**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 otherwise*

**assert** 0 **<=** k **<** len(a)

​

**return** all([a[i] **<=** a[k] **for** i **in** range(k)]) **and** all([a[i] **>** a[k] **for** i **in** range(k**+**1, len(a))])

**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)')

*# Write an array called a1 that will be incorrectly partitioned by the tryPartition algorithm above*

*# Your input when run on tryPartition algorithm should raise an out of bounds array access exception*

*# in the swap function or fail to partition correctly.*

​

*## Define an array a1 below of length > 0 that will be incorrectly partitioned by tryPartition algorithm.*

*## We will test whether your solution works in the subsequent cells.*

​

a1 **=** [**-**1, 7, 4, 2, 5, 9, 20, 15, 11, 18]

​

**assert**( len(a1) **>** 0)

​

*# Write an array called a2 that will be incorrectly partitioned by the tryPartition algorithm above*

*# Your input when run on tryPartition algorithm should raise an out of bounds array access exception*

*# in the swap function or fail to partition correctly.*

*# a2 must be different from a1*

​

​

a2 **=** [**-**1, 7, 4, 2, 5, 9, 20, 15, 10, 18]

​

**assert**( len(a2) **>** 0)

**assert** (a1 **!=** a2)

​

​

*# Write an array called a3 that will be incorrectly partitioned by the tryPartition algorithm above*

*# Your input when run on tryPartition algorithm should raise an out of bounds array access exception*

*# in the swap function or fail to partition correctly.*

*# a3 must be different from a1, a2*

​

​

a3 **=** [**-**1, 7, 4, 2, 5, 9, 20, 15, 10, 23]

**assert**( len(a3) **>** 0)

**assert** (a3 **!=** a2)

**assert** (a3 **!=** a1)

​

**def** dummyFunction():

**pass**

​

​

**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!')

​

**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,…,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 Θ(𝑛×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 Θ(𝑛×𝑘). This part will not be graded but is intended for your own edification.

YOUR ANSWER HERE def rapidSort(a, k):

# Step 1: Initialize a count array of size k (for values 1 to k)

count = [0] \* k

# Step 2: Count occurrences of each element

for num in a:

count[num - 1] += 1 # Increment the count for the number

# Step 3: Reconstruct the sorted array

sorted\_array = []

for i in range(k):

sorted\_array.extend([i + 1] \* count[i]) # Add 'count[i]' number of 'i+1' to the sorted array

return sorted\_array

**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 > pivot*

*## If an element at the beginning of the array is already <= pivot in the beginning of the array, it should not*

*## be moved by the algorithm.*

left\_partition, right\_partition **=** [],[]

**for** elt **in** a:

**if** elt **<=** pivot:

left\_partition.append(elt)

**else**:

right\_partition.append(elt)

**for** i, elt **in** enumerate(left\_partition **+** right\_partition):

a[i] **=** elt

**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!')

**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 2𝑚. We view our key as a bit vector of 𝑛 bits in binary. Eg., for 𝑛=4, the key 14=1110.

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=011100001101

.

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

011100001101×1110

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

0⋅1+1⋅1+0⋅1+1⋅01⋅1+0⋅1+0⋅1+0⋅01⋅1+0⋅1+1⋅1+1⋅0=110

(A) For a given matrix 𝐻 and two keys 𝑥,𝑦 that differ only in their 𝑖𝑡ℎ 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 Condition for H(x)=H(y)H(x)=H(y) Given that xx and yy differ only in their ii-th bit, forH(x)=H(y)H(x)=H(y) to hold, the rows of the matrix H that correspond to the ii-th bit in the keys xx and yy must be identical. Specifically, the entries in the matrix HH that correspond to the ii-th bit of both xx and yy must yield the same result in the matrix multiplication, i.e., the contribution to the hash value from this bit must be the same for both keys.

(B) Prove that the probability that two keys 𝑥,𝑦 such that 𝑥≠𝑦 collide under the random choice of a matrix 𝑥,𝑦 is at most 12𝑚.

YOUR ANSWER HERE (B) Probability of Collision

For two keys xx and yy (where x≠yx=y) to collide under a random matrix HH, the probability is at most: 12m2 m 1​This is because for a collision to occur, each of the mm rows in the matrix must align in such a way that the resulting hash values for xx and yy are the same. Each row contributes independently, and the probability of a match in any row is 1221​ . Therefore, the probability of a collision across all mm rows is 12m2 m1​ , which is the upper bound on the probability of a collision.

**from** random **import** random

**from** random **import** randint

​

**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):

**return** [dot\_product(lst\_a, lst) **for** lst\_a **in** H]

​

*# 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 >= 1/2 and 0 with probability < 1/2*

**return** [[randint(0,1) **for** j **in** range(n)] **for** i **in** range(m)]

​

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!')

**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 Θ(𝑛×𝑘).

**Problem 3 A**

Since  𝑥,𝑦 differe only in their 𝑖𝑡ℎ bits, we can assume 𝑥𝑖=0 and 𝑦𝑖=1. Therefore, 𝐻𝑥+𝐻𝑖=𝐻𝑦 wherein, + refers to entrywise XOR and 𝐻𝑖 is the 𝑖𝑡ℎ column of 𝐻. Thus, 𝐻𝑥=𝐻𝑦 if and only if 𝐻𝑖 has all zeros. This happens with probability 12𝑚.

**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 𝑖𝑡ℎ entries, we have the 𝑖𝑡ℎ entry of (𝑥+𝑦) is zero. Therefore, 𝐻(𝑥+𝑦) is just the XOR sum of 𝑘 columns of 𝐻corresponding 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 12𝑚.

**That's all folks**

Thank you for stopping by! -Sulay Cay ☺