Week 6 & 7 Live Coding Solutions

Week-6 Live coding problem 1

Write a function `type_of_heap(A)` that accept a heap `A` and return string `Max` if input heap is max heap, `Min` if input heap is a min heap and `None` otherwise.

Sample input 1

```
[1,2,3,4,5,6]
```

Output

```
Min
```

Sample input 2

```
[5,3,4,2,1]
```

Output

```
Max
```

Sample input 3

```
[1,5,4,7,6,3,2]
```

Output

```
None
```

Solution

Solution Code

```
def minheap(A):
    for i in range((len(A) - 2) // 2 + 1):
        if A[i] > A[2*i + 1] or (2*i + 2 != len(A) and A[i] > A[2*i + 2]):
            return False
    return True
def maxheap(A):
    for i in range((len(A) - 2) // 2 + 1):
        if A[i] < A[2*i + 1] or (2*i + 2 != len(A) and A[i] < A[2*i + 2]):
            return False
    return True
def type_of_heap(A):
    if minheap(A)==True:
        return 'Min'
    if maxheap(A)==True:
        return 'Max'
    return 'None'
```

Suffix Code(visible)

```
A=eval(input())
print(type_of_heap(A))
```

Public Test case

Input 1

```
[1,2,3,4,5,6,7,8,9]
```

Output

```
Min
```

Input 2

```
[5,3,4,2,1]
```

Output

```
Max
```

Input 3

```
[1,5,4,7,6,3,2]
```

Output

```
None
```

Private Test case

Input 1

```
[67,65,43,54,6,2,1,19,5]
```

Output

```
Max
```

Input 2

```
[1,2,3,4,8,5,6,7]
```

Output

```
Min
```

Input 3

```
[1,2,3,4,5,9,8,7,6,12,13]
```

Output

```
Min
```

Week-6 Live coding problem 2

Write a function `findRedundantEdges(E,n)` that accept an edge list `E` in increasing order of the edge weight where all edge weights are distinct and the number of vertices `n` (labeled from 0 to n-1) in a connected undirected graph and the function returns a list of redundant edges in increasing order of weight, so by removing these edges, the graph should remain connected with the minimum total cost of edges(minimum cost spanning tree). Try to write solution code of complexity O((E+V)logV).

Note - Selected edges tuples in the output list should be similar to input list edges tuples.

Hint- Union-find data structure

Sample input 2

```
4
[(0,1,10),(1,2,20),(2,3,30),(3,0,40),(1,3,50)]
```

Output

```
[(3, 0, 40), (1, 3, 50)]
```

Solution

Solution Code

```
class MakeUnionFind:
    def __init__(self):
        self.components = {}
        self.members = {}
        self.size = {}
    def make_union_find(self,vertices):
        for vertex in range(vertices):
            self.components[vertex] = vertex
            self.members[vertex] = [vertex]
            self.size[vertex] = 1
    def find(self,vertex):
        return self.components[vertex]
    def union(self,u,v):
        c_old = self.components[u]
        c_new = self.components[v]
        # Always add member in components which have greater size
        if self.size[c_new] >= self.size[c_old]:
            for x in self.members[c_old]:
                self.components[x] = c_new
                self.members[c_new].append(x)
                self.size[c_new] += 1
        else:
            for x in self.members[c_new]:
                self.components[x] = c_old
                self.members[c_old].append(x)
```

```
                self.size[c_old] += 1

def findRedundantEdges(E,n):
    st = MakeUnionFind()
    st.make_union_find(n)
    redlist=[]
    for edge in E:
        if st.find(edge[0])!=st.find(edge[1]):
            st.union(edge[0], edge[1])
        else:
            redlist.append(edge)
    return redlist
```

Suffix code(visible)

```
n = int(input())
E=eval(input())
print(findRedundantEdges(E,n))
```

Public Test case

Input 1

```
7
[(0,1,10),(0,2,50),(0,3,60),(5,6,75),(2,1,80),(6,4,90),(1,6,100),(2,5,110),
(1,3,150),(3,4,180),(2,4,200)]
```

