Zuha Aqib 26106

Question 1

Sources: `https://leetcode.com/problems/maximum-profit-in-job-scheduling/description/`

So we have n buyers, where each buyer wants to rent my property for a price of v_i for the time period $[x_i, y_i]$. We can assume this interval lies between $[0, 1]$. I want to maximise selecting such buyers that I get the most money - and there are no two overlapping buyers at the same time. This is called the **Weighted Interval Scheduling** problem.

So lets solve this question normally to understand the problem:

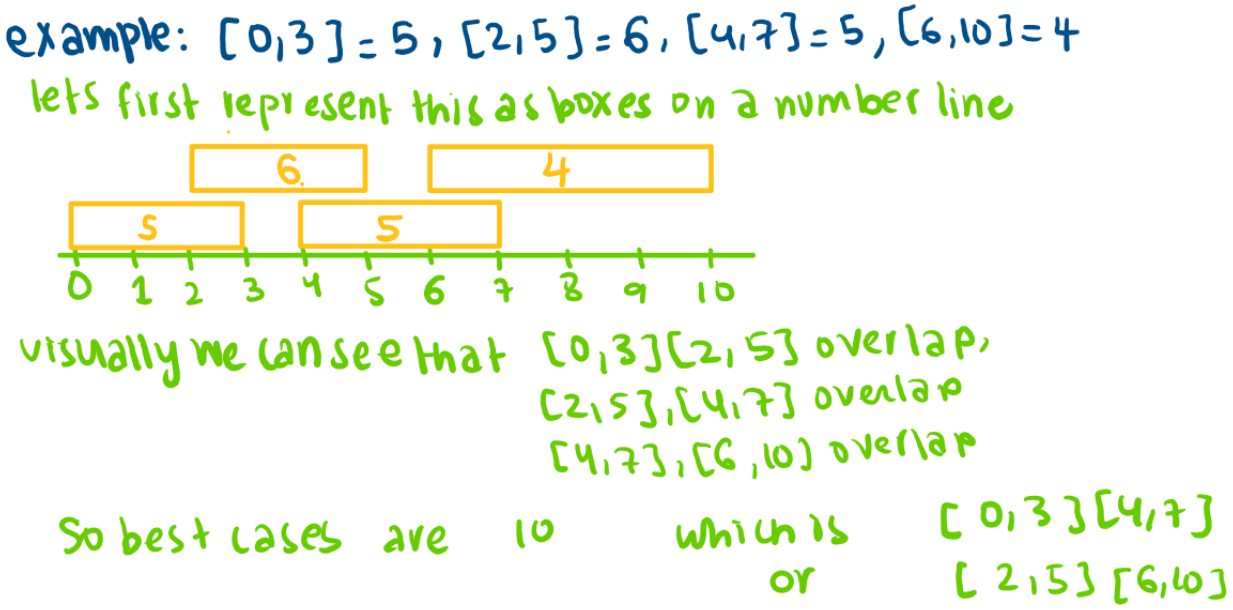


Figure 1: Enter Caption

But this is not always the case. What if i have 100's of buyers - I can't always solve them "visually" using a number line. Thus I need some kind of technique to solve them. While solving them visually, I was making paired combinations in my head and trying to find the best one. This can be done more properly:

