DSU on Tree

Ethen Yuen {ethening}

2026-06-27

Table of Contents

Table of Contents

What Kind of Problems?

Many tree tasks ask for an answer for **every subtree**.

- For every vertex u, look at all vertices inside the subtree of u.
- Compute some statistic: color frequency, depth frequency, maximum, count of active vertices, etc.
- No online updates to the tree or colors during the query process.

What Kind of Problems?

Many tree tasks ask for an answer for **every subtree**.

- For every vertex u, look at all vertices inside the subtree of u.
- Compute some statistic: color frequency, depth frequency, maximum, count of active vertices, etc.
- No online updates to the tree or colors during the query process.

The difficulty is not evaluating one subtree. The difficulty is doing this for **all** subtrees without recomputing everything.

Introductory Problem

CF600E - Lomsat gelral

Given a rooted tree with N vertices. Each vertex u has color c_u.
For every vertex u, find the sum of all colors that appear most frequently in the subtree of u.
Constraints: $N \leq 10^5$.

This is the classical DSU-on-tree introductory problem.

Misleading Name

DSU on Tree is a misleading name. It is popularized by the famous Codeforces tutorial about this.

- In fact, it is more related to the same heavy / light edge observation used in Heavy-Light Decomposition.
- However, since this name is well known in the community, we will keep using this name.

Approach 1: One Subtree by DFS

First, forget about all vertices. Suppose we only need the answer for one fixed subtree u.

We can DFS over subtree u and add one vertex at a time into a maintained state:

1. `cnt[color]`: how many times each color appears.
2. `sumByFreq[f]`: sum of colors that currently appear exactly f times.
3. `mxFreq`: current maximum frequency.

Approach 1: One Subtree by DFS

First, forget about all vertices. Suppose we only need the answer for one fixed subtree u.

We can DFS over subtree u and add one vertex at a time into a maintained state:

1. `cnt[color]`: how many times each color appears.
2. `sumByFreq[f]`: sum of colors that currently appear exactly f times.
3. `mxFreq`: current maximum frequency.

Then the answer is simply:

$$sumByFreq[mxFreq]$$

Adding One Color

Suppose we add one vertex with color c.

Let $old = cnt[c]$.

1. Remove c from bucket old: `sumByFreq[old] -= c`.
2. Increase its frequency: `cnt[c]++`.
3. Add c to bucket $old + 1$: `sumByFreq[old+1] += c`.
4. Update `mxFreq` if $old + 1$ is larger.

Adding Example: After Adding 2

Add colors in this order:

$$2,\ 1,\ 2$$

`cnt[color]`

color	1	2	3
cnt	0	1	0

We have seen color 2 once.

Current answer: $sumByFreq[1] = 2$.

`sumByFreq[f]`

freq	0	1	2
sum	1	2	0

Color 2 is in the frequency-1 bucket.

Adding Example: After Adding 1

Add colors in this order:

$$2,\ 1,\ 2$$

`cnt[color]`

color	1	2	3
cnt	1	1	0

We have seen colors 1 and 2 once.

`sumByFreq[f]`

freq	0	1	2
sum	0	3	0

Frequency-1 bucket now contains colors 1 and 2:

$$1 + 2 = 3.$$

Current answer: $sumByFreq[1] = 3$.

Adding Example: After Adding 2 Again

Add colors in this order:

$$2,\ 1,\ 2$$

`cnt[color]`

color	1	2	3
cnt	1	2	0

Color 2 changes from frequency 1 to 2.

`sumByFreq[f]`

freq	0	1	2
sum	0	1	2

Move color 2 between buckets:

$$sumByFreq[1] \mathrel{-}= 2,$$
$$sumByFreq[2] \mathrel{+}= 2.$$

Now $mxFreq = 2$, so the answer is $sumByFreq[2] = 2$.

Removing One Color

Removing color c is the reverse operation.

Let $old = cnt[c]$.

1. Remove c from bucket old: `sumByFreq[old] -= c`.
2. Decrease its frequency: `cnt[c]--`.
3. Add c to bucket $old - 1$: `sumByFreq[old - 1] += c`.
4. While `sumByFreq[mxFreq] == 0`, decrease `mxFreq`.

