AACPP - Session 7

Assignment 3 Solution, Assignment 4 Intro

Michalis Georgoulakis, Fabian Wenz

Assignment 3 Rewind

- We are given n rooms, m directed corridors
- **From room 1, all rooms are reachable**
- Dexter wants to start in some room s, run, and eventually return to s.
- He may once reverse the direction of any directed path before starting
  - Path edges: $z0 \longrightarrow z1 \longrightarrow \cdots \longrightarrow zk$
  - After flipping: $zk \longrightarrow \cdots \longrightarrow z1 \longrightarrow z0$

Goal

- How many rooms are loopie-friendly in original graph?
- For each proposed extra corridor (xi, yi), how many become loopie-friendly if that edge is added?

When is a starting room s loopie-friendly?

Dexter can flip **one** directed path first.

Question becomes:

- Is there some directed cycle through s after flipping one path?

This is the heart of the problem.

Brute checking flips is impossible: graph size $\leq 10^6$, queries$\leq 10^6$.

We need a **structural** insight.

Key Insight: Flip ⟹ Undirected Cycle

Critical Lemma

A room s is loopie-friendly in the directed graph G

iff

s lies on a simple cycle in the undirected graph $G \cup G^{-1}$.

This is the trick that makes the problem solvable.

- "⟹" easy: directed cycle ⟹ undirected cycle
- "⟸" needs argumentation (next slide)

Once proven, we completely drop directionality.

Why the undirected reduction works

Take a DFS tree of the **directed** graph G, rooted at 1 (all reachable).

Let C be an undirected cycle in $G \cup G^{-1}$ containing s.

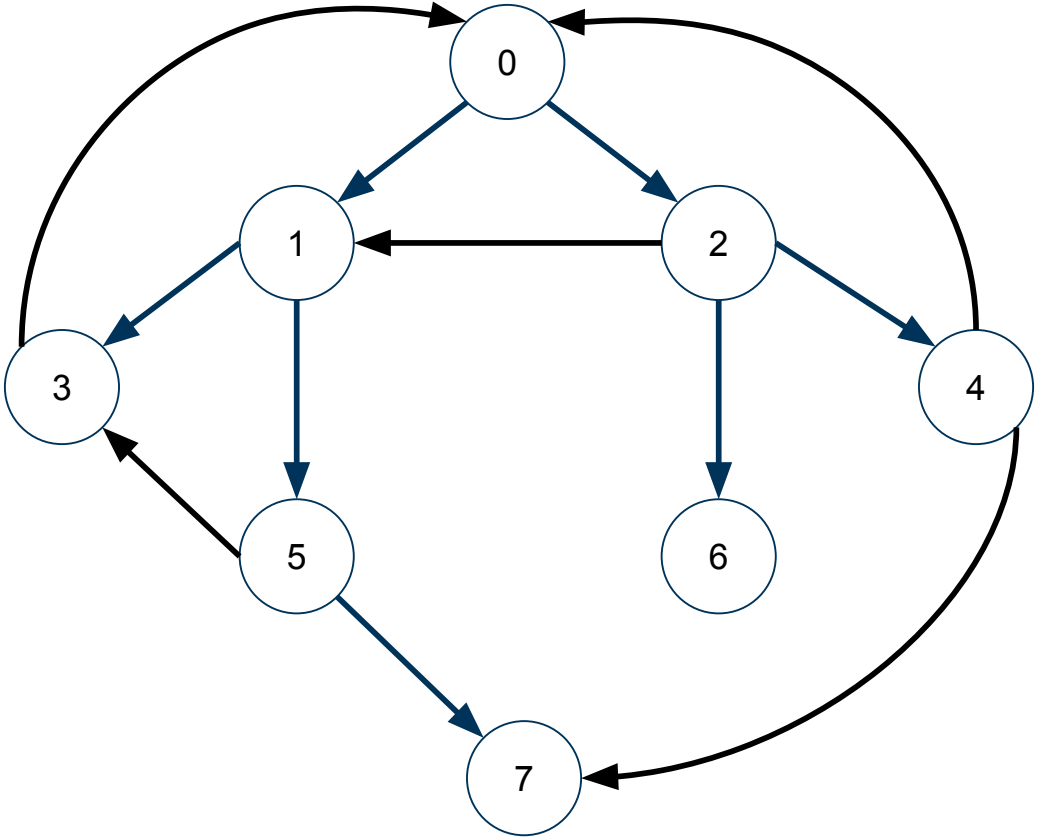
On this cycle:

- All edges but **one** are DFS-tree edges.
- The single non-tree edge is (v,u). Depending on where v,u sit relative to s, we can choose **one** directed path to flip (e.g., $s \longrightarrow v$, or $v \longrightarrow \text{LCA}(v,u)$, etc.)
- After the flip, the whole undirected cycle becomes a **directed** cycle through s.

