Assignment 3

Problem 1: Design a Correct Partition Algorithm

You are given code below for an incorrect partition algorithm that fails to partition arrays wrongly or cause out of bounds access in arrays. The comments include the invariants the algorithm wishes to maintain and will help you debug.

Your goal is to write test cases that demonstrate that the partitioning will fail in various ways.

```
def swap(a, i, j):
    assert 0 <= i < len(a), f'accessing index {i} beyond end of array {len(a)}'
    assert 0 <= j < len(a), f'accessing index {j} beyond end of array {len(a)}'
    a[i], a[j] = a[j], a[i]

def tryPartition(a):
    # implementation of Lomuto partitioning algorithm
    n = len(a)
    pivot = a[n-1] # choose last element as the pivot.
    i,j = 0,0 # initialize i and j both to be 0
    for j in range(n-1): # j = 0 to n-2 (inclusive)
        # Invariant: a[0] .. a[i] are <= pivot
        #            a[i+1]...a[j-1] are > pivot
        if a[j] <= pivot:
            swap(a, i+1, j)
            i = i + 1
    swap(a, i+1, n-1) # place pivot in its correct place.
    return i+1 # return the index where we placed the pivot
```

First write a function that will return True if an array is correctly partitioned at index `k`. I.e, all elements at indices `< k` are all `<= a[k]` and all elements indices `> k` are all `> a[k]`

```
def testIfPartitioned(a, k):
    # TODO : test if all elements at indices < k are all <= a[k]
    #         and all elements at indices > k are all > a[k]
    # return TRUE if the array is correctly partitioned around a[k] and return FALSE otherw
    assert 0 <= k < len(a)

    # your code here
    # if k == 0:
    #   left = []
    #   right = a[k+1:]
    #   for j in right:
    #     if j < a[k]:
    #       return False
    # elif k == len(a)-1:
```

```
    #   left = a[:k]
    #   right = []
    #   for i in left:
    #     if i > a[k]:
    #       return False
    # else:
    #   left = a[:k]
    #   right = a[k+1:]
    #   for i,j in zip(left,right):
    #     if i > a[k] or j < a[k]:
    #       return False

    # Check elements to the left of index k
    for i in range(k):
        if a[i] > a[k]:
            return False

    # Check elements to the right of index k
    for j in range(k+1, len(a)):
        if a[j] <= a[k]:
            return False

    return True

assert testIfPartitioned([-1, 5, 2, 3, 4, 8, 9, 14, 10, 23],5) == True, ' Test # 1 failed.'
assert testIfPartitioned([-1, 5, 2, 3, 4, 8, 9, 14, 11, 23],4) == False, ' Test # 2 failed.'
assert testIfPartitioned([-1, 5, 2, 3, 4, 8, 9, 14, 23, 21],0) == True, ' Test # 3 failed.'
assert testIfPartitioned([-1, 5, 2, 3, 4, 8, 9, 14, 22, 23],9) == True, ' Test # 4 failed.'
assert testIfPartitioned([-1, 5, 2, 3, 4, 8, 9, 14, 8, 23],5) == False, ' Test # 5 failed.'
assert testIfPartitioned([-1, 5, 2, 3, 4, 8, 9, 13, 9, -11],5) == False, ' Test # 6 failed.'
assert testIfPartitioned([4, 4, 4, 4, 4, 8, 9, 13, 9, 11],4) == True, ' Test # 7 failed.'
print('Passed all tests (10 points)')
```

```
Passed all tests (10 points)
```

```
# Write an array called a1 that will be incorrectly partitioned by the tryPartition algorith
# Your input when run on tryPartition algorithm should raise an out of bounds array access e
# in the swap function or fail to partition correctly.

## Define an array a1 below of length > 0 that will be incorrectly partitioned by tryPartit
## We will test whether your solution works in the subsequent cells.
# your code here
a1 = [-1,4,8]

assert( len(a1) > 0)

# Write an array called a2 that will be incorrectly partitioned by the tryPartition algorith
```

```
# Your input when run on tryPartition algorithm should raise an out of bounds array access e
# in the swap function or fail to partition correctly.
# a2 must be different from a1

# your code here
a2 = [9,2,0]

assert( len(a2) > 0)
assert (a1 != a2)


# Write an array called a3 that will be incorrectly partitioned by the tryPartition algorith
# Your input when run on tryPartition algorithm should raise an out of bounds array access e
# in the swap function or fail to partition correctly.
# a3 must be different from a1, a2

# your code here
a3 = [2,9,-5,4,3]

assert( len(a3) > 0)
assert (a3 != a2)
assert (a3 != a1)

def dummyFunction():
    pass

# your code here

try:
    j1 = tryPartition(a1)
    assert not testIfPartitioned(a1, j1)
    print('Partitioning was unsuccessful - this is what you were asked to break the code')
except Exception as e:
    print(f'Assertion failed {e} - this is fine since you were asked to break the code.')

try:
    j2 = tryPartition(a2)
    assert not testIfPartitioned(a2, j2)
except Exception as e:
    print(f'Assertion failed {e} - this is fine since you were asked to break the code.')


try:
    j3 = tryPartition(a3)
    assert not testIfPartitioned(a3, j3)
except Exception as e:
    print(f'Assertion failed {e} - this is fine since you were asked to break the code.')
```

```
dummyFunction()
```

```
print('Passed 5 points!')
```

```
Assertion failed accessing index 3 beyond end of array 3 - this is fine since you were asked
Assertion failed accessing index 5 beyond end of array 5 - this is fine since you were asked
Passed 5 points!
```