Output

```
[(2, 1, 80), (2, 5, 110), (1, 3, 150), (3, 4, 180), (2, 4, 200)]
```

Input 2

```
4
[(0,1,10),(1,2,20),(2,3,30),(3,0,40),(1,3,50)]
```

Output

```
[(3, 0, 40), (1, 3, 50)]
```

Input 3

```
4
[(0,2,1),(0,1,2),(0,3,3),(1,2,4),(2,3,6)]
```

Output

```
[(1, 2, 4), (2, 3, 6)]
```

Private Test case

Input 1

```
6
[(0,1,1),(1,2,3),(1,3,4),(0,2,5),(2,4,7),(2,3,12),(3,4,13),(1,5,15),(2,5,17),(3,5,21),(4,5,25)]
```

Output

```
[(0, 2, 5), (2, 3, 12), (3, 4, 13), (2, 5, 17), (3, 5, 21), (4, 5, 25)]
```

Input 2

```
6
[(0,1,1),(1,2,3),(1,3,4),(1,4,5),(0,4,7),(0,5,10),(2,3,12),(3,4,13),(1,5,15),(2,5,17),(3,5,21),(4,5,25)]
```

Output

```
[(0, 4, 7), (2, 3, 12), (3, 4, 13), (1, 5, 15), (2, 5, 17), (3, 5, 21), (4, 5, 25)]
```

Input 3

```
7
[(0,1,10),(1,2,50),(2,3,60),(3,0,75),(3,1,80),(6,4,90),(1,6,100),(2,5,110),(3,6,150),(3,4,180),(0,4,200)]
```

Output

```
[(3, 0, 75), (3, 1, 80), (3, 6, 150), (3, 4, 180), (0, 4, 200)]
```

Week-6 Live coding problem 3

Write a function `find_kth_largest(root, k)` that accept `root` as a reference of root node of BST of `n` elements and an integer k, where `0 < k <= n`. The function should return the `kth` largest element without doing any modification in Binary Search Tree. The complexity of the solution should be in order of `O(logn + k)`

Structure of the Tree class

```
class Tree:
    def __init__(self,initval=None):
        self.value = initval
        if self.value:
            self.left = Tree()
            self.right = Tree()
        else:
            self.left = None
            self.right = None
        return
```

Sample input

```
[5,4,6,3,2,1,7] #bst created using given sequence
3 #k
```

Output

```
5
```

Solution

Solution Code

```
def kthlargest(root):
    global count,result
    if root.right!=None:
        find_kth_largest(root.right,k)
        count += 1
        if count == k:
            result = root.value
            return
        find_kth_largest(root.left,k)
count = 0
result = -1
def find_kth_largest(root,k):
    kthlargest(root)
    return result
```

Suffix code(hidden)

```
class Tree:
# Constructor:
    def __init__(self,initval=None):
        self.value = initval
        if self.value:
            self.left = Tree()
            self.right = Tree()
        else:
            self.left = None
            self.right = None
        return
    # Only empty node has value None
    def isempty(self):
        return (self.value == None)

    def insert(self,v):
        if self.isempty():
            self.value = v
            self.left = Tree()
            self.right = Tree()
        if self.value == v:
            return
        if v < self.value:
            self.left.insert(v)
            return
        if v > self.value:
            self.right.insert(v)
            return

T = Tree()
bst = eval(input())
k = int(input())
for i in bst:
    T.insert(i)
print(find_kth_largest(T,k))
```

Public Test case

Input 1

```
[5,4,6,3,2,1,7]
3
```

Output

```
5
```

Input 2

```
[8,7,6,5,4,3,2,1]
2
```

Output

```
7
```

Input 3

```
[108, 348, 332, 463, 167, 148, 155, 331, 435, 349, 261, 336, 135, 449, 384,
183, 428, 262, 434, 276, 87, 29, 203, 24, 347, 119, 251, 370, 456, 433, 49,
421, 410, 57, 218, 226, 359, 163, 42, 179, 192, 10, 295, 235, 99, 286, 116,
290, 169, 146, 71, 34, 44, 141, 353, 132, 346, 488, 84, 16, 74, 289, 424, 59,
240, 252, 427, 250, 321, 281, 496, 288, 112, 408, 393, 247, 12, 387, 447,
278, 323, 338, 483, 379, 80, 114, 365, 118, 77, 164, 154, 325, 376, 180, 54,
140, 401, 223, 50, 14, 396, 25, 117, 38, 230, 144, 440, 206, 48, 388, 227,
268, 360, 300, 414, 274, 445, 200, 444, 106, 324, 490, 211, 477, 476, 238,
354, 204, 195, 258, 404, 26, 471, 263, 468, 176, 58, 110, 15, 19, 264, 378,
94, 439, 186, 193, 91, 419, 30, 102, 174, 7, 337, 136, 143, 88, 134, 291]
5
```