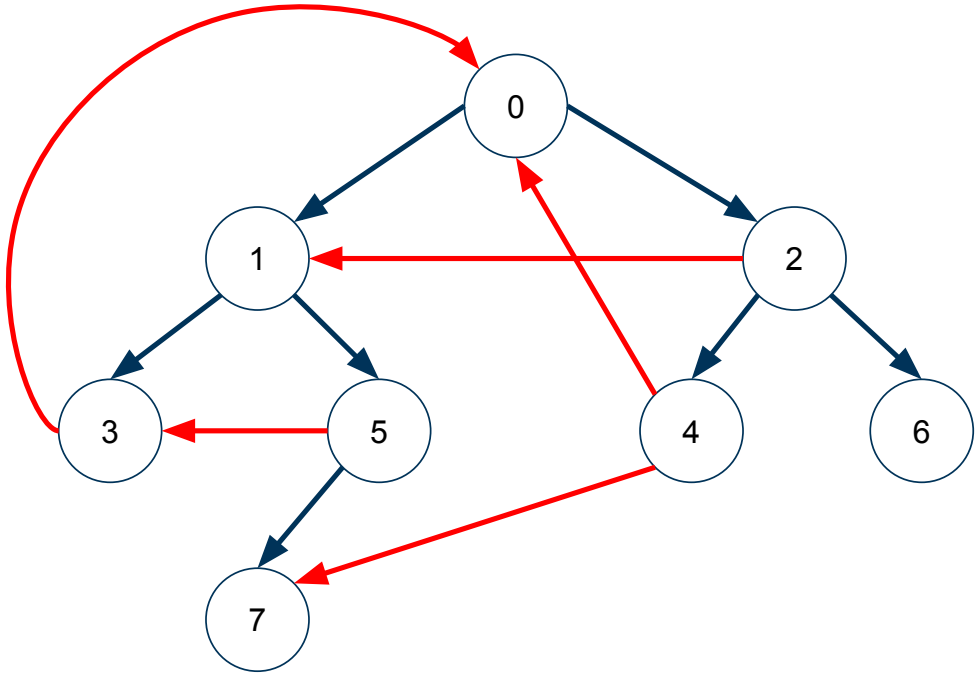
Thus every vertex on an undirected cycle is loopie-friendly.

We now have a purely undirected problem

Given an undirected graph



DFS Tree of Undirected Graph



- **Black:** Edges used to discover all nodes during DFS traversal.
- **Red:** These edges were not used to discover any new node, so they become back/side/cross edges, i.e., the ones that create all the cycles in the graph.

Naive attempt

We need to find all vertices lying on any simple cycle.

Idea:

- For each vertex s, run a DFS/BFS to see if there is a cycle returning to s.
- This finds all cycle vertices correctly, but it costs $O(n \cdot m)$.
  - For $n=m=10^6$: $O(n \cdot m) \approx 10^{12}$ operations $\longrightarrow$ bad.

We must exploit global structure of the graph to find cycle nodes in linear time.

Undirected Structure: Spanning Tree + Back Edges

In any connected graph:

- Pick a spanning tree (DFS or BFS)
- **Tree edges** = edges in the spanning tree
- **Non-tree edges** = edges that create cycles

Every simple cycle is: **Path in the tree + one non-tree edge**

So to mark all vertices on cycles:

- For each non-tree edge (u,v), mark all vertices on the unique tree-path between u and v.

We cannot do this naively: tree paths can be $\boldsymbol{O(n)}$.

We need a fast encoding.

