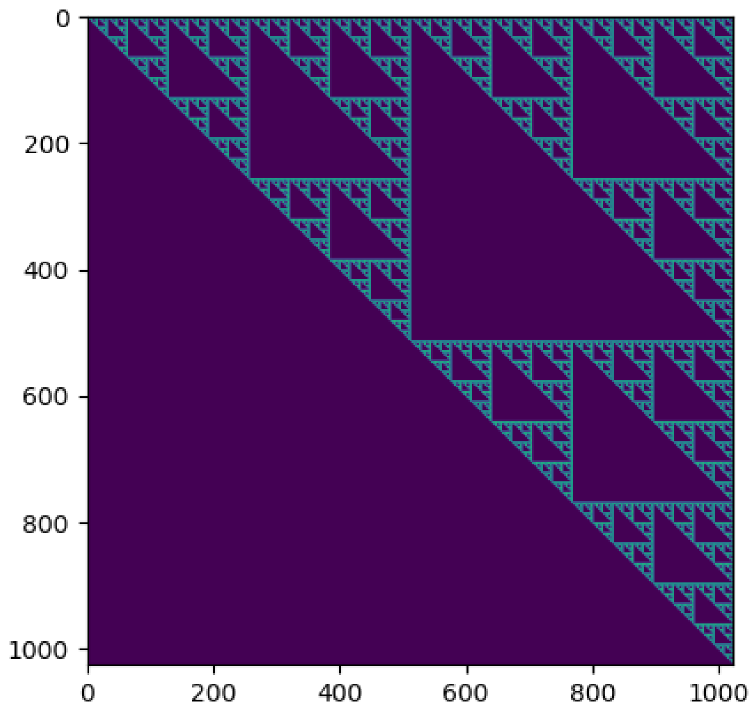
```
k= 0 :
[[1]]

k= 1 :
[[1 1]
 [0 1]]

k= 2 :
[[1 1 1 1]
 [0 1 0 1]
 [0 0 1 1]
 [0 0 0 1]]

k= 3 :
[[1 1 1 1 1 1 1 1]
 [0 1 0 1 0 1 0 1]
 [0 0 1 1 0 0 1 1]
 [0 0 0 1 0 0 0 1]
 [0 0 0 0 1 1 1 1]
 [0 0 0 0 0 1 0 1]
 [0 0 0 0 0 0 1 1]
 [0 0 0 0 0 0 0 1]]

k= 4 :
[[1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
 [0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1]
 [0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1]
 [0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 1]
 [0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1]
 [0 0 0 0 0 1 0 1 0 0 0 0 0 1 0 1]
 [0 0 0 0 0 0 1 1 0 0 0 0 0 0 1 1]
 [0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1]
 [0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1]
 [0 0 0 0 0 0 0 0 0 1 0 1 0 1 0 1]
 [0 0 0 0 0 0 0 0 0 0 1 1 0 0 1 1]
 [0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1]
 [0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1]]
```



Problem 3. Quick Sort with equal elements

We assumed in Topic 12 that all elements in the input list of randomized quick sort are distinct. The expected running to randomized quick sort is $O(n \log n)$ under this assumption. If the input list has many equal elements, partition may produce a highly unbalanced split irrespective of the position of the random pivot. For instance, in the extreme case when all the element are equal, one recursive call will be on an empty sublist and the other on the sublist consisting of all elements in the list excluding the pivot. This makes the worst case time $\Theta(n^2)$ irrespective of the choice of the pivot. In this problem, you will modify partition to make sure that randomized quick sort works well on lists with equal elements.

a) **Modified Partition.** Given list `A` and two indices p and r, the modified partition function `partition2(A,p,r)` rearranges `A[p...r]` around the pivot `x=A[r]` and returns a tuple of indices `(q1,q2)` such that:

- All elements in `A[p...q1-1]` are less than the pivot
- All elements `A[q1...q2]` are equal to the pivot
- All elements `A[q2+1...r]` are greater than the pivot

To do that, `partition2` maintains three sublists:

- Less-than sublist `A[p...i1]` consisting of elements less than pivot
- Equal sublist `A[i1+1...i2]` consisting of elements equal to the pivot
- Greater-than sublist `A[i2+1 ...j]` consisting of elements greater than the pivot

Initially, `i1 = p-1` and `i2 = p-1`. As in partition, the loop is on j starting with $j = p$ and ending with $j = r - 1$. Complete the code of `partition2`:

```
def partition2(A,p,r):
    x = A[r]
    i1 = p-1
    i2  = p-1
    for j in range(p,r):
        if A[j]==x:
            ...
            ....
        elif A[j]<x:
            ...
            ...
            ...
    i2+=1
    ....
    return (...,...)
```

Note that the `A[j]<x` case is more complex than the `A[j]==x` case.

b) **Modified quick Sort.** Rewrite `randPartition` and `randQuickSort` using `partition2` instead of `partition`. Call your functions `randPartition2` and `randQuickSort2`.

Test Program:

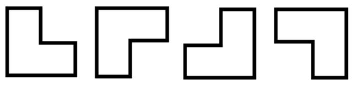
```
A1=[]
A2=[1]
A3=[0,1,2,0,1,2,0,1,2,1]
A4=[5,6,5,5,1,3,5,2,1,7,9,5,15,100,5, 2,17,5,56]
for A in (A1,A2,A3,A4):
    randQuickSort2(A)
    print("A sorted:",A)
```

Output:

```
A sorted: []
A sorted: [1]
A sorted: [0, 0, 0, 1, 1, 1, 1, 2, 2, 2]
A sorted: [1, 1, 2, 2, 3, 5, 5, 5, 5, 5, 5, 5, 6, 7, 9, 15, 17, 56, 100]
```

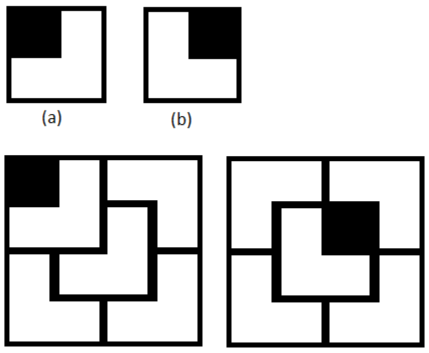
Problem 4 (*). Tiling a defective chessboard

We have a chessboard of dimension $n \times n$, where n a positive integer which we assume is a *power of* 2. The chessboard is *defective* in the sense that *one of its cells is missing.* We would like to tile the chessboard with *L-shaped tiles.* We are allowed to use the following 4 types of L-shaped tiles each consisting of 3 cells:



We are allowed to use as many L-shaped tiles as needed as long as they do not intersect, they cover all the cells of the defective board, and they are contained within the defective board.

In each of the following examples, the missing cell is the black one. In (a), we have $n = 2$, $i = 0$, and $j = 0$. In (b), we have $n = 2$, $i = 0$, and $j = 1$. In (c), we have $n = 4$, $i = 0$, and $j = 0$. In (d), we have $n = 4$, $i = 1$, and $j = 2$.



It can be shown that for any power n of 2, we can tile any $n \times n$ defective chessboard with L-shaped tiles; this will actually follow from your solution of this problem.

In this problem, you will write a program which given n and (i, j), finds the tiling.

The key is recursion: use divide-and-conquer.

Output: You are asked to store the tiling in an $n \times n$ 2-dimensional list $A[0 \ldots n-1][0 \ldots n-1]$. For the missing cells (i, j), we have $A[i][j] = 0$. For all other cells (u, v), $A[u][v]$ contains the tile index, for all $0 \leq u \leq n-1$ and $0 \leq v \leq n-1$ such that $(u, v) \neq (i, j)$. For example, the tiling in Figure (a) above is represented as:

$$\begin{matrix} 0 & 1 \\ 1 & 1. \end{matrix}$$

The value 0 indicates the missing cell. The cells set to 1 indicate the positions occupied by the tile. In Figure (d), the tiling is represented as:

$$\begin{matrix} 2 & 2 & 3 & 3 \\ 2 & 1 & 0 & 3 \\ 4 & 1 & 1 & 5 \\ 4 & 4 & 5 & 5. \end{matrix}$$

The cells set to 1 indicate the positions occupied by the first tile. Those set to 2 indicate the positions occupied by the second tile, and so on.

