1. You are given a graph G=(V, E) with a tree decomposition (X, T), as defined on Wikipedia. So for each node i of T, we have a bag Xi ⊆ V. Suppose the decomposition has width k, meaning the maximum bag size |Xi| is k+1. By merging adjacent bags if necessary, we can assume that there is no edge {i,j} in T with Xi ⊆ Xj or Xj ⊆ Xi.

(a). Argue that such a T has at most |V| nodes.

A. Given hint from discussions : Pick an arbitrary root node r in T. For each v in V, there may be multiple bags containing v. But because all these bags form a subtree of T, you can argue that among them there is a unique bag Xi which contains v and is "closest to r". (Either i=r, or else i has parent p and Xp does not contain v.) Argue that every bag Xj must be the "closest to r" bag for at least one vertex in Xj.

Algorithm :

For each tree node i, we pick a vertex V such that it belongs to X_i but not in its parents bag. Since bags X_i are subsets of V, we merged bags so there is no bag that is a subset of its parent. So each bag X_i contains a vertex v that is not in its parents bag. Such way, we can associate each node i of T with a unique vertex V. Since there are |V| vertices in G and each node in T corresponds to a unique vertex of G. So |T| ≤ |V|.

Example :

Consider the graph G with V = {1,2,3,4,5}. So a tree decomposition could look like :

T1 = {1,2}

T2 = {2,3}

T3 = {3,4}

T4 = {4,5}

This tree has 4 nodes, one for each bag X_i. We can associate N1 with 1, N2 with 2, N3 with 3 and N4 with 4. We can't associate a node N5 because vertex 5 is already included in bag X_4. From this example, |T| < |V|. We can check for n number of cases but this will be our final solution, that is |T| ≤ |V|.

(b) Suppose e={i,j} is an edge of T. If we remove e from T, then we get two trees TA and TB. Define A⊆ V as the union of all the bags in TA, and similarly define B ⊆ V as the union of bags in TB. (Note this implies A∪B=V.) Let S = Xi∩Xj. Argue that S separates A and B. That is, any path in G from a vertex u in A to a vertex v in B must include at least one vertex of S. (This is trivially true if either u or v is in S.)

A. We are given a tree T and an edge e = {i, j} in T. Removing e from T breaks the tree into two subtrees, which we call T_a and T_b . A is defined as the union of all vertices contained in the bags of T_a. B is defined as the union of all vertices contained in the bags of T_b. By construction, A and B are disjoint and their union is V (the vertex set of T). Let S = $X_i \cap X_j$, the intersection of the bags containing i and j.

To show S separates A and B:

- Consider any path P in G from a vertex u in A to a vertex v in B.
- P must contain edge e = {i, j} since removing this edge disconnects A and B in T.
- Therefore, P must contain a vertex from X_i and a vertex from X_j.
- But $X_i \cap X_j$= S, so P contains a vertex from S.
- This shows any path from A to B must intersect S, so S separates A and B as required.

(c) Explain how to modify (X, T) so that it is *smooth* (as defined on Wikipedia). This means: |Xi|=k+1 for each node i, and |Xi∩Xj|=k for each tree edge {i,j}.

A. Let k be the maximum size of any set Xi in the current partitioning. Any tree-decomposition of a graph G can be transformed to a smooth tree-decomposition of G with the same treewidth: apply the following operations until none is possible:

- For each node i where |Xi| < k+1, add arbitrary nodes to Xi until |Xi| = k+1.
- For each non-tree edge {i,j}, remove nodes from Xi and Xj until |Xi ∩ Xj| = k.

For each tree edge {i,j}:

- If |Xi ∩ Xj| > k, remove arbitrary nodes from the intersection until |Xi ∩ Xj| = k.
- If |Xi ∩ Xj| < k, add arbitrary nodes that are currently not in the intersection to the smaller set until |Xi ∩ Xj| = k.

After these steps, each set Xi will have size k+1 and each intersection of sets along a tree edge will have size exactly k. Therefore, the partitioning will now be smooth.

2) Suppose we are given a graph G=(V, E) with n vertices, and a tree decomposition (X, T) of width k. Wikipedia presents a dynamic programming approach solving the MIS (maximum independent set) problem in G. It chooses a root r in T, it defines O(2k n) subproblems, and it gives formulas to solve them all in bottom-up order. When k is a constant, the entire algorithm runs in O(n) time.

We want to redo this for 3-coloring in G. Given a subset W of V, a *3-coloring of W* is a function χ: W→{1,2,3} so that for every edge {u,v} in E that has both endpoints in W, χ(u)≠χ(v). We say G is *3-colorable* if there is a 3-coloring of V (all the vertices). Mimicking Wikipedia, we define our A and B subproblems as follows:

- Suppose i is a node in T, and s is a function s: Xi→{1,2,3}. **A(s,i)** is a boolean (true or false), telling us whether s extends to a 3-coloring χ: Di→{1,2,3}. (Di is defined on Wikipedia. "Extends" means that χ agrees with s on its domain Xi.)
- Suppose (j,i) is an edge in T, with j the parent of i, and s is a function s: Xi∩Xj→{1,2,3}. **B(s,i,j)** is a boolean, telling us whether s extends to a 3-coloring χ: Di→{1,2,3}. (Again, χ must agree with s on its domain Xi∩Xj.)

