Q1) Tree:

- 0:<{},[A,B,C,D],0,5>
  - 1:<{A},[B,C,D],6,2>
    - 2:<{A,B},[C,D],13,-1>
    - 3:<{A},[C,D],6,2>
      - 4:<{A,C},[D],14,0>
      - 5:<{A},[D],6,2>
        - 6:<{A,D},[],15,-3>
        - 7:<{A},[],6,2>
  - 8:<{},[B,C,D],0,5>
    - 9:<{B},[C,D],7,2>
      - 10:<{B,C},[D],15,0>
      - 11:<{B},[D],7,2>
        - 12:<{B,D},[],16,-3>
        - 13:<{B},[],7,2>
    - 14:<{},[C,D],0,5>
      - 15:<{C},[D],8,3>
        - 16:<{C,D},[],17,-2>
        - 17:<{C},[],8,3>
      - 18:<{},[D],0,5>
        - 19:<{D},[],9,0>
        - 20:<{},[],0,5>

Date: ____________

Question 1

Explanation:
As per the construction of the rooted binary tree, in a left-first depth-first order, the left node denotes the consequence of choosing a subsequent item from the yet-to-be chosen list of items. If the knapsack can't hold any more items in the current node, we move on to the right child in the parent node, which denotes the consequence of not having chosen that item, thus discarding it and moving onto the next item in the list of items. Then from the constructed tree, we discard all nodes that had exceeded the capacity of the knapsack, as they are the invalid nodes. Then from the remaining nodes, the optimal solution is the node which has the highest value, which in our case, is node 10 with a value of 15, having items B and C.

Date: ...

Q2) n buildings 1...n n-1 two way streets.
Street s_i 2 (a_i, b_i) connects buildings a_i & b_i
$1 \le a_i \le b_i \le n$ $1 \le i \le n-1$

We model the city as a tree/graph structure where each building can be represented as a node, & each street is an edge connecting two nodes. We can then create an adjacency list from the list of streets. Then a DFS can be performed on our tree to find the minium umber of lamps required to light all the streets. Dynamic Programming can be used to store the umber of lamps required to light the subtree rooted at each node. We are confident that a cycle will not arise since we have n nodes, n-1 edges & all nodes can be reached from any other node. Then our ~~pseudocode~~ algorithm is as follows:

```
min_lamps(n, streets)
    adj_lst = [[] for i ← 0 to n+1]
    for a,b in streets do
        add b to adj_lst[a]
        add a to adj_lst[b]
    end for
    ~~dp ← [0]*(n~~
    dp0 2. [0]* (n+1)  // an array of n+1 length filled with 0s
    dp1 2 [0]* (n+1)
    function dfs(node, parent)
        dp0[node] ← 1
        for neighbor in adj_lst[node] do
            if neighbor is not equal to parent
                dfs(neighbor, ~~parent~~ node)
                dp0[node] += min(dp0[neighbor], dp1[neighbor])
                dp1[node] += dp0[neighbor]
    end function.
    dfs(1, -1)  // -1 so that first node/root has no parents
    return min(dp0[1], dp1[1])
end function.
```

Date:

(b) Building the adjacency list takes $O(n)$ time, & so does iterating over the list of streets which has $n-1$ elements. DFS takes linear time [$O(n)$] ~~on each instance~~, since each node is visited exactly once, during which the smaller subtrees are also solved and stored inside our 2 tables dp0 & dp1, which means that no recalculation has to be done for each node, thus takes constant time. Hence, the overall runtime complexity is $O(n)$.

Q3) Longest Palindrome Subsequence

We keep a 2D table while using a bottom up manner to store all optimal solutions for each subproblem for the larger subproblem. Each index i, j of the table stores the length of the longest palindrome subsequence for the substring starting at index i, & ending at j. When either i or j is 0, the longest subsequence will also be 0 which is the base case. The algorithm is as follows:

```
lps(seq)
    rev = reverse(seq)
    table = [[0]* rev.length for i <- 0 to (seq.length +1)]
    for i <- 1 to seq.length +1 do
        for j <- 1 to rev.length +1 do
            if seq[i-1] is equal to rev[j-1]
                table[i,j] = table[i-1, j-1] +1
            else
                table[i,j] = max(table[i-1,j], table[i,j-1])
    return table[seq.length, rev.length].
end function.
```

Reversing the sequence takes $O(n)$ time. The table is initialized in $O(n^2)$ time (where n is the length of the string sequence). The loops take $O(n^2)$ time in total, & updating each index of the table takes constant time).

Hence the total runtime of the algorithm is $O(n^2)$.