Step 4 will run at most one step because max frequency only change in +1/-1 each step. This gives amortized $O(1)$ add/remove.

Why Naive Recomputation Fails

The single-subtree DFS is fine. The problem is repeating it for every vertex.

In a chain:

$$|subtree(1)| + |subtree(2)| + \cdots + |subtree(N)| = N + (N-1) + \cdots + 1 = O(N^2)$$

Why Naive Recomputation Fails

The single-subtree DFS is fine. The problem is repeating it for every vertex.

In a chain:

$$|subtree(1)| + |subtree(2)| + \cdots + |subtree(N)| = N + (N-1) + \cdots + 1 = O(N^2)$$

We need a way to reuse the frequency table between related subtrees, while still avoiding pollution between sibling subtrees.

Approach 2: Store a Map for Every Subtree

A natural tree DP idea: We reuse subtree info by merging them to calculate the new state.

- Let `mp[u][color]` = frequency of that color inside subtree u.
- To compute `mp[u]`, merge all child maps into it.

Conceptually, the frequency map is:

$$mp[u] = \{color[u]\} + \sum_{v \text{ child of } u} mp[v].$$

Approach 2: Store a Map for Every Subtree

A natural tree DP idea: We reuse subtree info by merging them to calculate the new state.

- Let `mp[u][color]` = frequency of that color inside subtree u.
- To compute `mp[u]`, merge all child maps into it.

Conceptually, the frequency map is:

$$mp[u] = \{color[u]\} + \sum_{v \text{ child of } u} mp[v].$$

To solve this task, we also need to store and update:

- `mxFreq[u]`: maximum frequency in subtree u,
- `sum[u]`: sum of colors whose frequency is `mxFreq[u]`.

Naive Map Merging

The most direct implementation:

- Create an empty map for u.
- Insert color u.
- For every child v, loop through all entries of `mp[v]` and add them into `mp[u]`.
- Whenever a color frequency changes, update `mxFreq[u]` and `sum[u]`.

Naive Map Merging

The most direct implementation:

- Create an empty map for u.
- Insert color u.
- For every child v, loop through all entries of `mp[v]` and add them into `mp[u]`.
- Whenever a color frequency changes, update `mxFreq[u]` and `sum[u]`.

Problem: the same information may be transferred into new map again and again.
In a chain, the map sizes transferred are:

$$1 + 2 + 3 + \cdots + N = O(N^2).$$

Therefore:

- with `std::map`: $O(N^2 \log N)$ worst case,
- with hash maps: $O(N^2)$ expected, still too slow.

Small-to-Large Map Merging

The standard fix: when merging two maps, always move the smaller one into the larger one.

- Among u and all children, keep the largest map as the base map.
- Insert all entries from smaller maps into this largest map.
- Then attach that map to vertex u.

Small-to-Large Map Merging

The standard fix: when merging two maps, always move the smaller one into the larger one.

- Among u and all children, keep the largest map as the base map.
- Insert all entries from smaller maps into this largest map.
- Then attach that map to vertex u.

Why is this fast?

- Whenever an entry is moved, its new container size at least doubles.
- So each entry can be moved only $O(\log N)$ times.

Small-to-Large Complexity

Total number of moved entries:

$$O(N \log N).$$

So the total time depends on the map implementation:

- `std::map`: each update costs $O(\log N)$.

$$O(N \log^2 N)$$

- `std::unordered_map`: each update is expected $O(1)$.

$$O(N \log N) \text{ expected}$$

Small-to-Large Map Merging

Small-to-large is a valid technique. But there are some problems with it:

- It keeps many dynamic containers.
- `std::map` has large node overhead.
- `std::unordered_map` has large bucket overhead and unstable constants.

In this lecture, we are going to talk about another tool. The main idea is still important:

Save the big part. Rebuild or move the small parts.

DSU on Tree keeps this philosophy, but uses **one global data structure** instead of many maps.

Table of Contents

Heavy Child

Before we go into DSU on Tree, let's revise some concepts of Heavy-light decomposition.