Output

```
477
```

Private Test case

Input 1

```
[15,4,7,8,5,3,9,13,16,1]
1
```

Output

```
16
```

Input 2

```
[15,4,7,8,5,3,9,13,16,1]
10
```

Output

```
1
```

Input 3

```
[364, 266, 305, 157, 133, 391, 316, 68, 409, 432, 172, 39, 467, 92, 277, 82,
425, 311, 107, 204, 120, 497, 320, 178, 359, 90, 206, 239, 153, 1, 91, 31,
392, 106, 209, 262, 303, 122, 430, 195, 191, 156, 60, 344, 285, 67, 268, 496,
225, 4, 96, 396, 358, 356, 429, 235, 108, 291, 275, 388, 341, 465, 118, 12,
363, 161, 104, 486, 197, 20, 18, 472, 164, 199, 366, 26, 336, 227, 287, 244,
132, 272, 258, 110, 299, 184, 142, 86, 243, 50, 185, 167, 294, 116, 11, 180,
416, 176, 450, 10, 245, 492, 264, 121, 421, 454, 362, 162, 386, 6, 37, 426,
408, 41, 134, 298, 30, 25, 74, 155, 301, 128, 489, 340, 329, 446, 3, 282, 13,
233, 475, 64, 190, 315, 51, 373, 61, 474, 399, 213, 248, 208, 331, 179, 471,
140, 249, 415, 77, 186, 317, 188, 57, 230, 293, 148, 457, 355, 260, 276, 177,
322, 189, 418]
10
```

Output

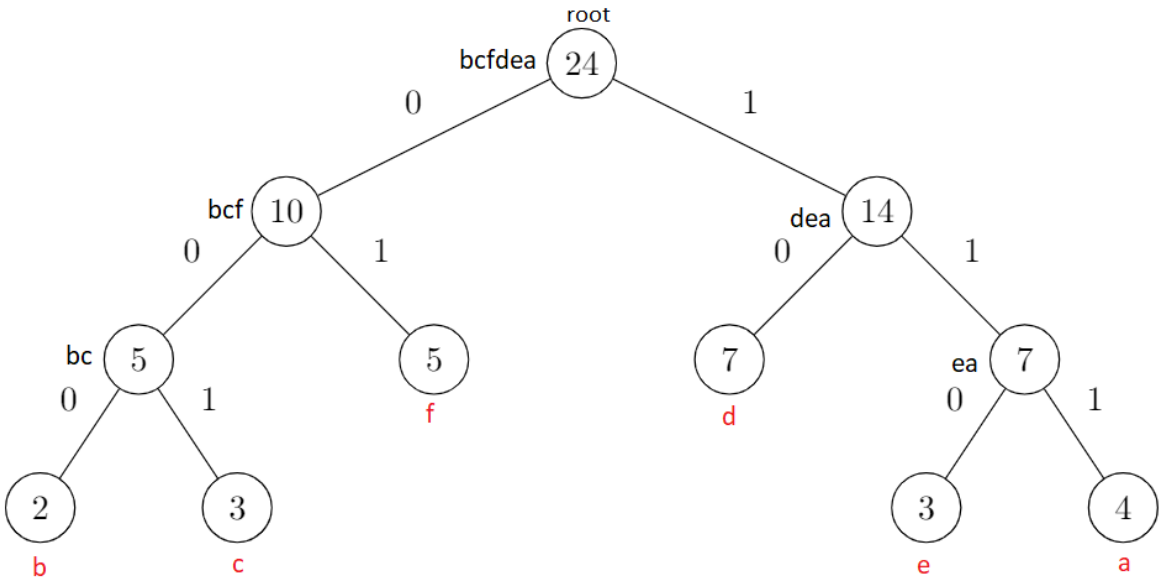
```
467
```

Week-7 Live coding problem 1

Write a function **decode(root, ciphertext)** that accepts a variable `root` which contains the reference of the root node of Huffman tree and an encoded message `ciphertext` in the form of a string (using 0 and 1). The function returns the decoded message in the form of a string.