(a) Give a boolean formula computing B(s,i,j) in terms of A(s',i). Clearly describe the range of choices for s'.

A. Given a graph G = (V, E), a tree decomposition is a pair (X, T), where X = {X_1, …, X_n} is a family of subsets (sometimes called bags) of V, and T is a tree whose nodes are the subsets X_i, satisfying the following properties:

- The union of all sets X_i equals V. That is, each graph vertex is associated with at least one tree node.
- For every edge (v, w) in the graph, there is a subset X_i that contains both v and w. That is, vertices are adjacent in the graph only when the corresponding subtrees have a node in common.
- If X_i and X_j both contain a vertex v, then all nodes X_k of the tree in the (unique) path between X_i and X_j contain v as well. That is, the nodes associated with vertex v form a connected subset of T. This is also known as coherence, or the running intersection property. It can be stated equivalently that if X_i, X_j and X_k are nodes, and X_k is on the path from X_i to X_j, then $X_i \cap X_j \subseteq X_k$. Let D_i be the union of the sets X_j descending from X_i.

For each node i in T and each function s: Xi ⟶ {1,2,3}(For all : $X_j \subseteq$ V, it colors the edge with both endpoints) :

- A(s,i) = true if s can be extended to a valid 3-coloring of Di
- A(s,i) = false otherwise

For each edge (j,i) in T where j is the parent of i, and each function s: Xi ⋂ Xj ⟶ {1,2,3}:

- B(s,i,j) = true if s can be extended to a valid 3-coloring of Di
- B(s,i,j) = false otherwise

The recurrence relations are:

- For a non-leaf node i with children c1, c2, ..., ck:
  - A(s,i) = true if there exists a child cj such that B(s|Xi ⋂ Xcj, i, cj) = true
  - A(s,i) = false otherwise
- For an edge (j,i) where j is the parent of i:
  - B(s,i,j) = true if there exists a 3-coloring χ of Di that agrees with s on Xi ⋂ Xj
  - B(s,i,j) can be computed by checking if A(s',i) = true for any extension s' of s that assigns colors to Xi - Xi⋂Xj

Boolean Formula computing B(s,i,j) in terms of A(s',i). :

$$B(s,i,j) = \exists s' . (\forall v \in X_i \cap X_j . s'(v) = s(v)) \land A(s',i)$$

Where:

- s' ranges over all functions Xi ⟶ {1,2,3}
- "s' agrees with s" means ∀v ∈ Xi ⋂ Xj . s'(v) = s(v)
- A(s',i) evaluates to true/false based on subproblem definition

We consider all possible extensions s' of the partial coloring s that assign colors to the remaining vertices $X_i - X_i \cap X_j$. If any such extension s' leads to A(s',i) being true, meaning s' can be extended to a valid 3-coloring of Di, then B(s,i,j) is true. Otherwise, if no such extension s' exists, B(s,i,j) is false.

b) Give a boolean formula computing A(s,i) in terms of B(s',j,i) (where j ranges over children of i). What is your formula when i has no children?

A. Boolean formula for computing A(s,i) in terms of B(s',j,i) for node i with children j:

$$A(s,i) = \bigvee_{j \text{ child of } i}: \exists s'. (\forall v \in X_i \cap X_j.\ s'(v) = s(v)) \land B(s',j,i)$$

Where:

- j ranges over all children of node i
- s' ranges over functions $X_i \cap X_j \rightarrow \{1,2,3\}$
- "s' agrees with s" means $\forall v \in X_i \cap X_j\ .\ s'(v) = s(v)$
- B(s',j,i) evaluates to true/false based on subproblem definition

When i has no children, the problem is simplified to A(s,i) being true if s is a valid 3-coloring of Xi. Because the only way s extends to a 3-coloring of Di is if s itself is already a valid 3-coloring of Xi.

c) Estimate the time you need to decide whether G is 3-colorable, in terms of n and k. It should be O(n) when k is a constant.

A. For each node i in the tree T, there are $O(3^k)$ possible functions $s: X_i \rightarrow \{1,2,3\}$
For each s, computing A(s,i) takes O(1) time if i is a leaf. Likewise, computing each B(s,i,j) takes O(1) time.
Since T has n nodes, the total work is:

$$O(n * 3^k * 1) = O(n * 3^k)$$

As k is a constant, this is O(n).
Therefore, the overall running time to decide if G is 3-colorable is O(n) when the treewidth k is a constant.

3. Consider the following game. A "dealer" produces a sequence $s_1 \cdots s_n$ of "cards," face up, where each card s_i has a value v_i . Then two players take turns picking a card from the sequence, but can only pick the first or the last card of the (remaining) sequence. The goal is to collect cards of largest total value. (For example, you can think of the cards as bills of different denominations.) Assume n is even.