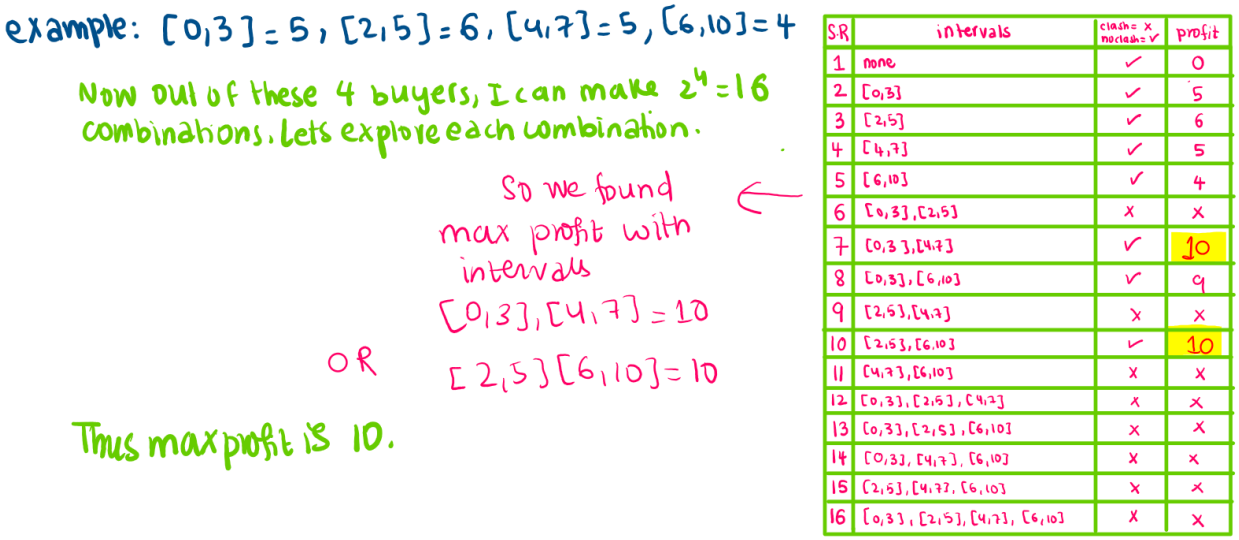


Figure 2: Practice of scheduling using brute-force

We call this the Brute Force Solution. If i have a set of intervals of size n, I can make exactly 2^n different combinations. Each combinations take $O(1)$ time. If i omit the empty set combination, this gives me $2^n - 1$ combinations, and thus a runtime of $O(2^n - 1) = O(2^n)$. We can state this algorithm more formally:

Brute-Force Exponential Algorithm:

```
# This code may not be runnable as it is pseudocode
def is_non_overlapping(subset):
    # first sort them in start basis, to have a single loop
    subset = sort(subset, value=subset[i][0])

    # now iterate over all intervals and check if its end date is after the start date
    for i=0 to subset.length: # subset.length is inclusive
        if subset[i][1] > subset[i+1][0]:
            # if the end date is after the next one's start date, its an overlap
            return False

    # if until now no overlap is found, return True
    return True

def max_profit_brute_force (intervals):
    # the input will be in the form [(start, end, profit), ...]
    n = intervals.length
    max_profit = 0

```

```
    # first create all possible combinations, aka the powerset (2^n)
    combinations = create_combinations(intervals)

    # iterate over all combinations to find the max profit
    for combination in combinations:
        if is_non_overlapping(combination) == True:
            # if it doesnt overlap, compute the profit of this combination
            total = 0
            for j=0 to combination.length: # combination.length is inclusive
                total += combination[i][2] # start=0, end=1, profit=2
            # now that we have profit, re-compute the max
            max_profit = max(max_profit, total)

    return max_profit
```

Listing 1: Algorithm for Brute Force Solution

But this is very work intensive as many of the combinations can be deleted before looking at them. Thus we implement a dynamic programming solution, in which we remove every interval that clashes at the start. First we sort the intervals on the basis of end time, then we add the intervals on their basis. An algorithm to soldify the understanding:

Dynamic Programming n^2 Algorithm:

```
# This code may not be runnable as its a pseudocode
def max_profit_dp(intervals):
    # the input will be in the form [(start, end, profit), ...]
    # first sort them in endbasis
    intervals = sort(intervals, value=intervals[i][1])

    # create a table for dp, it has tuples (end_time, max_profit_so_far) which saves the max_profit till the time end_time.
    # at dp[i], where i is an interval, we can know what is the max profit at each interval using dp[i][0]
    dp = [(0, 0)] # base case is that at time 0, profit is 0

    # iterate over all intervals
    for i=0 to intervals.length: # intervals.length is inclusive
        # extract all intervals[i] into variables for easier reference
        start_i = intervals[i][0]
        end_i = intervals[i][1]
        profit_i = intervals[i][2]

        # now find the tuple in dp that ENDS before this interval STARTS i.e. dp[j][0] <= start_i
        profit = 0
```

```
        for dp_i in dp:
            # for every dp_interval, if the end_time is BEFORE the start_i, then save its profit value.
            if dp_i[0] <= start_i:
                # if profit has already been assigned, pick the higher one
                profit = max(dp_i[1], profit)

        # previous profit is the profit at the previous interval which is just "profit"
        # new profit is the previous profit plus the current profit
        new_profit = profit + profit_i

        # add this new profit and its end time to dp
        dp.add((end_i, new_profit))

    # after all are traversed, return the profit of the last one at the end
    return dp[dp.length-1][1]
```

Listing 2: Algorith for Dynamic Programming Solution

So a small summary of the algorithm is to first sort according to end times. And get a dp-array, which will save the max profit possible at the end of each interval. This algorithm is incremental as it is looks behind, finds the best possible interval, and copies its profit forward. So after I have sorted and created an array (with the base case 0,0 in it), I start traversal: i get each interval, and search the dp array for the RIGHTMOST non-overlapping value. i.e. i try and find the HIGHEST POSSIBLE end_time that ends BEFORE my specific interval starts. I get its profit, and insert in dp (end_time_of_current_interval, profit_of_found + profit_of_current_interval). I continue this loop until all intervals are traversed, and then I return the last value in dp.