For every vertex u:

- The child with largest subtree size is called the **heavy child**.
- Other children are called **light children**.
- The edge to the heavy child is a heavy edge; other child edges are light edges.

Heavy Child

Before we go into DSU on Tree, let's revise some concepts of Heavy-light decomposition.

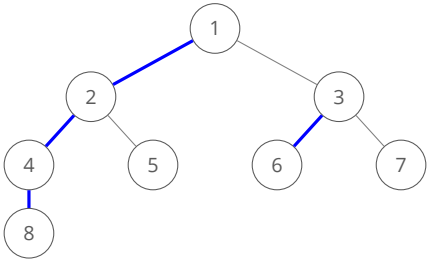
For every vertex u:

- The child with largest subtree size is called the **heavy child**.
- Other children are called **light children**.
- The edge to the heavy child is a heavy edge; other child edges are light edges.

This is the exact same classification used in Heavy-Light Decomposition (HLD).

We do not need the full HLD path-query machinery (Segment trees over chains) today. We only need its crucial size property.

A Small Tree Example



Blue edges are the heavy edges in this example.

Think about it. What happens to the subtree size when we move from a light child to its parent?

Light Edge Bound

Suppose v is a light child of u.

- Since v is not the heavy child, there is some other child of u whose subtree size is at least $sz[v]$.
- Therefore $sz[u] \geq 1 + sz[v] + sz[v] > 2sz[v]$.
- When moving upward through a light edge, the subtree size **at least doubles**.

Light Edge Bound

Suppose v is a light child of u.

- Since v is not the heavy child, there is some other child of u whose subtree size is at least $sz[v]$.
- Therefore $sz[u] \geq 1 + sz[v] + sz[v] > 2sz[v]$.
- When moving upward through a light edge, the subtree size **at least doubles**.

A subtree size can double at most $\log_2 N$ times.

Any path-to-root contains at most $O(\log N)$ light edges.

Why This Matters

If we can arrange our algorithm so that expensive repeated work **only happens after crossing light edges**, then every vertex pays the cost only $O(\log N)$ times.

This is the entire complexity proof of DSU on tree.

So the algorithm should try to:

- **Keep** the heavy child's information.
- **Recompute** the light children's information.
- Make recomputation cheap by using simple global arrays.

Table of Contents

One Global Counter

Imagine we have only one global state:

`cnt[color]`, `sumByFreq[f]`, `mxFreq`

One Global Counter

Imagine we have only one global state:

`cnt[color]`, `sumByFreq[f]`, `mxFreq`

For a single fixed subtree, this is enough.

One Global Counter

Imagine we have only one global state:

`cnt[color]`, `sumByFreq[f]`, `mxFreq`

For a single fixed subtree, this is enough.

But if we process two sibling subtrees one after another:

- Data from the first sibling pollutes the second sibling.
- Subtree A's colors will mix into Subtree B's calculations.
- We must clear the global array at the right time.

Two Different DFS Routines

It is easier to understand the algorithm with two separate routines:

1. `solve(u)`:
   - controls the recursion order,
   - records answers for vertices.
2. `addSubtree(u, delta)`:
   - only scans vertices in subtree u,
   - applies `addVertex(x)` if `delta=+1`,
   - applies `removeVertex(x)` if `delta=-1`.

This keeps the mental model clean: solving and adding/removing are different actions.

DSU on Tree

Inside `solve(u)`:

1. For every light child v:

   `solve(v), addSubtree(v, -1).`

2. Solve the heavy child v_h. Do not remove it: `solve(v_h)`.
3. Add u itself.
4. Add every light child subtree back:

   `addSubtree(v, +1).`

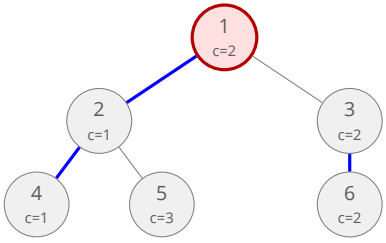
5. Now the global state is exactly subtree u. Record the answer.