Structure of node in given Huffman tree

```
class Node:
    def __init__(self,frequency,symbol = None,left = None,right = None):
        self.frequency = frequency
        self.symbol = symbol
        self.left = left
        self.right = right
```



- Leaf node contains the final symbol (in red color)

Sample Input

```
0001110001110000011011001 #Encoded message
```

Output

```
bababcdef # Decoded message
```

Solution

Solution code

```
# Solution
def decode(root,ciphertext):
    message =''
    temp = root
    for i in ciphertext:
        if i == '0':
            temp = temp.left
        if i == '1':
            temp = temp.right
        if temp.left == None and temp.right == None:
            message += temp.symbol
            temp = root
    return message
```

Suffix Code(Hidden)

```
class Node:
    def __init__(self,frequency,symbol=None,left=None,right=None):
        self.frequency = frequency
        self.symbol = symbol
        self.left = left
        self.right = right

def Huffman(s):
    char = list(s)
    freqlist=[]
    unique = set(char)
    for c in unique:
        freqlist.append((char.count(c),c))
    nodes = []
    for nd in sorted(freqlist):
        nodes.append((nd,Node(nd[0],nd[1])))
    while len(nodes) > 1:
        nodes.sort()
        L = nodes[0][1]
        R = nodes[1][1]
        newnode = Node(L.frequency + R.frequency,L.symbol+R.symbol,L,R)
        nodes.pop(0)
        nodes.pop(0)
        nodes.append(((L.frequency+R.frequency,L.symbol+R.symbol),newnode))
    return newnode

# huffman code
'''a 111
b 000
c 001
d 10
e 110
f 01'''

s = 'aaaacccbbddddddeeeffffff'
cipher = input()
res = Huffman(s)
print(decode(res,cipher))
```

Public Test case

Input 1

```
00011100011100000011011001
```

Output

```
bababcdef
```

Input 2

```
11100000110110011110000011011001
```

Output

```
abcdefabcdef
```

Input 3

```
01110100010001111110000011011001
```

Output

```
fedcbaabcdef
```

Private Test case

Input 1

```
011101000100011111100000110110011110000011011001110000011011001
```

Output

```
fedcbaabcdefabcdefabcdef
```

Input 2

```
11111111111111111111111111111111
```

Output

```
aaaaaaaaaaa
```

Input 3

```
10011001100110011001100
```

Output

```
dfdfdfdfdfdf
```

Week-7 Live coding problem 2

Write a method `MaxValueSelection(items, C)` that accepts a dictionary `items` where each key of the dictionary represents the item name and the corresponding value is a tuple `(number of units, value of all units)` and function accept one more variable `C` which represents the maximum capacity of units you can select from all items to get maximum value.

Sample input

```
{1:(10,60),2:(20,100),3:(30,120)}
50
```

Output

```
240.0
```

Solution

Solution Code

```
def MaxValueSelection(items, C):
    itemlist = []
    for i,j in items.items():
        itemlist.append((j[1]/j[0],i,j[0]))
    itemlist.sort(reverse=True)
    maxvalue = 0
    for itm in itemlist:
        if C > itm[2]:
            maxvalue += itm[0]*itm[2]
            C = C - itm[2]
        else:
            maxvalue += C*itm[0]
            C = 0
            break
    return maxvalue
```

Suffix code(Visible)

```
items = eval(input())
C = int(input())
print(round(MaxValueSelection(items, C),2))
```

Public Test Case

Input 1

```
{1:(10,60),2:(20,100),3:(30,120)}
50
```

Output

```
240.0
```

Input 2

```
{1:(6,110),2:(7,120),3:(3,2)}
10
```

Output

```
178.57
```

Input 3

```
{1:(4,400),2:(9,1800),3:(10,3500),4:(5,4000),5:(2,1000),6:(1,200)}
15
```

Output

```
7800.0
```

Input 4

```
{1:(4,400),2:(9,1800),3:(10,3500),4:(20,4000),5:(2,1000),6:(1,200)}
20
```

Output

```
6100.0
```

Input 5

```
{1:(4,1600),2:(9,2700),3:(10,3500),4:(5,4000),5:(2,1000),6:(2,1200),7:
(2,1350),8:(9,1800),9:(10,2300),10:(5,1530),11:(2,100),12:(1,120),13:
(2,1600),14:(3,2700),15:(7,3500),16:(8,4000),17:(1,1000),18:(6,1200),19:
(1,1350),20:(10,1800),21:(2,2300),22:(10,1530),23:(4,100),24:(1,125)}
1
```