Debug the function

Point out where the bug is and what the fix is for the tryPartition function. Note that the answer below is not graded.

YOUR ANSWER HERE

Problem 2. Rapid Sorting of Arrays with Bounded Number of Elements.

Thus far, we have presented sorting algorithms that are comparison-based. Ie., they make no assumptions about the elements in the array just that we have a `<=` comparison operator. We now ask you to develop a rapid sorting algorithm for an array of size n when it is given to you that all elements in the array are between $1, \ldots, k$ for a given k. Eg., consider an array with n = 100000 elements wherein all elements are between 1,..., k = 100.

Develop a sorting algorithm using partition that runs in $\Theta(n \times k)$ time for such arrays. **Hint** You can choose your pivots in a simple manner each time.

Part A

Describe your algorithm as pseudocode and argue why it runs in time $\Theta(n \times k)$. This part will not be graded but is intended for your own edification.

YOUR ANSWER HERE

Part B

Complete the implementation of a function `boundedSort(a, k)` by completing the `simplePatition` function. Given an array `a` and a fixed `pivot` element, it should partition the array "in-place" so that all elements `<= pivot` are on one side of the array and elements `> pivot` on the other. You should not create a new array in your code.

```
def swap(a, i, j):
    assert 0 <= i < len(a), f'accessing index {i} beyond end of array {len(a)}'
    assert 0 <= j < len(a), f'accessing index {j} beyond end of array {len(a)}'
    a[i], a[j] = a[j], a[i]
```

```
def simplePartition(a, pivot):
    ## To do: partition the array a according to pivot.
    # Your array must be partitioned into two regions - <= pivot followed by elements > pivo
    ## If an element at the beginning of the array is already <= pivot in the beginning of t
    ## be moved by the algorithm.
    # your code here
    left = 0
    right = len(a) - 1

    while left <= right:
      while left <= right and a[left] <= pivot:
          left += 1
      while left <= right and a[right] > pivot:
          right -= 1
      if left < right:
          swap(a, left, right)
          left += 1
          right -= 1
```

```
def boundedSort(a, k):
    for j in range(1, k):
        simplePartition(a, j)
```

```
a = [1, 3, 6, 1, 5, 4, 1, 1, 2, 3, 3, 1, 3, 5, 2, 2, 4]
print(a)
simplePartition(a, 1)
print(a)
assert(a[:5] == [1,1,1,1,1]), 'Simple partition test 1 failed'

simplePartition(a, 2)
print(a)
assert(a[:5] == [1,1,1,1,1]), 'Simple partition test 2(A) failed'
assert(a[5:8] == [2,2,2]), 'Simple Partition test 2(B) failed'


simplePartition(a, 3)
print(a)
```