Trick: “Difference on Tree” + Subtree Sum

To mark the tree-path between u and v: Let a=LCA(u,v)

We apply the following (difference array on a tree):

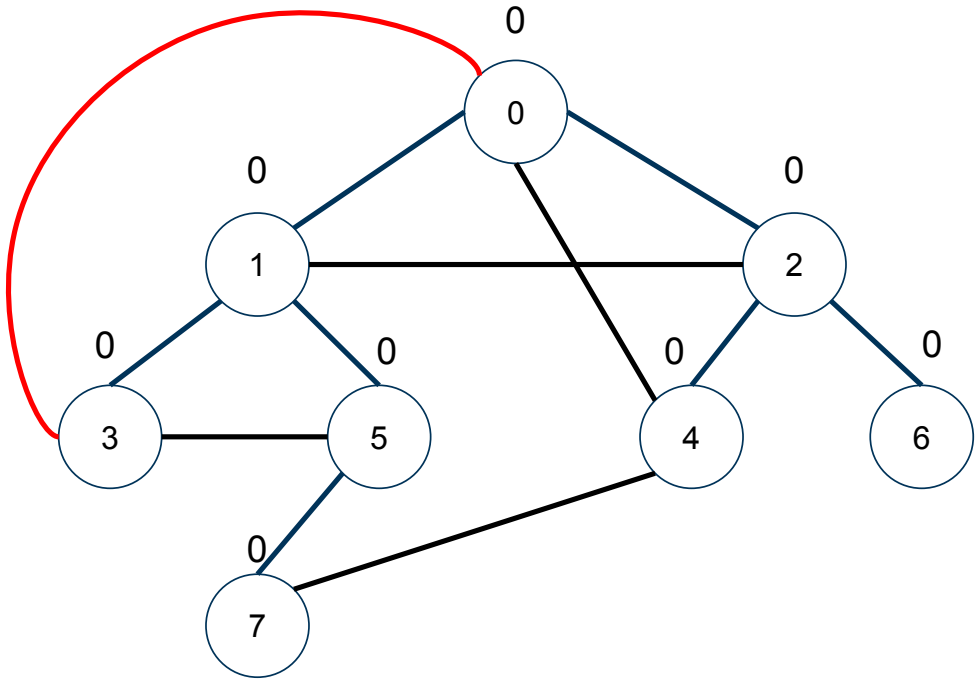
- $d[u]$++
- $d[v]$++
- $d[a]$−−
- $d[\text{parent}(a)]$−−

Then run a **post-order DFS**:

- Each vertex accumulates the sum of its children
- If final d[v] > 0 $\Longrightarrow$ v lies on some cycle

This gives the base answer for q=0.