Lets practice this algorithm using an example:

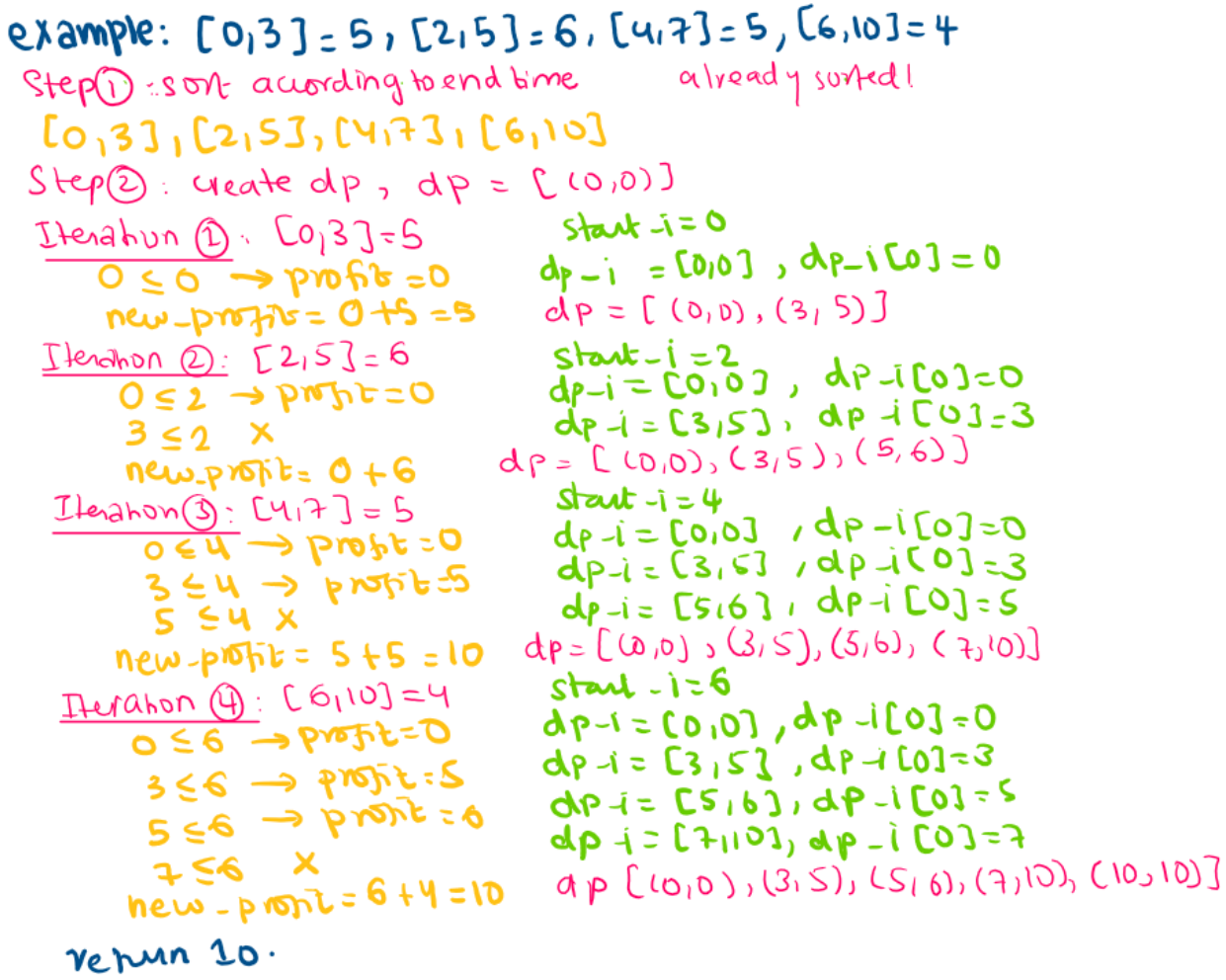


Figure 3: Example of Dynamic Programming Algorithm

But this is still not good - at each interval i, for n intervals, I am making i comparisions with the dp array, which gives me

$$1 + ... + n \text{ work in total}$$

which is equal to $\frac{n(n+1)}{2}$ which has complexity $O(n^2)$.

