DP on Trees and DAGs

Basic definitions

-> A tree is a connected acyclic undirected graph. If you root it (choose one node as root), every node has parent / children structure.

-> A DAG is a directed acyclic graph: edges have directions and there is no way to follow edges and get back to the the same node by following directions (no directed cycles).

-> Dynamic programming (DP) is a technique where you break a big problem into subproblems, solve each subproblem, and combine results. If subproblems overlap or have optimal substructure, DP helps reuse results and avoid recomputation.

Why trees / DAGs are good for DP

Here is the intuitive reason:

-> In a rooted tree, each node's result often depends on its children (subtrees). Once you solve for children (subproblems), you can compute for the node. There is a clear hierarchy: parent depends on children, not the other way around.

-> In a DAG, edges point in a direction and you can topologically order the nodes (an order such that all edges go from earlier to later). Then you can compute DP values in that order (so when you process a node, all its prerequisites are already computed).

Because of these properties, you avoid cycles (no infinite dependences), and the subproblems are well defined.

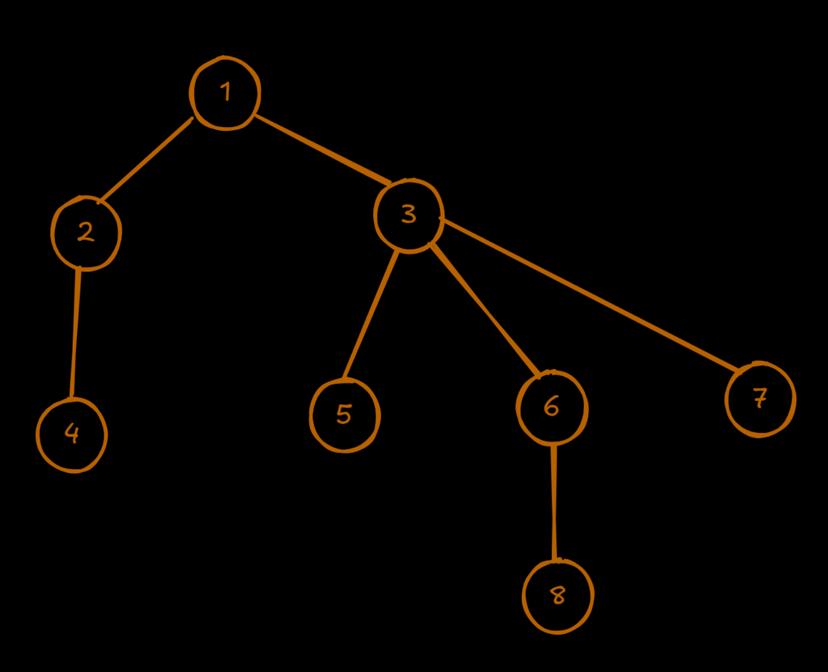
Example:

We have a tree with N nodes.
Each node i has some coins C[i].

We must choose some nodes such that:

- No two connected (adjacent) nodes are chosen at the same time.
- The sum of coins from the chosen nodes is maximum.

We need to find that maximum sum.



How this is similar to array DP

Remember the classic array DP problem:

Given an array A, choose elements so that no two adjacent elements are taken, and sum is maximum.

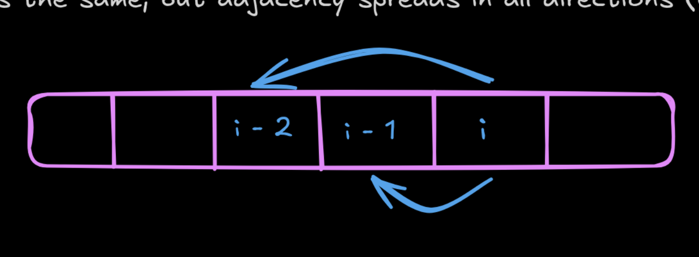
We do something like:

```
dp[i] = max(dp[i-1], dp[i-2] + A[i])
```

- Either don't take A[i], or
- Take A[i] and skip its neighbor.

-> But in arrays, adjacency = previous or next index.
-> In trees, adjacency = directly connected by an edge.

So the idea is the same, but adjacency spreads in all directions (because a node can have many neighbors).



What changes for trees

In arrays → linear relationship.
In trees → each node can have many children.

So instead of doing:

```
dp[i] depends on dp[i-1], dp[i-2]
```

we now do:

```
dp[v] depends on all its children
```

That's the only big change!