DFS Tree of Undirected Graph



$d[u]{+}{+}$

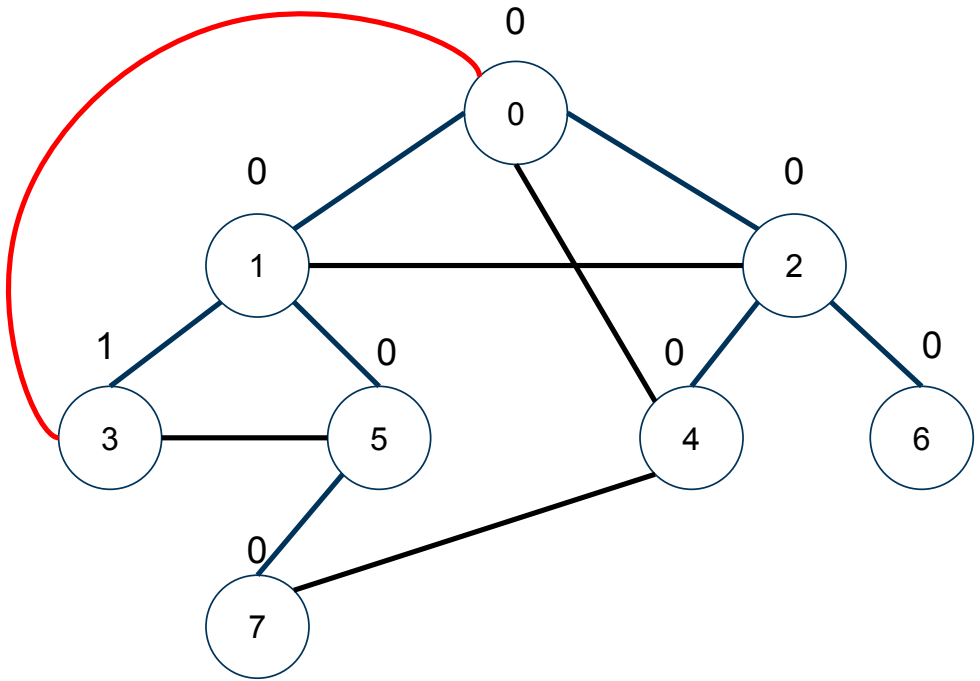
$d[v]{+}{+}$

$d[a]{-}{-}$

$d[\text{parent}(a)]{-}{-}$

Non-tree edges: {**(0,3)**, (0,4), (1,2), (3,5), (4,7)}
Edge: (0,3)
LCA(0,3)=0

DFS Tree of Undirected Graph



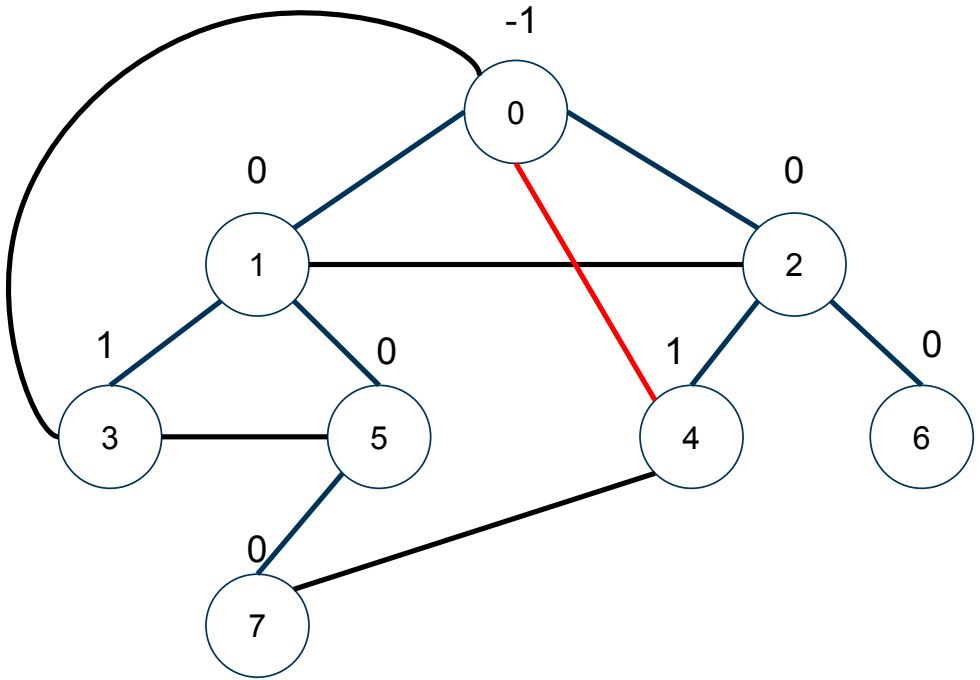
$d[0]{+}{+}$

$d[3]{+}{+}$

$d[0]{-}{-}$

Non-tree edges: {**(0,3)**, (0,4), (1,2), (3,5), (4,7)}
Edge: (0,3)
LCA(0,3)=0