```
assert(a[:5] == [1,1,1,1,1]), 'Simple partition test 3(A) failed'
assert(a[5:8] == [2,2,2]), 'Simple Partition test 3(B) failed'
assert(a[8:12] == [3,3,3,3]), 'Simple Partition test 3(C) failed'

simplePartition(a, 4)
print(a)
assert(a[:5] == [1,1,1,1,1]), 'Simple partition test 4(A) failed'
assert(a[5:8] == [2,2,2]), 'Simple Partition test 4(B) failed'
assert(a[8:12] == [3,3,3,3]), 'Simple Partition test 4(C) failed'
assert(a[12:14]==[4,4]), 'Simple Partition test 4(D) failed'

simplePartition(a, 5)
print(a)
assert(a == [1]*5+[2]*3+[3]*4+[4]*2+[5]*2+[6]), 'Simple Parition test 5 failed'

print('Passed all tests : 10 points!')
```

```
[1, 3, 6, 1, 5, 4, 1, 1, 2, 3, 3, 1, 3, 5, 2, 2, 4]
[1, 1, 1, 1, 1, 4, 5, 6, 2, 3, 3, 3, 3, 5, 2, 2, 4]
[1, 1, 1, 1, 1, 2, 2, 2, 6, 3, 3, 3, 3, 5, 5, 4, 4]
[1, 1, 1, 1, 1, 2, 2, 2, 3, 3, 3, 3, 6, 5, 5, 4, 4]
[1, 1, 1, 1, 1, 2, 2, 2, 3, 3, 3, 3, 4, 4, 5, 5, 6]
[1, 1, 1, 1, 1, 2, 2, 2, 3, 3, 3, 3, 4, 4, 5, 5, 6]
Passed all tests : 10 points!
```

Problem 3: Design a Universal Family Hash Function

Suppose we are interested in hashing n bit keys into m bit hash values to hash into a table of size 2^m. We view our key as a bit vector of n bits in binary. Eg., for $n = 4$, the key $14 = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 0 \end{pmatrix}$.

The hash family is defined by random boolean matrices H with m rows and n columns. To compute the hash function, we perform a matrix multiplication. Eg., with $m = 3$ and $n = 4$, we can have a matrix H such as

$$H = \begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 \end{bmatrix}$$

.

The value of the hash function $H(14)$ is now obtained by multiplying

$$\begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 \end{bmatrix} \times \begin{pmatrix} 1 \\ 1 \\ 1 \\ 0 \end{pmatrix}$$

The matrix multiplication is carried out using AND for multiplication and XOR instead of addition. For the example above, we compute the value of hash function as

$$\begin{pmatrix} 0\cdot 1 + 1\cdot 1 + 0\cdot 1 + 1\cdot 0 \\ 1\cdot 1 + 0\cdot 1 + 0\cdot 1 + 0\cdot 0 \\ 1\cdot 1 + 0\cdot 1 + 1\cdot 1 + 1\cdot 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix}$$

(A) For a given matrix H and two keys x, y that differ only in their i^{th} bits, provide a condition for $Hx = Hy$ holding. (**Hint** It may help to play with examples where you have two numbers x, y that just differ at a particular bit position. Figure out which entries in the matrix are multiplied with these bits that differ).

YOUR ANSWER HERE

(B) Prove that the probability that two keys x, y such that $x \neq y$ collide under the random choice of a matrix x, y is at most $\frac{1}{2^m}$.