The difficulty of the problem is in figuring out the recursive function. The purpose of the recursive function is to set the values of A, which will be printed in the test program as shown below.

To get started, use a global variable *tileIndex* initialized to 1. For global variables, see Problem 1 of Programming Assignment 7. The variable *tileIndex* is the tile index. After placing the first tile, we increment *tileIndex*. In general, the cells set to *tileIndex* indicate the positions occupied by the tile whose index is *tileIndex*. Increment *tileIndex* in the recursive function each time you place a tile. When done, redo it without global variable (the use of global variables is discouraged as they result in dependencies between functions and variables defined outside those functions). To do it without global variables, pass `tileIndex` to the the recursive function `tileDefectiveChessBoardRec` (see the below code), increment `tileIndex` in the recursive function each time you place a tile, and let the recursive function return the tile index.

What is the running time of your algorithm? Derive a recurrence for the running time of your recursive function and solve it using Master Theorem.

Below is the code skeleton, which you are asked to complete.

```
def tileDefectiveChessBoard(n,i,j): # Wrapper function
    """ Tile an n by n tileDefective chess board with missing cell (i,j)
    Function assumes n is a power of 2, 0<=i<=n-1,and 0<=j<=n-1 """

    def tileDefectiveChessBoardRec(A,tileIndex,startX, endX, startY, endY, i,  j):
        """Tile defective chess board [startX ... endX]*[startY ... endY] with missing cell (i,j)
        Function assumes that endX-startX +1 = endY -startY +1 is a power of 2
        It  also assumes that  startX <=i<=endX and   startY <=j<=endY  """
        ....

    A = [[1 for v in range(n)] for u in range(n)]
    A[i][j] = 0 # zero indicates the missng cell
    tileDefectiveChessBoardRec(A,1,....) #set the parameter tileIndex  to 1
    return A


# Test program
import numpy # numerical python module
for (n,i,j) in ((1,0,0),(2,0,0),(2,0,1),(2,1,1),(4,0,0),(8,0,0),(8,2,4)):
    print("\ntileDefectiveChessBoard("+str(n)+","+str(i)+","+str(j)+")")
    A = tileDefectiveChessBoard(n,i,j)
    print(numpy.matrix(A))
```

Output:

```
tileDefectiveChessBoard(1,0,0):
[[0]]

tileDefectiveChessBoard(2,0,0):
[[0 1]
 [1 1]]

tileDefectiveChessBoard(2,0,1):
[[1 0]
 [1 1]]

tileDefectiveChessBoard(2,1,1):
[[1 1]
 [1 0]]

tileDefectiveChessBoard(4,0,0):
[[0 2 3 3]
 [2 2 1 3]
 [4 1 1 5]
 [4 4 5 5]]

tileDefectiveChessBoard(8,0,0):
[[ 0  3  4  4  8  8  9  9]
 [ 3  3  2  4  8  7  7  9]
 [ 5  2  2  6 10 10  7 11]
 [ 5  5  6  6  1 10 11 11]
 [13 13 14  1  1 18 19 19]
 [13 12 14 14 18 18 17 19]
 [15 12 12 16 20 17 17 21]
 [15 15 16 16 20 20 21 21]]
```

```
tileDefectiveChessBoard(8,2,4):
[[ 3  3  4  4  8  8  9  9]
 [ 3  2  2  4  8  7  7  9]
 [ 5  2  6  6  0 10  7 11]
 [ 5  5  6  1 10 10 11 11]
 [13 13 14  1  1 18 19 19]
 [13 12 14 14 18 18 17 19]
 [15 12 12 16 20 17 17 21]
 [15 15 16 16 20 20 21 21]]
```