DFS Tree of Undirected Graph



$d[0]{+}{+}$

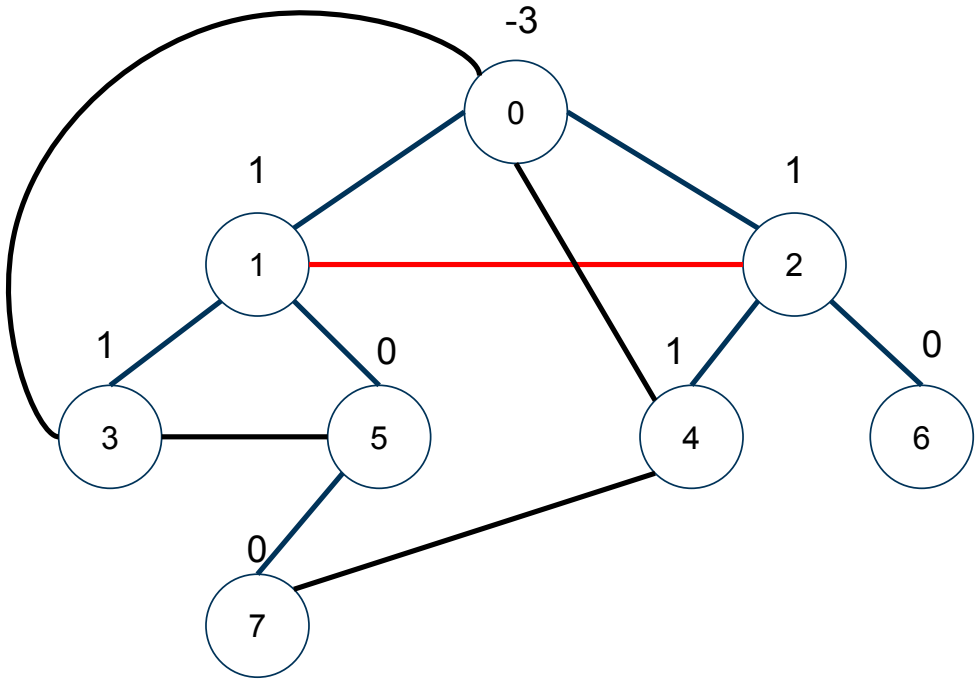
$d[4]{+}{+}$

$d[0]{-}{-}$

$d[0]{-}{-}$

Non-tree edges: {**(0,3)**, **(0,4)**, (1,2), (3,5), (4,7)}
Edge: (0,4)
LCA(0,4)=0

DFS Tree of Undirected Graph



$d[1]{+}{+}$

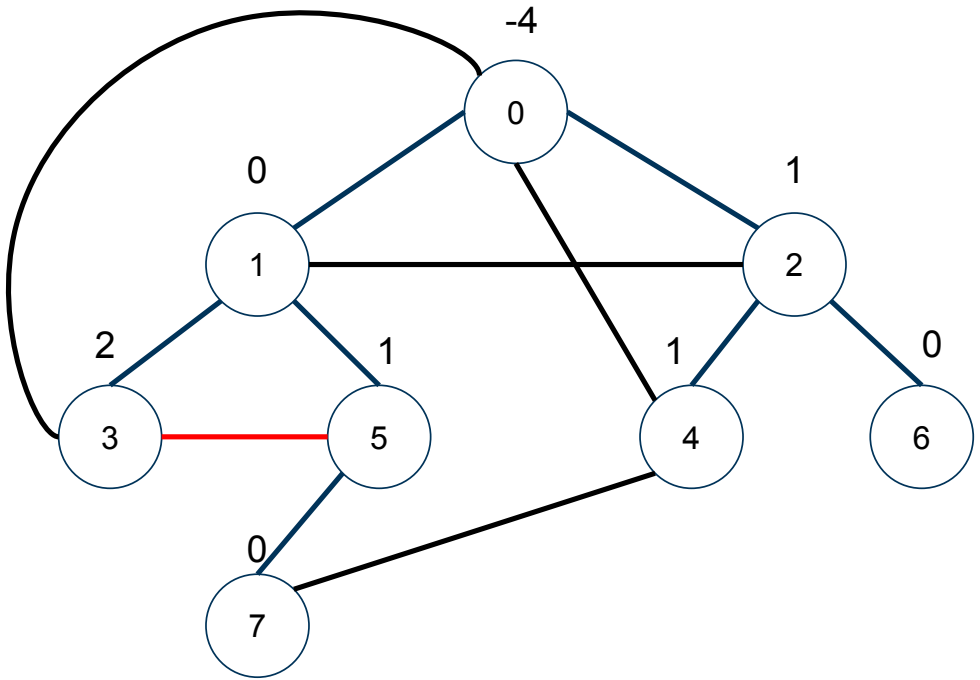
$d[2]{+}{+}$

$d[0]{-}{-}$

$d[0]{-}{-}$

Non-tree edges: {**(0,3)**, **(0,4)**, **(1,2)**, (3,5), (4,7)}
Edge: (1,2)
LCA(1,2)=0