I can make this better. Instead of traversing the entire dp array, I can perform a binary search on it, searching only the RIGHT of each half. This reduces my work done at each n interval, as binary search is of only $O(logn)$ work.
An updated algorithm:

Dynamic Programming with Binary Search *nlogn* Algorithm:

```
# This code may not be runnable as its a pseudocode
def custom_bisect_right(dp, value):
    """Find the index of the last dp[i][0] <= value using binary search"""
```

```
    # so here we are trying to find the RIGHTMOST value that ends before value STARTS
    low = 0
    high = len(dp)

    while low < high:
        mid = (low + high) // 2
        if dp[mid][0] <= value:
            low = mid + 1
        else:
            high = mid

    return low  # This is like bisect_right

def max_profit_dp(intervals):
    # the input will be in the form [(start, end, profit), ...]
    # first sort them in endbasis
    intervals = sort(intervals, value=intervals[i][1])

    # create a table for dp, it has tuples (end_time, max_profit_so_far) which saves the max_profit till the time end_time.
    # at dp[i], where i is an interval, we can know what is the max profit at each interval using dp[i][0]
    dp = [(0, 0)] # base case is that at time 0, profit is 0

    # iterate over all intervals
    for i=0 to intervals.length: # intervals.length is inclusive
        # extract all intervals[i] into variables for easier reference
        start_i = intervals[i][0]
        end_i = intervals[i][1]
        profit_i = intervals[i][2]

        # now find the tuple in dp that ENDS before this interval STARTS i.e. dp[j][0] <= start_i using binary_right_search
        # this is a defined method, it takes the array as input, performs a binary search always on the right side, if it has exceeded, it sends the last possible value
        profit = custom_bisect_right(dp, start_i)

        # previous profit is the profit at the previous interval which is just "profit"
        # new profit is the previous profit plus the current profit
        new_profit = profit + profit_i

        # add this new profit and its end time to dp
        dp.add((end_i, new_profit))
```

```
43
44      # after all are traversed, return the profit of the last one at the
        end
45      return dp[dp.length-1][1]
```

Listing 3: Algorith for Dynamic Programming Solution

Thus the complexity of this algorithm is that it first sorts ($O(nlog(n))$) and then traverses over all intervals ($O(n)$), and at each interval, it traverses dp - which we perform a BINARY SEARCH on dp, on the right half, to find the right-most, which results in complexity ($O(logn)$) so thus,

the total complexity is

$$O(nlogn)$$

.

Zuha Aqib 26106, Zehra Ahmed 26965, Farah Inayat 26912

Question 2

A Hamiltonian Path is a path in a graph that visits each vertex exactly once.

- If such a path exists: we say the graph has a Hamiltonian Path.
- If the path is a cycle (starts and ends at the same vertex), it's called a Hamiltonian Cycle.

A naive brute-force algorithm would be: let's check every permutation of the vertices and see if there's a path following that order.

Naive Brute Force Algorithm:

```python
from itertools import permutations
def is_hamiltonian_path(graph, n):
    # graph is an adjacency list.
    # n is number of vertices
    for perm in permutations(range(n)):
        valid = True
        for i in range(n - 1):
            if perm[i+1] not in graph[perm[i]]:
                valid = False
                break
        if valid:
            return True
    return False
```

Listing 4: Naive Brute Force Algorithm

The time complexity of brute force is $n!$ permutations, for each we check $n-1$ edges, so $O(n!) * O(n) = O(n! * n)$ which becomes infeasible for $n > 10 - 12$.

Instead of blindly trying every permutation, we build the solution incrementally using Dynamic Programming. For each subset S of visited nodes and each ending vertex $v \in S$, we track whether there exists a path that:

- visits all nodes in S exactly once
- ends at vertex v

We define:

$$dp[S][v] = \text{True if there is a path visiting all nodes in } S \text{ and ending at } v$$

We use **bitmasking** to efficiently represent subsets of nodes. For n nodes, there are 2^n subsets, and for each subset, we can end at any of the n vertices. So we use a 2D array of size $[2^n][n]$. We

reduce $n!$ time to $O(n^2 2^n)$ using bitmasking + memoization.

Define dp[mask][u]:

- *mask* is a bitmask representing the set of visited nodes.
- u is the current (ending) vertex.
- $dp[mask][u] = True$ if there's a path that visits exactly the nodes in *mask* and ends at u.