Date: ____________

Question 4

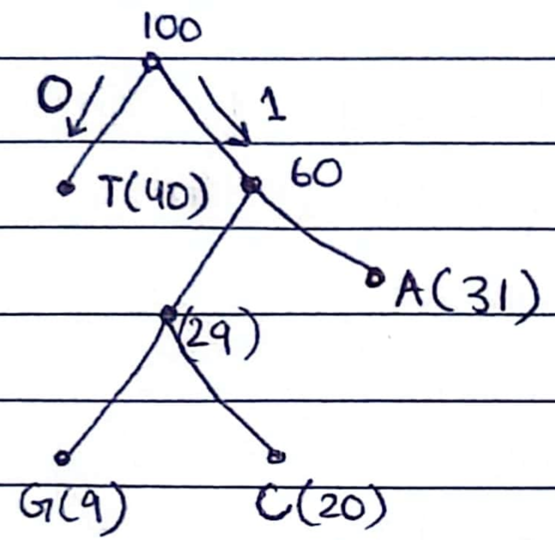
a) The uniqueness of edges' weights in G1 implies that every edge k has a distinct weight. Let's denote these weights as "a" and "b" for two edges, where "a" < "b". Since a and b are both positive, it follows that $a^2 < b^2$. Prim's algorithm works by iteratively adding the edge with the smallest weight to the tree. In G2, where the weights are the squares of the corresponding edge's weights in G1, the weight of edge b will be greater than the weight of edge a. Consequently, both corresponding edges will be added in G2 as well, and the minimum spanning trees obtained will be identical in both G1 and G2.

b) Dijkstra's algorithm considers the entire path from source vertex s to destination vertex t. The difference in edge weights between G1 and G2 can lead to different shortest paths. Consider G1 where a path from s to t has a single edge with weight 6. Additionally, there exists an alternate path from s to t with two edges, each with weight 4. Dijkstra's algorithm in G1 selects the first path. Since 6 is less than 4+4=8. However, in G2, the first path has an edge weight of 36, while the second path has a total edge weight of 16+16=32. Hence, the algorithm in G2 will select the second path. This counterexample disproves the equivalence of the shortest paths obtained in G1 and G2.

Date: ____________

Q5: Yes, It's possible to have 0 min-edit distance if and only if the word is a palindrome that means it should read the same forward and backward, for example; WOW, deed, noon etc are palindromes. such words will not require any edits to transform it into it's reverse.

Q7:



G(9) C(20)

Sym	Freq	code
A	31%	11
C	20%	101
G	9%	100
T	40%	0

Q8: Yes, solving fraction knapsack problem using dynamic programming is an overskill that's true because in dynamic programming we solve all the sub-problems first and then we look for the main problem hence, it will take nW in Worst-case where n is the number of items and W is the Weight/capacity. On the other hand using a more typical approach of knapsack we can solve fractional knapsack problem through a greedy approch/algorithm. In the greedy algorithm

Date: ________

We will use a sorting algorithim, where we sort the items based on value to weight ratio such that higher ratio will be sorted first (~~ascen~~ descending order) so we can take the max first. In this we have to iterate over this once and we are guaranteed the optimal solution. Some times we not even have to go through ~~the~~ each item so, We can reduce it to O(1) ~~lin~~ or just $\Theta(n)$ in this case otherwise in worst case it will be nlogn. So, yes O(nlogn) is better than O(nW).

Q9: Algorithm for FindPartyInvitees # denotes comments

Input n # n: number of people

Input pairs # a list of pairs representing people who know eachother

Output invitees # a list of people invited to the party

- Initialize array aquaintances_Count of size n to store number of aquaintances each person has among potential invitees. Set all counts to 0 initially.

- For each pair of people (person1, person2) in the list pairs:
  - Increment aquaintances_Count [person1] by 1
  - Increment acquaintances_Count [person2] by 1

- Initialize an empty list invitees to store the party invitees.

- Repeat Until no more people can be invited:

  a. Find a person who has the maximum number of aquaintances among the potential invitees but hasn't reached the limit of five acquaintances at the party.

Date:

- Let max_acquaintances_count = 0
  Let max_acquaintances_person = None

For each person in the range 0 to n-1:
if acquatainces_count[person] > max_acquaentance_count and ~~acquaintance~~ ~~acquaintances_count[person] > acquaintance~~ acquaintance_count[person] < 5:
Set max-acquaintance_count = 0
Set max-acquaintance_person = None
For each person in the range 0 to n-1:
if acquaintance_count[person] > max_acquaintance_count and acquaintaince_count[person] < 5:
Set max-acquaintance-count = acquaintances_count[person]
Set max_acquaintance_person = person

If max-acquaintance-person is None, break the loop as no more people can be invited.

Add max-acquaintances_person to the invitee list

Update the acquaintance counts for all people who know the newly invited person.
- For each person in the list pairs:
  - If max-acquaintances_person is one of the pair:
    Decrement acquaintances_count of others person in the pair by 1

Return list of invitees

Date:

Run time analysis:

- constructing the adjacency list will take $O(m)$
- Updating acquaintance counts will take $O(m)$ too

  so, total for this is $O(m) + O(m) = O(m)$

- Selecting Invitees:

  $O(n)$ for iterations

  $O(1)$ for comparisons so $O(n) \times O(1) = O(n)$

Overall the complexity can be expressed as $O(m+n)$; however since m can be as large as $O(n^2)$ in the worst case the time complexity will then be $O(n^2+n)$ so $O(n^2)$

Run time for this algorithm is $O(n^2)$