DFS Tree of Undirected Graph



$d[3]$++

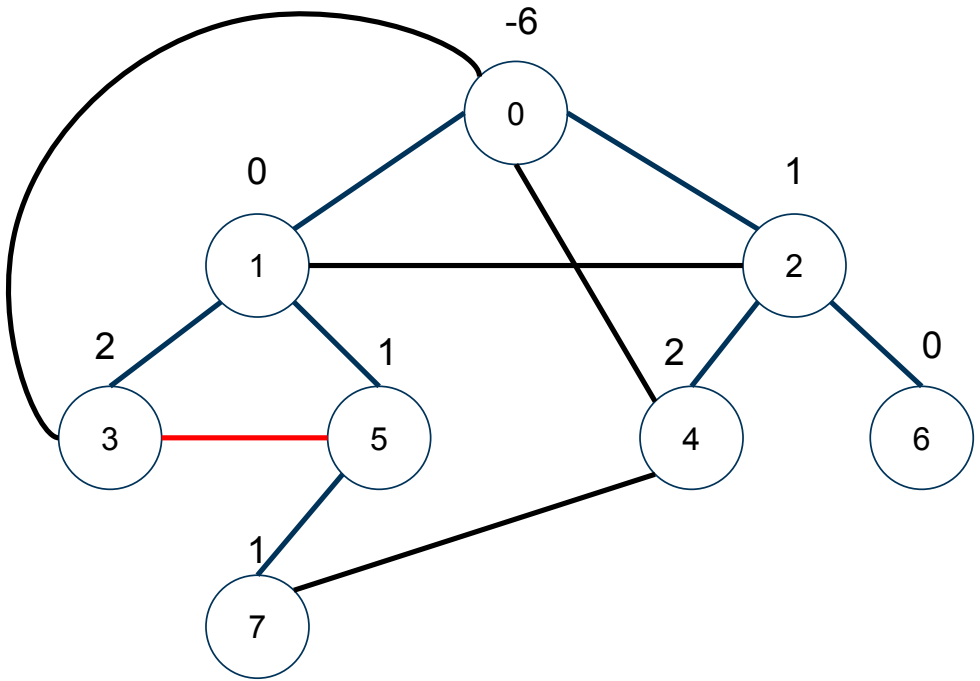
$d[5]$++

$d[1]$--

$d[0]$--

Non-tree edges: {**(0,3)**, **(0,4)**, **(1,2)**, **(3,5)**, (4,7)}
Edge: (3,5)
LCA(3,5)=1

DFS Tree of Undirected Graph



$d[4]$++

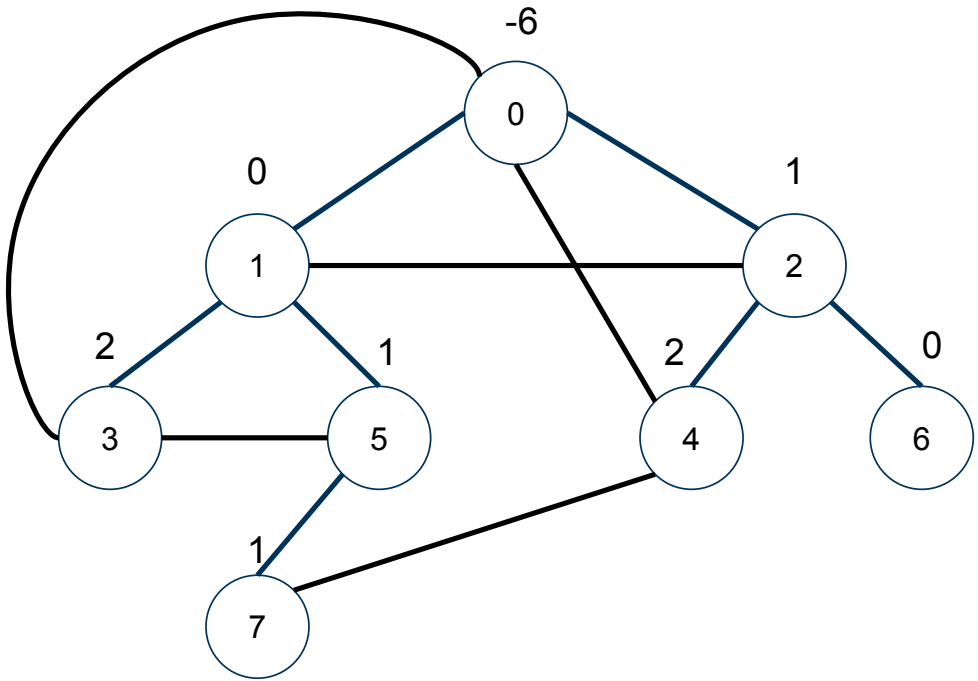
$d[7]$++

$d[0]$--

$d[0]$--

Non-tree edges: {**(0,3)**, **(0,4)**, **(1,2)**, **(3,5)**, **(4,7)**}
Edge: (4,7)
LCA(4,7)=0