The heavy child is not mathematically special for the answer. It is special because leaving the largest part in memory minimizes repeated rebuilding.

Walkthrough

We will trace `solve(1)`.

In every slide, **state** means: the vertices currently counted inside the global arrays.



- Blue edge = heavy edge.
- Gray edge = light edge.
- Heavy child of 1 is 2.
- Light child of 1 is 3.

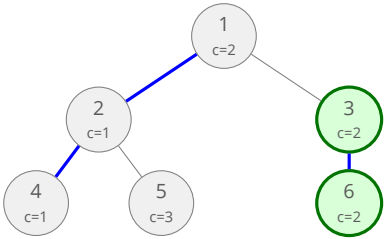
Goal at answer time for vertex 1:

$$state = subtree(1) = \{1, 2, 3, 4, 5, 6\}.$$

Step 1: solve(1) Starts with Light Child 3

The first two calls are:

`solve(3)`, `addSubtree(3, -1)`.



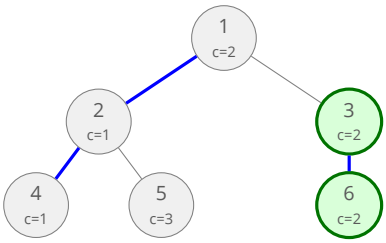
State trace:

$$\emptyset \xrightarrow{\texttt{solve(3)}} \{3, 6\} \xrightarrow{\texttt{addSubtree(3,-1)}} \emptyset$$

Step 1: solve(1) Starts with Light Child 3

The first two calls are:

`solve(3)`, `addSubtree(3, -1)`.



State trace:

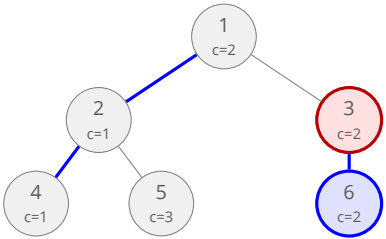
$$\emptyset \xrightarrow{\texttt{solve(3)}} \{3, 6\} \xrightarrow{\texttt{addSubtree(3,-1)}} \emptyset$$

What happened?

- `solve(3)` saved `ans[3]`.
- Then the parent removed subtree 3.
- Subtree 3 should not pollute sibling subtree 2.

Inside solve(3)

Why does `solve(3)` return with state $\{3, 6\}$?



Internal trace:

$$\emptyset \xrightarrow{\texttt{solve(6)}} \{6\} \xrightarrow{\texttt{addVertex(3)}} \{3, 6\}$$

Now the state is exactly subtree 3, so we record:

$$ans[3] = \text{current answer}.$$

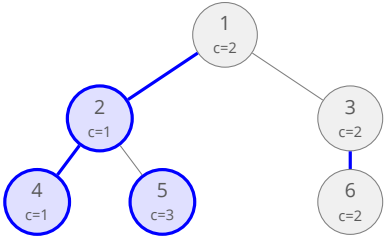
No light child exists under 3, so nothing has to be rebuilt inside this call.

Step 2: solve Heavy Child 2

Back in `solve(1)`, the state is empty again.

Now call:

`solve(2)`.



State after the call:

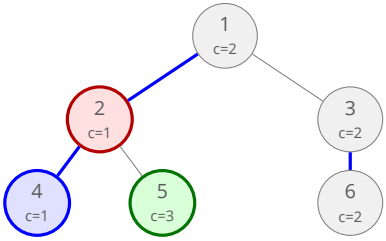
$$state = \{2, 4, 5\}.$$

This time the parent does **not** remove subtree 2.

- 2 is the heavy child of 1.
- Subtree 2 is the large part reused for subtree 1.

Inside solve(2)

The same rule is used recursively.



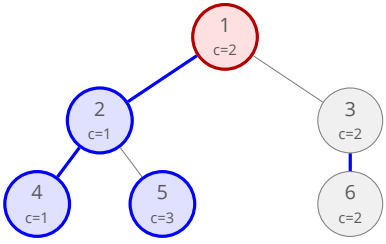
Trace inside `solve(2)`:

$$\emptyset \xrightarrow{\texttt{solve(5)}} \{5\} \xrightarrow{\texttt{remove 5}} \emptyset$$
$$\xrightarrow{\texttt{solve(4)}} \{4\} \xrightarrow{\texttt{add 2}} \{2,4\} \xrightarrow{\texttt{add 5}} \{2,4,5\}.$$

Then we record `ans[2]` because the state is exactly subtree 2.