A better explanation of the algortihm is that,

- Initialize: For each node v, the path that starts and ends at v (only v in the path) is valid. So, $dp[2^v][v] =$ True
- Transition: For each subset *mask* and each vertex u in that subset, try to extend the path to any neighbor v not yet in *mask*.
- Final Answer: Check if any $dp[2^n - 1][v] =$ True for some v

We also reconstruct the actual path using a `parent` table.

Optimized Algorithm:

```python
def hamiltonian_path(graph, n):
    from collections import defaultdict

    dp = [[False] * n for _ in range(1 << n)]
    parent = [[-1] * n for _ in range(1 << n)]  # To recover path

    # Base case: single node path
    for i in range(n):
        dp[1 << i][i] = True

    for mask in range(1 << n):
        for u in range(n):
            if not dp[mask][u]:
                continue
            for v in graph[u]:
                if mask & (1 << v):
                    continue
                next_mask = mask | (1 << v)
                if not dp[next_mask][v]:
                    dp[next_mask][v] = True
                    parent[next_mask][v] = u  # Store the predecessor

    # Check if there's a path that visits all nodes
    full_mask = (1 << n) - 1
```

```
    for end in range(n):
        if dp[full_mask][end]:
            # Reconstruct path
            path = []
            mask = full_mask
            u = end
            while u != -1:
                path.append(u)
                prev = parent[mask][u]
                mask ^= (1 << u)
                u = prev
            return path[::-1]  # Return reversed path
    return None
```

Listing 5: Optimized Algorithm

The time and space complexity is as follows,

- There are 2^n subsets of vertices
- For each subset, we store up to n end vertices
- For each $(mask, u)$ pair, we check up to n neighbors

$$\text{Time Complexity: } O(n^2 \cdot 2^n)$$

$$\text{Space Complexity: } O(n \cdot 2^n)$$

This dynamic programming approach drastically reduces the search space from $n!$ permutations to 2^n subsets with polynomial overhead. By using bitmasks to track visited vertices and storing subpath information efficiently, we achieve a much faster (yet still exponential) algorithm that can handle graphs up to around $n = 20$. Moreover, we not only detect the existence of a Hamiltonian Path, but also reconstruct and return the path itself.

Zuha Aqib 26106

Question 3

Sources: `https://www.cs.tufts.edu/comp/150GT/documents/Prufer%20sequences%20-%20from%20[%20Gross,%20Yellen%20]%20%20Graph%20Theory%20and%20Its%20Applications,%203e.pdf`

Beginning this question, we understand that we have a labelled acyclic graph, also known as a labelled tree.
Discussing Cayley's Formula,

$$T_n = n^{n-2}$$

where T_n is the number of different labelled trees with n vertices. This means that however many nodes or vertices i have in my tree, i can make exponential (to the power of nodes-2) different types of trees - basically the arrangement of the nodes. I can make n^{n-2} permutations of n vertices.

