PROBLEM 1)

```python
#globally allocate dp to avoid recursion pains
dp = []

#check if subString ss is a palindrome using python recursion
def isPalindrome(ss):
    return ss == ss[::-1]

#input string to check, l == left index == 0, r == right index ==
len(string) - 1
#returns the number of slices needed to have every substring be
palidromic, + 1 to get the # of substrings instead of the number of cuts
def minPSubstrCnt(string, l, r):

    #if our current substr is one char or is a palindrome or empty
    if l >= r or isPalindrome(string[l:r + 1]):
        return 0


    # we have calculated this subset before!
    if dp[l][r] < r - l:
        return dp[l][r]

    for k in range(l, r):
        # count = the min number of total cuts given a cut at index k
        count = (
            1 + minPSubstrCnt(string, l, k)
            + minPSubstrCnt(string, k + 1, l)
        )
        #set the dp to the minimum we could find this iteration
        dp[l][r] = min(dp[l][r], count)

    #return the lowest for this subset now after calculations
    return dp[l][r]


def helperfunc(string):
    global dp
```

```
    for r in range(len(string)):
        dp.append([])
        for c in range(r):
            dp.append(r - c)

    return minPSubstrCnt(string, 0, len(string) - 1) + 1 # number of min palindromic substrings

    # allocates dp to my description below

# this runs a O(n^2) time complexity as allocating DP is n * n / 2 in size and we have to check n * n recursions at worst
# To do so I store a DP[l][r] == minPSubstrCnt(string, l, k) which would be set initially to DP[l][r] == r - l as that is the max num of cuts at length r - l + 1
# dp[l][r] stores subproblems so I do not have to recalculate them within my recursion
```

Problem 2)

Smooth shuffle can be calculated in O(n) linear time complexity.

If X / 2 + 1 > Y or Y / 2 + 1 > X then return false as a smoothstack is impossible

Start by setting X, Y, and Z as stacks representing their string counterparts.

isValidSmoothStack(X, Y, Z, on = X, count = 0):

Xtop = X.pop();

Ytop = Y.pop();

```
While !Z.empty() {
        Count ++;

        Ztop = Z.pop();

        isY = Ztop == Ytop
        isX = Ztop == Xtop
```

```
if(count == 2
	if(onX){
		if(!isY){
			Return false;
		} else {
			On = Y;
			Count = 0;
		}
	} else { //onY
		if(!isX){
			Return true;
		} else {
			On = X;
			Count = 0;
		}
	}
} else

//settle both having this one
If isY && isX {
	Temp = count;


	Temp = count;
	if(!on == X){
	On = X;
	 Temp = 0;
	}

	Xs = X.pop();
	Xside = isValidSmoothStack(X, Y, Z, on, temp);

	if(!on==Y){
		On = Y;
		Count = 0;
	}

	X.push(Xs);
	Y.pop();
	Return Xside || isValidSmoothStack(X, Y, Z, on, count);
	//if either work;
} else {
	if( isX){
		x.pop()
```

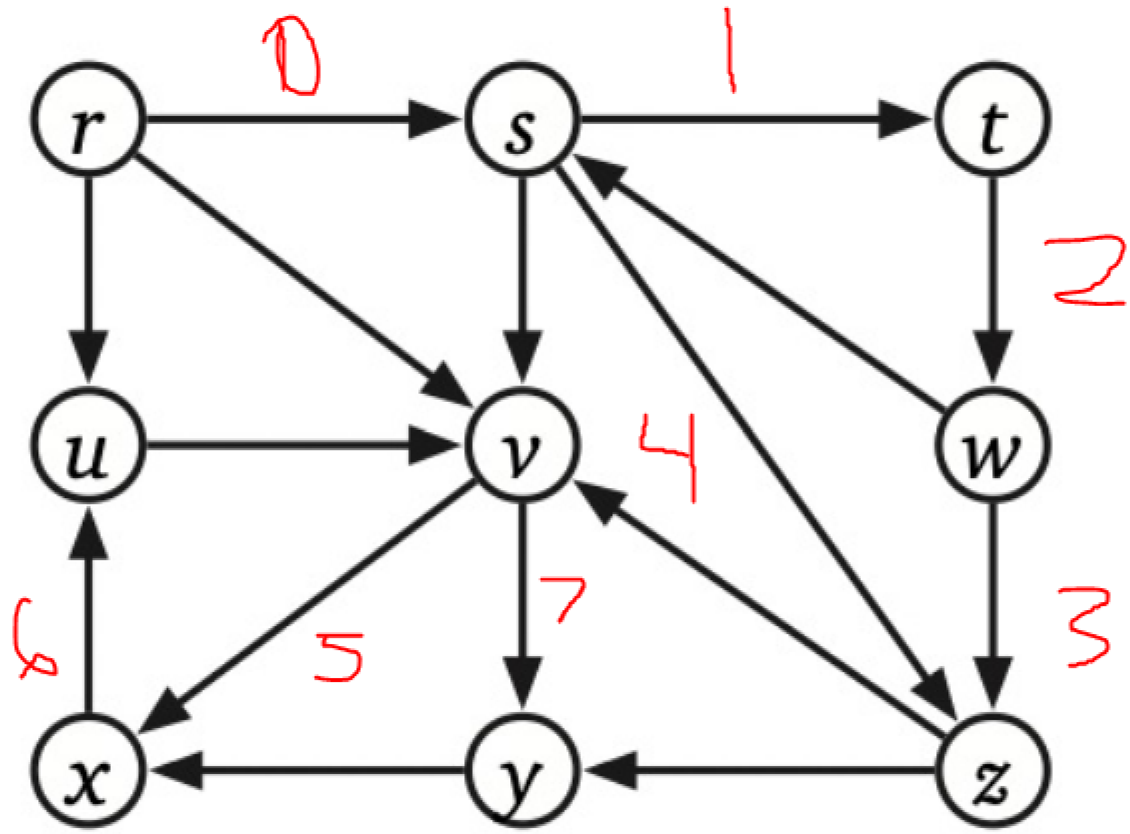
```
                if(on != x){
                        On = x;
                        Count = 0;
                }
        } else {
                y.pop()
                if(on != y){
                        On = y;
                        Count = 0;
                }
        }
    }
}
```

Return True; //at LAST!