Step 3: Add Vertex 1

Now `solve(1)` has finished all children.



Before adding vertex 1:

$$state = \{2, 4, 5\}.$$

Operation:

`addVertex(1)`.

After:

$$state = \{1, 2, 4, 5\}.$$

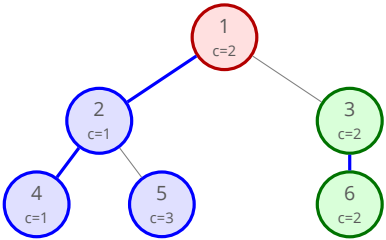
This is not subtree 1 yet. We are still missing the light subtree 3.

Step 4: Rebuild Light Subtree 3

We do not call `solve(3)` again.

We only scan its vertices:

`addSubtree(3,+1)`.



State trace:

$$\{1,2,4,5\} \xrightarrow{\texttt{addVertex(3)}} \{1,2,3,4,5\}$$

$$\{1,2,3,4,5\} \xrightarrow{\texttt{addVertex(6)}} \{1,2,3,4,5,6\}.$$

Now the state is exactly subtree 1, so record:

$$ans[1] = \text{current global answer}.$$

Walkthrough Summary

For vertex 1, the global maintained table changes like this:

$$\emptyset \xrightarrow{\texttt{solve(3)}} \{3,6\} \xrightarrow{\texttt{addSubtree(3,-1)}} \emptyset \xrightarrow{\texttt{solve(2)}} \{2,4,5\}$$

$$\xrightarrow{\texttt{addVertex(1)}} \{1,2,4,5\} \xrightarrow{\texttt{addSubtree(3,+1)}} \{1,2,3,4,5,6\}$$

The key invariant is: **at answer time for u, the global table is exactly subtree u.**

What Was Reused? What Was Rebuilt?

For vertex 1:

part	what happens	why
light subtree 3	solved, removed, then re-added	small enough to re-add
heavy subtree 2	solved and kept	largest child subtree
vertex 1	added last	answer needs whole subtree 1

DSU on tree is exactly this tradeoff:

$$\text{leave one large child} + \text{rebuild all smaller children.}$$

With this algorithm, the total time complexity is $O(N \log N)$.

What Is True at Each Moment?

After solving light children:

- Their answers are already recorded in the `ans` array.
- Their contribution is completely cleared from the global state.

After solving the heavy child:

- The global state contains exactly the heavy child's subtree.

After adding u and all light subtrees:

- The global state contains exactly subtree u.
- We can answer every query attached to u.

Flattening the Tree

To add or remove a whole subtree quickly in code, use **DFS order**.

- During the first DFS, store vertices in an array `ord`.
- Let `tin[u]` be the entry time, and `tout[u]` be the exit time.
- Then subtree u is a contiguous range:

$$[tin[u], tout[u])$$

This turns "add all vertices in subtree u" into a simple $O(sz[u])$ loop over an interval array, which is very cache-friendly.

Why the Complexity Is $O(N \log N)$

Fix a vertex x. When can x be added by a brute-force subtree loop?

- When some ancestor u is adding one of its light subtrees.
- That means the path from u down to x starts with a light edge.

Why the Complexity Is $O(N \log N)$

Fix a vertex x. When can x be added by a brute-force subtree loop?

- When some ancestor u is adding one of its light subtrees.
- That means the path from u down to x starts with a light edge.

Along the path from x to the root, there are only $O(\log N)$ light edges.

Therefore each vertex is brute-force added $O(\log N)$ times.
If one vertex update is $O(1)$, total time is $O(N \log N)$.

What About Clearing?

Clearing a light subtree also loops over vertices.

What About Clearing?

Clearing a light subtree also loops over vertices.