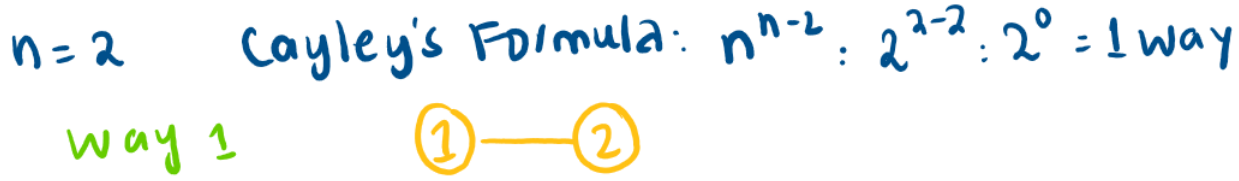


Figure 4: Displaying how $n = 2$ gives $2^{2-2} = 1$ graph

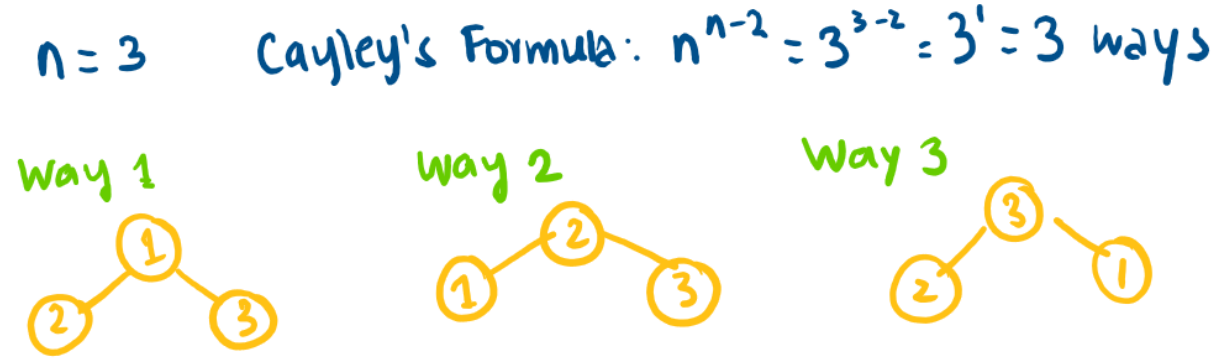


Figure 5: Displaying how $n = 3$ gives $3^{3-2} = 3$ graphs.

Now the Prüfer code says that each labelled tree corresponds to exactly one sequence of length $n-2$ and vice versa. This is a bijective (one-to-one and onto) mapping between:

- All labelled trees with n vertices, and
- All sequences of length $n-2$ with entries from $\{1, 2, ..., n\}$

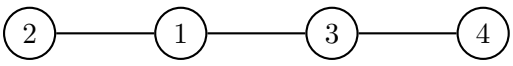
So the way to construct a Prüfer code from a graph is using an algorithm:

Algorithm to convert to Prüfer code:

```
# This is not runnable as it is in pseudocode
def labelled_to_prufer (graph):
    prufer_code = []

    # repeat for all nodes until 2 remain
    while graph.nodes.length > 2:
        # first find the smallest leaf node and its neighbour
        smallest_node = graph.node[0]
        smallest_neighbour = graph.node[0].connections[0]

        # loop on all nodes to find smallest leaf node
        for node in graph.nodes:
            # if the node is a leaf
            if node.connections.length == 1:
                # if its label is smaller than original
                if node.value < smallest_node.value:
                    smallest_node = node
                    smallest_neighbour = node.connections[0]

        # remove the smallest node in the graph
        graph.remove(smallest_node)

        # add the value of the neighbour in the graph
        prufer_code.add(smallest_neighbour.value)

    return prufer_code
```

Practice of converting to Prüfer code:

Using an example:



Iter.	# Nodes	Leaves	Smallest Leaf	Neighbour	Prüfer Code	Graph
0	-	-	-	-	-	2 — 1 — 3 — 4
1	4, possible	[2, 4]	2	1	[1]	1 — 3 — 4
2	3, possible	[1, 4]	1	3	[1, 3]	3 — 4
3	2, impossible	–	–	–	–	–

Final Prüfer code: [1, 3]

Creating graph from Prüfer code:

Now to prove that Prüfer is one-to-one, i.e. for each graph there exists one sequence and for each sequence there exists on graph, we can use the following algorithm:

Algorithm to convert graph using Prüfer code:

```
# This is not runnable as it is in pseudocode
def prufer_to_labelled (prufer_code):
    # first compute the number of nodes in the graph. as prufer ends at n-2 nodes, the number of nodes is +2 then prufer
    graph_size = prufer_code.length + 2

    # make an array of "counts" for each node appearence in prufer
    count = []

    # make an array of all labels, at the end we would have joined n-2 nodes, 2 are left, they will be connected
    labels = []

    # intiate the graph with no edges, only nodes
    for i = 1 to graph_size: # graph_size is inclusive
        # initiate node with value i and insert in graph
        node = Node(i)
        graph.add(node)
        # for that node, add 1 as base case count
        count.add(1)
        # add this node value to labels
        labels.add(i)

    # traverse the prufer code and increment count for each node
    for p in prufer_code:
        count[p-1] += 1

    # now for each value in prufer code,
    for p in prufer_code:
        # find the smallest node with count 1 (i.e. it had not appeared in the prufer_code, meaning it was the first leaf removed)
        smallest_node = graph.node[graph_size-1] # start with the last node, if smallest found, change, else last
        # traverse the count array to find the FIRST 1
        for i = 0 to graph_size-1: # graph_size is exclusive
            if count[i+1] == 1:
                # extract the node in the graph
                smallest_node = graph.node[i]
                # as soon as we find the first count=1, we do not need to find more, this is the smallest label. stop traversal
                break;
```

```
37
38          # now connect this smallest_node to the current prufer_code p
39          node_p = graph.node[p-1]
40          node_p.add_connection(smallest_node)
41          smallest_node.add_connection(node_p)
42
43          # decrement p count from count-array
44          count[p-1] -= 1
45          # decrement smallest_node from count-array
46          count[smallest_node.value-1] -= 1
47
48          # remove this smallest_node from labels
49          labels.remove(smallest_node.value)
50
51      # n-2 nodes are connected, now connect last two in labels
52      node_1 = graph.node[labels[0]-1]
53      node_2 = graph.node[labels[1]-1]
54      node_1.add_connection(node_2)
55      node_2.add_connection(node_1)
56
57      return graph
```