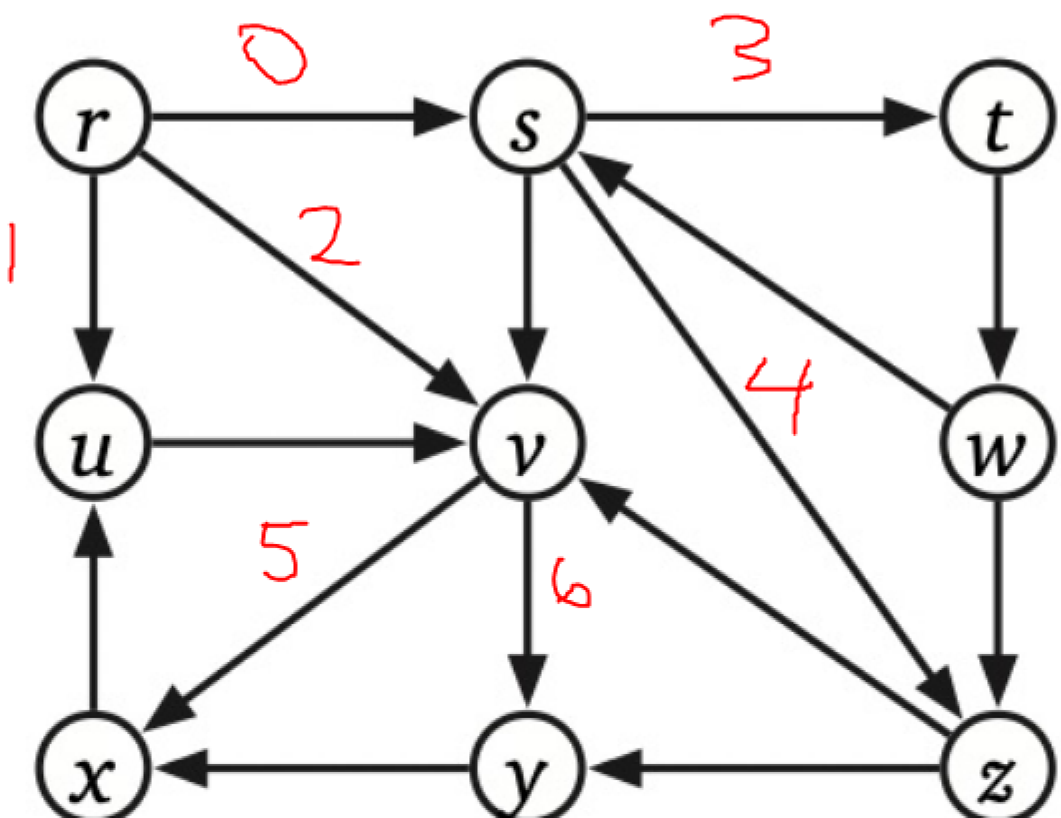
This will recursivley split to solve ties and run in O(n) time and at worst O(N log N) at max splits

Problem 3)

a) Depth First Spanning Tree rooted at R



b) Breadth First Spanning Tree rooted at R



c) NONE

As Topological Graphs MUST be Directed Acyclic Graphsm and this graph has a cycle

d) R is the only strongly connected component as no other node connects to R
And R connects to all other components

Problem 4)

```
def getNext (u, X, Y, v):
    max = v;
    for i in range(X):
        if(X[i] > u.x and Y[i] > u.y and X[i] <= v.x and Y[i] <= v.y and
length(X[i],Y[i], v.x, v.y) > length(max.x, max.y, v.x, v.y)):
            max = X[i], Y[i]

    return max



def maxMono(X, Y, u, v):

    l = 0

    while u != v:
        nextnode = getNext(u, X, Y, v);
        l += length(u.x, u.y, nextnode.x, nextnode.y)

    return l
```

This algo runs in O(n^2) time complexity and is technically a greedy algorithm as it simply selects the node furthest from v that is still monotonically increasing from the current node that can still reach V afterwards, this will always return the furthest result as the algo will always try to stay away from V while being forced to go to it.

Problem 5)

```
def canwalk(u, w, G):

    #assuming that G is an adjacency list and u is a vertex
    #(index in G which points to a list of adjacent vertecies)

    visited = [False] * len(G) #init all non visited

    # I am going to use DFS because it is optimal in this case
    # V are vertecies and u is the starting point

    Open = [u]

    while len(Open) > 0:

        u = open.pop()

        if(u == w):
            return True

        for v in G[u]:
            #v is a connected vertex

            if(visited[v]):
                continue

            visited[v] = True

            for w in G[v]:

                if(visited[w]):
                    continue

                if(angle(u, v, w) < 90):
                    Open.append(w)#we can outright skip V and "visit both"
                    visited[w] = True
    return False
```