The same analysis works:

- The parent removes a child subtree only when that child is light.
- So every remove loop is also associated with a light edge.

Total add/remove vertex visits:

$$O(N \log N)$$

Space:

$$O(N + \text{range of maintained values})$$ No maps needed!

Table of Contents

CF600E: State Design

For CF600E, our first naive solution with per vertex add / deletion is useful! The global state can be:

- `cnt[c]`: current frequency of color c.
- `sumByFreq[f]`: sum of colors whose frequency is exactly f.
- `mxFreq`: maximum frequency currently seen.

CF600E: State Design

For CF600E, our first naive solution with per vertex add / deletion is useful! The global state can be:

- `cnt[c]`: current frequency of color c.
- `sumByFreq[f]`: sum of colors whose frequency is exactly f.
- `mxFreq`: maximum frequency currently seen.

When adding a vertex of color c:

- move c out of bucket `cnt[c]`,
- increase `cnt[c]`,
- move c into bucket `cnt[c]`,
- update `mxFreq`.

Answer:

$$ans[u] = sumByFreq[mxFreq].$$

A Subtle Point About Deletion

DSU on tree **does not require arbitrary deletion**.

- After `solve(v)` returns for a light child v, the parent deletes the **whole subtree** v.
- After deleting it, the global state should be completely empty.
- So we can decrement `cnt[color]` and update `sumByFreq`, then forcefully set `mxFreq = 0` after the state is empty.

This leads to simpler state maintenance than fully dynamic data structures like Mo's Algorithm on Trees.

Task 2: Palindrome-Anagram Paths

Consider another task:

Palindrome-Anagram Paths

> Given a tree with a lowercase letter on each edge. Count the number of paths whose edge letters can be rearranged into a palindrome.

Task 2: Palindrome-Anagram Paths

Consider another task:

Palindrome-Anagram Paths

> Given a tree with a lowercase letter on each edge. Count the number of paths whose edge letters can be rearranged into a palindrome.

A string can be rearranged into a palindrome iff at most one character appears odd number of times.

$$\text{good} \iff \text{number of odd-frequency letters} \leq 1$$

Represent Odd Counts by a Mask

Since we only care about parity, represent letters by a 26-bit mask.

- bit $i = 1$ means letter i appears odd number of times.
- passing through one edge toggles exactly one bit.
- a path is good iff its mask has 0 or 1 set bit.

Root the tree. Let:

$$pref[x] = \text{xor mask of edge letters on root} \to x.$$

Then for any two vertices x, y:

$$mask(x \to y) = pref[x] \oplus pref[y].$$

How to Count Each Path Once?

Every path has a highest vertex:

$$l = \mathrm{lca}(x, y).$$

We count the pair (x, y) when processing l.

At vertex u, after the heavy child is kept:

- the state already contains vertices from processed child subtrees,
- when scanning a new light child subtree, query against the current state,
- then add that light child subtree into the state.

This ensures two endpoints from the same child subtree are not counted at u.

What Should the State Store?

The global state stores prefix masks of vertices already available for pairing:

$$cntMask[m] = \#\{v \text{ currently in state} : pref[v] = m\}.$$

When we scan a vertex x from a new child subtree, we need vertices y already in the state such that:

$$pref[x] \oplus pref[y]$$

has at most one set bit.

Querying One Endpoint

If the current endpoint is x, the valid masks for $pref[y]$ are:

$$pref[x], \quad pref[x] \oplus 2^0, \quad pref[x] \oplus 2^1, \quad \ldots, \quad pref[x] \oplus 2^{25}.$$

So the number of valid partners already in the state is:

$$cntMask[pref[x]] + \sum_{i=0}^{25} cntMask[pref[x] \oplus (1 \ll i)].$$

Each scanned endpoint costs $O(26)$.
Total complexity stays:

$$O(26N + 26N \log N)$$

because every vertex is scanned through light-subtree rebuilds only $O(\log N)$ times.