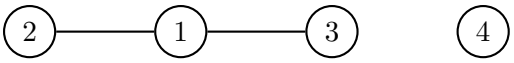
Both algorithms prove that its one-to-one, as it shows how to build the Prüfer code from a tree, and then how to reconstruct the tree from a Prüfer code.

Practice of converting to graph:

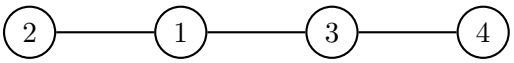
Using the same example, $[1, 3]$
number of nodes in graph is $2 + 2 = 4$

labels are $= [1, 2, 3, 4]$
inital count array $= [1, 1, 1, 1]$ i.e. all start with 1 base case
adding $+1$ to each occurence in prüfer code results in count array $= [2, 1, 2, 1]$ as 1 occurs and 3 occurs
smallest 1 found at $count[1]$ meaning $node = 2$. this means $node = 2$ was the first leaf removed. thus connect $node = 2$ with first p in prüfer which is $= 1$

now decrement count of $p = 1$ and $node = 2$
count array $= [1, 0, 2, 1]$
remove $node = 2$ from labels, labels $= [1, 3, 4]$
now pick smallest 1 again, which is found at $count[0]$ which is $node = 1$. thus connect $node = 1$ with second p in prüfer which is $= 3$



now decrement count of $p = 3$ and $node = 1$
count array $= [0, 0, 1, 1]$
remove $node = 1$ from labels, labels $= [3, 4]$
now prüfer code is empty, so connect last 2 in labels, which is $= 3$ and 4



And thus we have achieved the graph we started with.

Conclusion:

Thus Prüfer code is a bijective encoding of labelled trees of size n into sequences of length $n - 2$. This proves Cayley's formula because:

- Number of sequences of length $n - 2$ from $\{1, ..., n\}$ is n^{n-2}
- Hence, number of labelled trees with n nodes is also n^{n-2}

Zuha Aqib 26106, Zehra Ahmed 26965, Farah Inayat 26912

Question 4

First we build an understanding of the question, then move to $Min - Cut$.

Understanding:

So first to explain the problem: we have a graph, of nodes. Each node is like an atom, it can either point up (+1) or point down (-1). All these atoms have a number (edge) between them that represents how strongly they want to be aligned ($J_{ij} > 0$) i.e. both want to be up or both want to be down, or not aligned ($J_{ij} < 0$) i.e. both are in opposite directions or they have no impact or dependency on each other ($J_{ij} = 0$). In the question we have a restriction that $J_{ij} \geq 0$ thus ruling out all the "non aligned".

Each atom (node) has a magnetic field around it, called h_i. If h_i is positive, the atom wants to point up, if it is negative, the atom wants to point down, and if it is 0, the atom doesnt care.

Our goal is to choose for every atom whether it's pointing up or down, in a way that makes the total "disagreement" as small as possible. Each disagreement costs energy. We want to make atoms happy: let them match their neighbors (if J is positive) and follow their own preference (based on h).

But the problem for this is that I would have to try and assign every node according to its neighbouring node's state, J_{ij}, and h_i. This means I have to try every combinations of atoms pointing up or down to find the best one. But, one thing to note is that all J values are 0 or positive, so I will try and reduce it to the $Min - Cut$. Let us first understand the energy E formula.

$$E(\{s_i\}) = -\sum_{ij} J_{ij} s_i s_j - \sum_i h_i s_i.$$

So first lets discuss the first summation, $-\sum_{ij} J_{ij} s_i s_j$, so we have established in the question that J_{ij} is always positive, so the overall sign of the summation is dependent on s_i and s_j. Now if the signs on s_i and s_j are same, i.e. both nodes have the same spin direction (both are positive or both are negative), then the overall summation is positive, and after applying the negative sign, thus the value is NEGATIVE, thus decreasing the energy (minimized, we want this). If the signs on s_i and s_j are different, i.e. both nodes have different spin direction (one is positive and one is negative), then the overall summation is negative, and after applying the negative sign, thus the value is POSITIVE, thus increasing the energy (we don't want this).