Output

```
1350.0
```

Private test case

Input 1

```
{1:(4,400),2:(9,1800),3:(10,3500),4:(5,4000),5:(2,1000),6:(1,200)}
8
```

Output

```
5350.0
```

Input 2

```
{1:(4,1600),2:(9,2700),3:(10,3500),4:(5,4000),5:(2,1000),6:(2,1200),7:
(2,1350),8:(9,1800),9:(10,2300),10:(5,1530),11:(2,100),12:(1,120)}
25
```

Output

```
12650.0
```

Input 3

```
{1:(4,1600),2:(9,2700),3:(10,3500),4:(5,4000),5:(2,1000),6:(2,1200),7:
(2,1350),8:(9,1800),9:(10,2300),10:(5,1530),11:(2,100),12:(1,120)}
10
```

Output

```
7050.0
```

Input 4

```
{1:(4,1600),2:(9,2700),3:(10,3500),4:(5,4000),5:(2,1000),6:(2,1200),7:
(2,1350),8:(9,1800),9:(10,2300),10:(5,1530),11:(2,100),12:(1,120),13:
(2,1600),14:(3,2700),15:(7,3500),16:(8,4000),17:(1,1000),18:(6,1200),19:
(1,1350),20:(10,1800),21:(2,2300),22:(10,1530),23:(4,100),24:(1,125)}
30
```

Output

```
21500.0
```

Input 5

```
{1:(4,1600),2:(9,2700),3:(10,3500),4:(5,4000),5:(2,1000),6:(2,1200),7:
(2,1350),8:(9,1800),9:(10,2300),10:(5,1530),11:(2,100),12:(1,120),13:
(2,1600),14:(3,2700),15:(7,3500),16:(8,4000),17:(1,1000),18:(6,1200),19:
(1,1350),20:(10,1800),21:(2,2300),22:(10,1530),23:(4,100),24:(1,125)}
2
```

Output

```
2500.0
```

Week-7 Live coding problem 3

Write a function `IsCodeValid(hfcode, message)` that accept a dictionary `hfcode` in which key represents the character and corresponding value represents the Huffman code for that character and function accept one more string `message` (encoded message generated using Huffman codes). The function returns `True` if `message` is valid, otherwise return `False`.

Sample Input

```
{'a':'000', 'b':'0010', 'c':'0011', 'd':'01', 'e':'10', 'f':'11'} #huffman code
101010110101000000010011 #Encoded message
```

Output

```
False
```

Sample Input

```
{'a':'000', 'b':'0010', 'c':'0011', 'd':'01', 'e':'10', 'f':'11'}
111010111001011100011
```

Output

```
True
```

Solution

```
def IsCodeValid(hfcode,message):
    emsg = ''
    huffcode ={}
    maxlength=0
    for i,j in hfcode.items():
        huffcode[j]=i
        if len(j) > maxlength:
            maxlength=len(j)
    cd = ''
    for b in message:
        cd += b
        if len(cd) > maxlength:
            return False
        if cd in huffcode:
            emsg += huffcode[cd]
            cd = ''
    if cd == '':
        return True
    else:
        return False
```

Suffix Code(visible)

```
hfcode = eval(input())
message = input()
print(IsCodeValid(hfcode,message))
```

Public Test case

Input 1

```
{'a':'000', 'b':'0010', 'c':'0011', 'd':'01', 'e':'10', 'f':'11'}
10101011010100000010011
```

Output

```
False
```

Input 2

```
{'a':'000', 'b':'0010', 'c':'0011', 'd':'01', 'e':'10', 'f':'11'}
111010111001011100011
```

Output

```
True
```

Input 3

```
{'a':'111', 'b':'000', 'c':'001', 'd':'10', 'e':'110', 'f':'01'}
11100000110110010111010001000111
```

Output

```
True
```

Private Test case

Input 1

```
{'a':'111', 'b':'000', 'c':'001', 'd':'10', 'e':'110', 'f':'01'}
11100000110110010111010000010001111
```

Output

```
False
```

Input 2

```
{'a':'111', 'b':'000', 'c':'001', 'd':'10', 'e':'110', 'f':'01'}
1101110111111101010111110110
```

Output

```
True
```

Input 3

```
{'a':'111', 'b':'000', 'c':'001', 'd':'10', 'e':'110', 'f':'01'}
111011
```

Output

```
False
```