1. What is the running time of DFS if the graph is given as an adjacency list and adjacency matrix? Justify your running time.

If the graph is given as adjacency list:

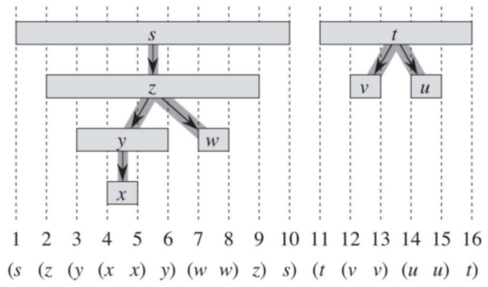
- Each node have a list of all its neightbour edges. Assume there is V nodes and E edges in the graph.
- We need to traverse its adjacency neighbours for each nodes.
- For a directed graph, the time complexity in this case is O(v) + O(E) = O(V + E)
- For a un-directed graph, each node appears twice, the time would be O(V) + O(2E) = O(V + E)

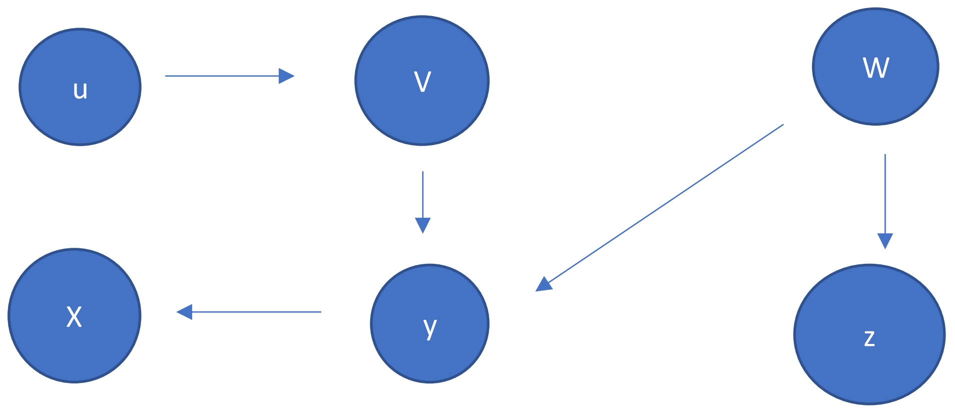
- If the graph is represented as an **adjacency matrix** (a V x V array):
    - For each node, we will have to traverse an entire row of length V in the matrix to discover all its outgoing edges. The Time complexity would be **O(V * V) = O(V ^ 2)**.

2. Write a method that takes any two nodes *u* and *v* in a tree *T*, and quickly determines if the node *u* in the tree is a *descendant* or *ancestor* of node *v*

We can solve this problem using depth first search of the tree. While doing dfs we can observe a relation between the order in which we visit a node and its ancestors. If we assign in-time and out-time to each node when entering and leaving that node in dfs then we can see that for each pair of ancestor-descendant the in-time of ancestor is less than that of descendant and out-time of ancestor is more than that of descendant, so using this relation we can find the result for each pair of node in O(1) time.

3. Draw the parenthesis structure of the dfs of Figure 1 (start from u, assume that DFS considers vertices in





4.Subsets.

Given an ordered integer array *'nums'* of unique elements, write the pseudo code of a function to return *all possible subsets (the power set)*.The solution set must not contain duplicate subsets. You should use the DFS algorithm. You can return the solution in any order.

Example:

nums = [1, 2, 3]

Output :[[], [1], [2], [3], [1, 2], [1, 3], [2, 3], [1, 2, 3]]

```
def subsets(self, nums: List[int]) -> List[List[int]]:
        res = []
        subset = []

        def dfs(i):
            if i >= len(nums):
                res.append(subset.copy())
                return
            subset.append(nums[i])
            dfs(i+1)

            subset.pop()
            dfs(i+1)

        dfs(0)
        return res
```