Small Example

Suppose we are scanning a vertex x with:

$$pref[x] = 001.$$

The current state has:

mask m	000	001	010	011
`cntMask[m]`	2	3	1	4

If the alphabet is only $\{a, b\}$ for this example, valid partner masks are:

$$001,\ 000,\ 011.$$

new paths through current LCA $= cntMask[001] + cntMask[000] + cntMask[011] = 3 + 2 + 4 = 9.$

DSU-on-Tree Flow for This Task

At vertex u:

- solve light children, then remove them,
- solve heavy child and leave its vertices in the state,
- add vertex u itself to the state,
- for each light child v:
  1. first scan subtree v and query compatible masks,
  2. then add subtree v into the state.

The ordering is important: query before add, otherwise pairs inside the same light subtree would be counted at the wrong LCA.

Skeleton: Subtree Updates

First, isolate the “scan this whole subtree” operation.

```
void addSubtree(int u, int delta) {
  for (int i = tin[u]; i < tout[u]; i++)
    applyVertex(ord[i], delta);
}
```

- `delta = +1`: add every vertex in subtree u.
- `delta = -1`: remove every vertex in subtree u.

Skeleton: solve(u)

Then `solve(u)` only controls the order.

```
void solve(int u, int p) {
  for (int v : adj[u]) if (v != p && v != heavy[u]) {
    solve(v, u);
    addSubtree(v, -1); // state is empty here
  }

  if (heavy[u]) solve(heavy[u], u); // left in the global state
  applyVertex(u, +1);

  for (int v : adj[u]) if (v != p && v != heavy[u])
    addSubtree(v, +1);

  ans[u] = currentState(); // state is subtree u
}
```

Checkpoint

Before moving on, you should be able to explain:

1. Why do we solve light children before the heavy child?
2. Why do we leave the heavy child in the global state?
3. Why is every vertex rebuilt only $O(\log N)$ times?

Think about it. Are you really clear what is happening?

Table of Contents

Data Structures Optimized Tree Queries

In fact, there are many ways to optimize tree queries. Which one should we use?

Technique	Maintains info based on...	Complexity
DSU on Tree	Subtree Size (`siz`)	$O(N \log N)$
Long-chain Decomp	Maximum Depth (`mxdep`)	$O(N)$
Segment Tree Merge	SegTree node structures	$O(N \log N)$
Set Small-to-Large	Irregular objects	$O(N \log^2 N)$

The correct technique depends entirely on the **shape of the state** you are merging. Let's briefly look at the others.

When DSU on Tree is Enough

DSU on Tree is best for offline aggregate statistics (distinct colors, maximums, frequencies).

It relies on two assumptions:

1. Inserting or removing a **single vertex** into the global state is very cheap ($O(1)$ array update).
2. A **flat global state** is enough to answer the queries.

If you need to query something complex *inside* the state (e.g., "how many colors appear between L and R times?"), a flat array cannot answer this in $O(1)$.

How Segment Tree Merge Works

Instead of one global tree, every vertex starts with its own dynamically allocated Segment Tree (often containing just 1 element).

To merge node X from one tree and node Y from another:

- If X is empty, just return Y.
- If Y is empty, just return X.
- If both exist, merge their values (e.g., $X.val += Y.val$), then recursively merge their left and right children.

Because a recursion only happens when **both** trees have data in the exact same range, the total number of merges across the entire tree is strictly bounded by the total number of initial insertions: $O(N \log N)$.

Segment Tree Merge vs DSU on Tree

What if we just use DSU on Tree, but replace the global array with a global Segment Tree?

- Every vertex is re-added $O(\log N)$ times.
- Each addition takes $O(\log N)$ in the Segment Tree.
- Total time: $O(N \log^2 N)$.

Segment Tree Merge vs DSU on Tree

What if we just use DSU on Tree, but replace the global array with a global Segment Tree?

- Every vertex is re-added $O(\log N)$ times.
- Each addition takes $O(\log N)$ in the Segment Tree.
- Total time: $O(N \log^2 N)$.

Segment Tree Merge avoids this extra log factor because it skips empty ranges entirely.

- It is required when you need to perform **Segment Tree operations** (like Binary Search on Segment Tree) on the subtree state, or apply **global lazy tags** to the DP state.