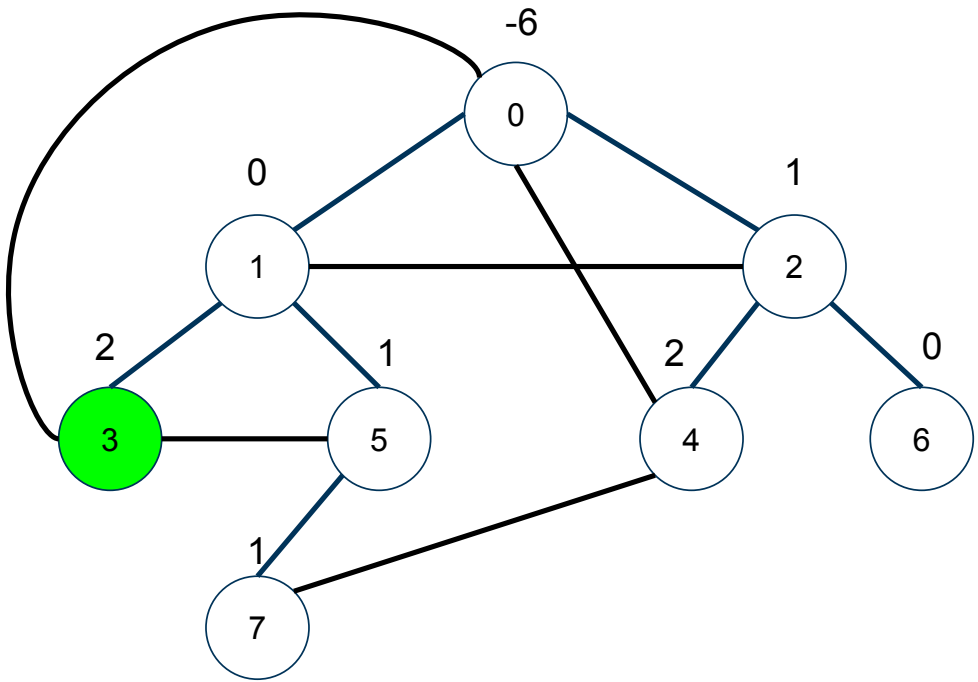
Post-Order Traversal



Each vertex accumulates the sum of its children

If final d[v] > 0 $\Longrightarrow$ v lies on some cycle

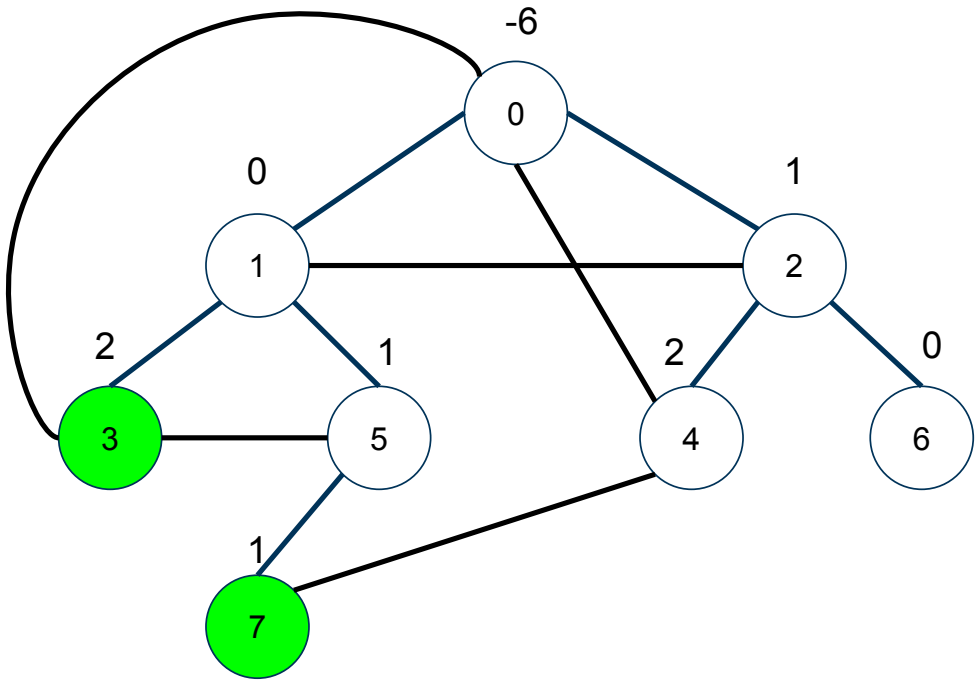
Post-Order Traversal