So we don't have s_i and s_j. Our best case is that for all $J_{ij} \geq 0$, s_i and s_j is same, thus we will have the least Energy E. But this is not always the case. THUS we want to select such minimum amount of $J_{ij} > 0$ to have different signs s_i and s_j to minimize the PENALISING (increasing of energy) and maximise the reduction of energy (same signs).

So how do i select the minimum amount of $J_{ij} > 0$? Lets discuss $Min - Cut$ here. But first, lets discuss the second summation, $-\sum_i h_i s_i$. Here we want the summation to be overall positive,

as if the summation is positive, the outside negative sign will make it negative, thus REDUCING the energy (we want this). So how do we make it positive? We want h_i and s_i to have same signs, i.e. both are positive or both are negative. This also follows suit that if h_i is positive, it means the atom has a magnetic field to go up, and if its assigned s_i as $+1$, then it is good - we have assigned s_i correctly. And thus the overall sign is positive. Just like vice versa, if h_i is negative, and s_i is negative, then it is overall positive, thus a REDUCTION on the energy (we want).
But if h_i and s_i is different, then the overall sign is negative, after applying negative sign, then it gets positive. Which increases the energy - we don't want this. So we want to assign s_i the same sign as h_i. Again this is our ideal situation (best case).

So overall we want to minimize the energy. We want to

- assign all the nodes $s_i = +1$ where $h_i > 0$ and $s_i = -1$ where $h_i < 0$ and
- we want the smallest amount of $J_{ij} > 0$ edges to have different s_i and s_j

So lets first address finding the smallest amount of $J_{ij} > 0$ edges. We do this using $Min - Cut$.

Min-Cut:

So first lets convert our graph to $Min - Cut$. In our input, we are given all J_{ij}'s (the graph is undirected as said by Sir Jibran in Discord group, and all $J_{ij} = J_{ji}$), and we are given all h_i's, so we can make an directed graph using those J_{ij}'s as edges going forward and J_{ji}'s going backward and i's as vertices.

I add two nodes:
We have original nodes i. We add two extra nodes: SOURCE represented by s and SINK represented by t. I keep node s on the left and node t on the right.

How do i connect s to the original graph?
For each of the vertices i, I extract the set of all that have $h_i > 0$. I connect each of them to node s, with the edge in between as h_i.

How do i connect t to the original graph?
For each of the vertices i, I extract the set of all that have $h_i < 0$. I connect each of them to node t, with the edge in between as $|h_i|$.

What about nodes that are $h_i = 0$?
They are connected to neither s nor t.

Why did we do this type of connections?
If the atom wants to point up i.e. $h_i > 0$, connect it to the SOURCE node with cost h_i. It prefers i to be on the same side as s. We can say that the SOURCE node s represents atoms pointing up. If the atom wants to point down i.e. $h_i < 0$, connect it to the SINK node with cost $|h_i|$. It prefers i to be on the same side as t. We can say that the SINK node t represents atoms pointing down.

Find Min-Cut
Now we have a connected graph from s to t, so we perform *Ford Fulkerson* Algorithm, and find

the $Min - Cut$. Find the smallest value of edges to cut that disconnect s from t. The cost of the cut is: the sum of weights of edges that are cut (i.e., go from source side to sink side).

Division of nodes into two sets:
A cut divides the nodes into two sets: *One side* with the source $s \rightarrow$ assign those nodes $s_i = +1$. Atoms on the Source side = pointing up. This is called set A. The other side with the sink $t \rightarrow$ assign those nodes $s_i = -1$. Atoms on the Sink side = pointing down. This is called set B.

How does the mincut prove I have a minimized energy?
I have found the minimzed capacity of edges. Each of those edges may be of the three cases:

1. The edge is between s and i, thus s is in set A and i is in set B. This means that i had $h_i > 0$ and was assigned $s_i = -1$, we go against the magnetic field, increases energy, cut cost is h_i.
2. The edge is between i and j, i.e. J_{ij}, so now i and j are in different sets A and B thus having opposite signs s_i and s_j, thus increasing enegery, peanalising $2 * J_{ij}$.
3. The edge is between i and t, thus t is in set B and i is in set A. This means that i had $h_i < 0$ and was assigned $s_i = +1$, we go against the magnetic field, increases energy, cut cost is $|h_i|$.

Thus every edge we cut corresponds exactly to an increase in energy. This means the total cut cost = total extra energy. Since min-cut can be solved in polynomial time, the problem is in P for the ferromagnetic case.