YOUR ANSWER HERE

```
from random import random

def dot_product(lst_a, lst_b):
    and_list = [elt_a * elt_b for (elt_a, elt_b) in zip(lst_a, lst_b)]
    return 0 if sum(and_list)% 2 == 0 else 1

# encode a matrix as a list of lists with each row as a list.
# for instance, the example above is written as the matrix
# H = [[0,1,0,1],[1,0,0,0],[1,0,1,1]]
# encode column vectors simply as a list of elements.
# you can use the dot_product function provided to you.
def matrix_multiplication(H, lst):
    # your code here
    result = []
    for lst_a in H:
      result.append(dot_product(lst_a, lst))
    return result


# Generate a random m \times n matrix
# see the comment next to matrix_multiplication for how your matrix must be returned.
```

```
def return_random_hash_function(m, n):
    # return a random hash function wherein each entry is chosen as 1 with probability >= 1/
    # your code here
    import random
    rand_matrix = []
    count = 1
    while count <= m:
      rand_matrix.append([random.randint(0, 1) for _ in range(n)])
      count+=1
    return rand_matrix
```

```
A1 = [[0,1,0,1],[1,0,0,0],[1,0,1,1]]
b1 = [1,1,1,0]
c1 = matrix_multiplication(A1, b1)
print('c1=', c1)
assert c1 == [1,1,0] , 'Test 1 failed'

A2 = [ [1,1],[0,1]]
b2 = [1,0]
c2 = matrix_multiplication(A2, b2)
print('c2=', c2)
assert c2 == [1, 0], 'Test 2 failed'

A3 = [ [1,1,1,0],[0,1,1,0]]
b3 =  [1, 0,0,1]
c3 = matrix_multiplication(A3, b3)
print('c3=', c3)
assert c3 == [1, 0], 'Test 3 failed'

H = return_random_hash_function(5,4)
print('H=', H)
assert len(H) == 5, 'Test 5 failed'
assert all(len(row) == 4 for row in H), 'Test 6 failed'
assert all(elt == 0 or elt == 1 for row in H for elt in row ),  'Test 7 failed'

H2 = return_random_hash_function(6,3)
print('H2=', H2)
assert len(H2) == 6, 'Test 8 failed'
assert all(len(row) == 3 for row in H2),  'Test 9 failed'
assert all(elt == 0 or elt == 1 for row in H2 for elt in row ), 'Test 10 failed'
print('Tests passed: 10 points!')
```

```
c1= [1, 1, 0]
c2= [1, 0]
c3= [1, 0]
H= [[0, 1, 0, 1], [1, 1, 0, 0], [1, 0, 0, 1], [0, 1, 1, 1], [1, 0, 1, 1]]
H2= [[0, 0, 0], [1, 1, 0], [0, 0, 0], [1, 1, 1], [1, 0, 1], [1, 1, 1]]
```

```
Tests passed: 10 points!
```

Manually Graded Answers

Problem 1

The bug is in the initialization of i in the algorithm. It must be i =-1 rather than i = 0. Due to this, either the first element of the array is never considered during the partition or there could be an access to i+1 that is out of array bounds.

Problem 2 A

```
for k = 1 to n
   j = partition array a with k as pivot
```

The running time is $\Theta(n \times k)$.

Problem 3 A

Since x, y differe only in their i^{th} bits, we can assume $x_i = 0$ and $y_i = 1$. Therefore, $Hx + H_i = Hy$ wherein, + refers to entrywise XOR and H_i is the i^{th} column of H. Thus, $Hx = Hy$ if and only if H_i has all zeros. This happens with probability $\frac{1}{2^m}$.

Problem 3 B

Let us assume that x and y differ in k out of n positions. We know that $Hx = Hy$ if and only if $Hx + Hy = 0$ where + is XOR and 0 is the vector of all zeros. But $Hx + Hy = H(x + y)$ since AND distributes over XOR.

Whenever x and y agree in the i^{th} entries, we have the i^{th} entry of $(x + y)$ is zero. Therefore, $H(x + y)$ is just the XOR sum of k columns of H corresponding to positions where x and y differ.

Thus, one of the columns must equal the sum of the remaining $k - 1$ columns. Let us fix these $k - 1$ columns as given and the last column as randomly chosen. The probability that each of the m entries of the last column matches the sum of the first $k - 1$ column is $\frac{1}{2^m}$.

That's all folks