(a) Show a sequence of cards such that it is not optimal for the first player to start by picking up the available card of larger value. That is, the natural greedy strategy is suboptimal.(P1 and P2 are Player 1 and Player 2)

A. Consider the cards s1 = 1, s2 = 2, s3 = 7, s4 = 3. As this is a greedy approach, P1 chooses 3 which leaves P2 with [1,2,7]. P2 can choose between 1 and 7, but as we consider this greedy we go with 7 which leaves P1 with [1,2]. Even if he chooses 2, the total becomes 5 which is less than of P2 (8). So in such cases, greedy is suboptimal.

If P1 chose 1 instead of 3, then he would have won because whatever P2 takes, he can get the highest valued card.

(b) Give an O(n^2) algorithm to compute an optimal strategy for the first player. Given the initial sequence, your algorithm should precompute in O(n^2) time some information, and then the first player should be able to make each move optimally in O(1) time by looking up the precomputed information.

A. To compute an optimal strategy, we can write an algorithm where the basic idea would be Player 1 trying to increase his value whereas player 2 tries to minimize the value of player 1. Considering there are n cards, we can create a (n x n) array A such that where A(i,j) (where 1$\leq i \leq j \leq n$) can be the max sum of cards when 2 players are playing optimally and we fill A[i][j].

```
MaximumValue(V[n]):                     #where V[n] =(v_1,v_2.....v_n)
        for i in (1,n):
                A[i,i] == V[i]
        for len in (2,n):
                for i in (1, n - len + 1):
                        j =i + len -1
                        if((len % 2) == 0):
                                A[i,j] = max(V[i] + A[ i+1, j] , V[j] + A[i , j-1])
                        else:
                                A[i,j] = min(A[ i+1, j] ,A[i , j-1])
```

From the above algorithm, the conditional statements play the main case as if the length of the array is even, P1 plays to maximize his values. The other case player 2 tries to minimize the values. The first step can be computed from the matrix as if A[0,n-1] is greater than A[1,n-2] P1 chooses the first card or vice versa.

4. Do DPV problem 5.22 (or 5.23) about updating an MST after changing the weight of one edge. The problem has four parts; two of the parts are very easy. Note that "linear time" means O(|V|+|E|).

You are given a graph G = (V, E) with positive edge weights, and a minimum spanning tree T = (V, E') with respect to these weights; you may assume G and T are given as adjacency lists. Now suppose the weight of a particular edge e ∈ E is modified from w(e) to a new value $\widehat{w}$(e). You wish to quickly update the minimum spanning tree T to reflect this change, without recomputing the entire tree from scratch. There are four cases. In each case give a linear-time algorithm for updating the tree.

(a) e ∉ E' and $\widehat{w}$(e) > w(e).

Since e is not originally in the MST E',that implies it is a higher weight edge that was excluded from the MST initially. Now if w'(e) increases further, it is still excluded from the MST due to its high weight.
So the MST E' remains unaffected and valid.

Therefore, T' = T. This takes O(1).

(b) e ∉ E' and $\widehat{w}$(e) < w(e).

We check if adding e creates a cycle with T. If no cycle is formed (which is a invalid case) , add e to T. Else, add e and find maximum weight edge e' in the cycle and remove e' from T. This takes O(|V|).

(c) e ∈ E' and $\widehat{w}$(e) < w(e)

Same as Case a. T' = T. This takes O(1).

(d) e ∈ E' and $\widehat{w}$(e) > w(e)

Removing e divides T into 2 subtrees. Find the lightest edge in G and add it to the T'. This takes O(|V| + |E|).

5. You are given a simple graph G where each edge is colored either red or blue, and an integer k. Describe an efficient algorithm that either finds a spanning tree T in G with exactly k red edges, or else it announces that no such tree exists. Give a running time analysis for your algorithm. Time O(|V|2) is possible

A. Case 1 : Consider weight(red) = 0 and weight(blue) =1. We can get a tree T_{max} where there are most number of red edges K_{max} by using MST (As per hw2 discussions). In this case if K_{max} > K, we can say there is no solution.
Case 2 : Consider the opposite case, weight(red) = 1 and weight(blue) =0. We can get a tree T_{min} where there are least number of red edges K_{min} by using MST. In this case if K_{min} > K, we can say there is no solution.
Case 3 : red(T1) < k < red(T2)
Considering spanning trees as bases, we have Given any edge x in T1-T2, we can find an edge y in T2-T1, so T' = T1-{x}+{y} is still a spanning tree. We can implement this in our case while T has <k red edges, we can pick an edge from T_{max} - T and add it to T. Now we can find a cycle in T+ e, remove an edge e' from this where it is not present in T_{max} and return T + e - e'. This loop goes on until number of red edges in T = k.
Time complexity would be O$|V|^2$ as we are implementing Prims or Krushkal's algorithm for first 2 cases, where as time complexity for calculating third case would be near to O|V|.