Rooting the tree

To make "children" and "subtree" meaningful, we choose a root — say node 1.

That way:
- Each node has a parent.
- Each node has children (nodes below it).

Now we can talk about:
"the subtree of node v" — meaning v and all nodes below it.

DP states and transitions

Define two DP states for node v:

1. dp_take[v] = maximum sum from the subtree rooted at v if you take the node v.
2. dp_not_take[v] = maximum sum from the subtree rooted at v if you do not take node v.

Then:

If you take v: you get the weight of v, plus for each child u
you must use the "not take" state for the child because you can't take the child if you took its parent:

$$\text{dp\_take}[v] = \text{weight}[v] + \sum_{u \in \text{children}(v)} \text{dp\_not\_take}[u]$$

If you don't take v: for each child you can choose either to take or not take (whichever is better):

$$\text{dp\_not\_take}[v] = \sum_{u \in \text{children}(v)} \max\left(\text{dp\_take}[u], \text{dp\_not\_take}[u]\right)$$

Then the answer for the subtree is max(dp_take[root], dp_not_take[root]).

Code

```
void dfs(int v, int parent) {
    dp_take[v] = weight[v];
    dp_not_take[v] = 0;

    for (child u : children of v) {
        if (u == parent) continue;

        dfs(u, v);

        dp_take[v] += dp_not_take[u];
        dp_not_take[v] += max(dp_take[u], dp_not_take[u]);
    }
}
```

Time Complexity

Each edge and node is visited once → O(N).

> Tree DP is just normal DP — but instead of looking at previous elements, you look at children.
>
> Each node makes a decision (include or not), and you merge results from its children.

Extending to DAGs

Now let's see how to adapt to a DAG.

- In DAG, nodes have incoming edges and outgoing edges. You can order nodes in a topological order, such that if there is an edge u→v, then u appears before v.
- You define a DP value for each node depending on its incoming edges (or outgoing), depending on the problem.
- You process nodes in topological order so that when you compute the DP of a node, all the predecessors (or children, depending on direction) are already computed.

Tree DP = dependencies spread down a tree.

DAG DP = dependencies spread along directed edges, but possibly branching in both directions (multiple parents and children).

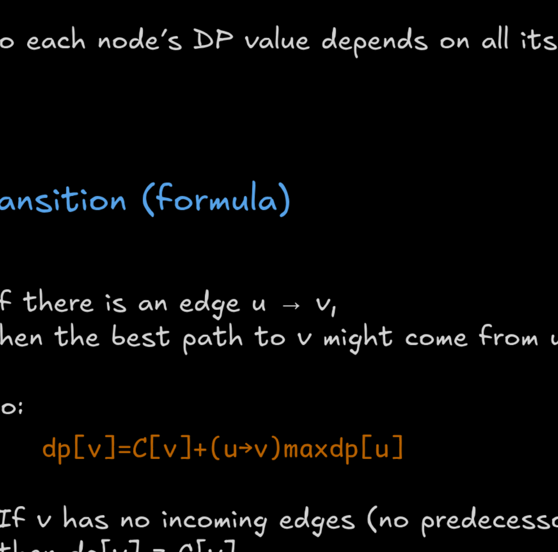
Example

Given a Directed Acyclic Graph (DAG) with N nodes.
Each node i has a value C[i].
Find the maximum sum of values along any path in the DAG.

That means we want the path with the largest total value, following edge directions.

A DAG is a directed graph with no cycles.
- You can move only in the direction of arrows.
- You can't come back (no cycles).

We need to find the longest path sum (not by edge count, but by coin sum).



Define DP state

Just like we did dp[v] for subtree in trees,
we'll define:

```
dp[v] = maximum sum of values along any path ending at node v
```

So each node's DP value depends on all its predecessors (nodes that have an edge leading into it).

Transition (formula)

If there is an edge u → v,
then the best path to v might come from u.

So:

```
dp[v]=C[v]+(u→v)maxdp[u]
```

If v has no incoming edges (no predecessors),
then dp[v] = C[v].

Since the graph has no cycles,
we can safely process nodes in topological order.

Topological order = an ordering of nodes such that all edges go from earlier → later.

That ensures when you compute dp[v], all its predecessors u already have their dp values ready.

Think like this

- In Tree DP, each node waits for all its children to be solved (DFS order).
- In DAG DP, each node waits for all its predecessors to be solved (topo order).