Long-Chain Decomposition

What if the problem is strictly about **depth**, with no colors or arbitrary weights?

Using DSU on Tree's $O(N \log N)$ is actually suboptimal. We can change the definition of the special child.

- Heavy-light decomposition: choose child with largest **subtree size**.
- Long-chain decomposition: choose child with largest **maximum depth**.
- The chosen child is called the **deep child**.

Long-Chain Decomposition

What if the problem is strictly about **depth**, with no colors or arbitrary weights?

Using DSU on Tree's $O(N \log N)$ is actually suboptimal. We can change the definition of the special child.

- Heavy-light decomposition: choose child with largest **subtree size**.
- Long-chain decomposition: choose child with largest **maximum depth**.
- The chosen child is called the **deep child**.

It is used when the DP state is indexed mainly by depth:

$$dp[u][j] = \text{some information about depth } j \text{ inside subtree } u.$$

How Long-Chain DP Works

Suppose we maintain the depth-based DP array for subtree u using a `std::vector`.

Instead of copying arrays to merge them:

1. **Inherit:** Parent u takes the deep child's vector in $O(1)$ time using `std::swap`.
2. **Shift:** Parent u appends its own value (depth 0) to the back of the vector. *(Note: the vector stores depths in reverse order so pushing is $O(1)$).*
3. **Merge:** For each light child v, iterate through v's small vector and add its values to u's vector at the correct depth offsets.

We never copy the longest path! The long chain's array is just passed upwards and modified in-place.

Long-Chain DP Complexity

Why is this so fast?

- The deep child's state is inherited in $O(1)$ time.
- We only pay a loop cost when merging a short (light) child chain.
- Because a short child only merges up to its own depth, the sum of all merging operations is bounded by the tree's total maximum depth.

By avoiding the size constraint and bounding by height, the total time complexity drops to $O(N)$.

Set / Container Small-to-Large Merging

If the information is completely irregular, you might not be able to use a flat array or a Segment Tree.

Examples: maintaining disjoint intervals, strings, complex structs, or when a subtree's state must be dynamically destroyed/cleared.

The standard fallback is **Small-to-Large Merging**:

- Maintain a physical container (e.g., `std::set`) for every subtree.
- To merge, always iterate through the **smaller** container and insert its elements into the **larger** container.
- Use `std::swap` to pass the merged large container to the parent in $O(1)$.
- **Complexity:** An element moves to a new container at most $O(\log N)$ times. Since `std::set` insertion is $O(\log N)$, total time is $O(N \log^2 N)$.

A Useful Question

When seeing a tree statistics or tree DP problem, ask:

1. **Is the state indexed purely by depth?**
   $\Rightarrow$ *Long-Chain Decomposition* ($O(N)$).
2. **Do I need SegTree Binary Search or Lazy Tags on the state?**
   $\Rightarrow$ *Segment Tree Merge* ($O(N \log N)$).
3. **Are the objects highly irregular (e.g., intervals)?**
   $\Rightarrow$ *Set Heuristic Merge* ($O(N \log^2 N)$).
4. **Otherwise (Standard offline frequencies/counts):**
   $\Rightarrow$ **DSU on Tree** ($O(N \log N)$, low memory, very fast!).

Think about it. Does a flat, global array represent everything you need? If yes, stick to DSU on Tree.

Table of Contents

Summary

- **DSU on Tree** keeps the heavy child's global state and rebuilds light subtrees.
- The $O(N \log N)$ proof is just the HLD light-edge bound.
- It is best for static offline subtree statistics with cheap vertex insertion.

Practice Task

- **CF600E - Lomsat gelral**
- **CF570D - Tree Requests**
- **CF208E - Blood Cousins**
- **IOI 2011 - Race**
- **ABC311 Ex - Many Illumination Plans** (Not really DSU on tree but the idea is somewhat related)

References

- **Arpa's Blog on Codeforces:** *DSU on tree* (The original popularizer of this specific $O(N \log N)$ technique in the competitive programming community).
- **OI Wiki:** *DSU on Tree (dsu on tree / 树上启发式合并)* (Simplified Chinese).