d[3] has no children

d[3] = 2 > 0

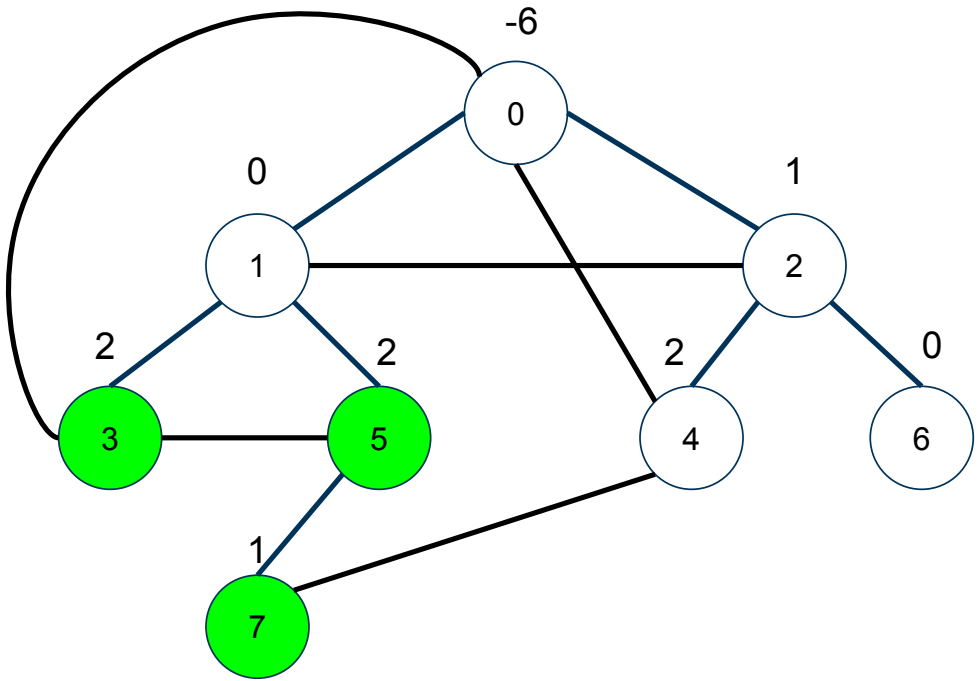
Post-Order Traversal



d[7] has no children

d[7] = 1 > 0

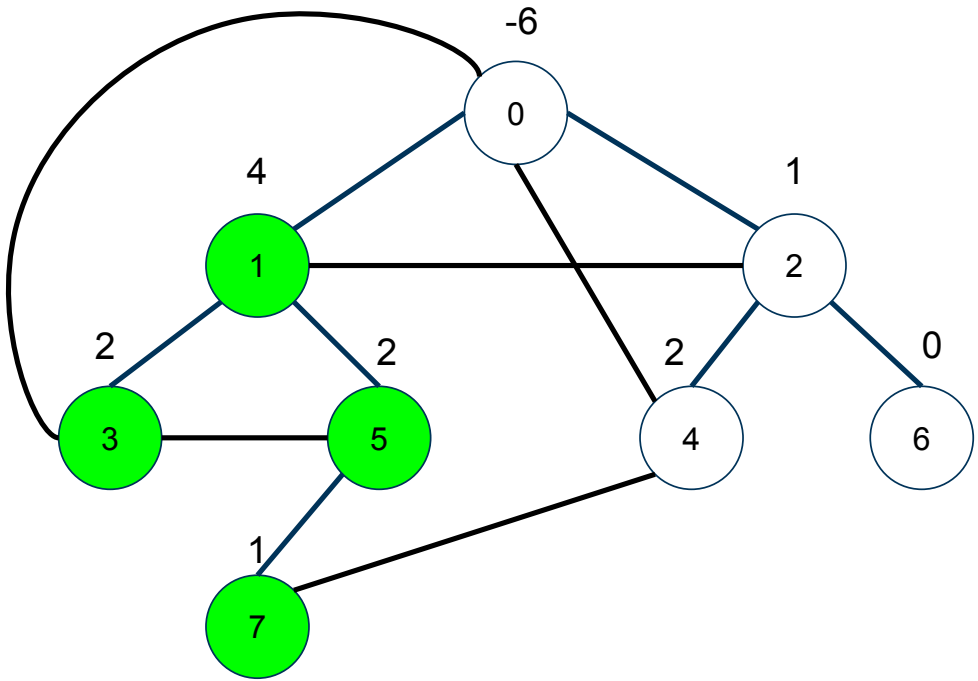
Post-Order Traversal



d[5] += d[7] => d[5]=2

d[5] = 2 > 0

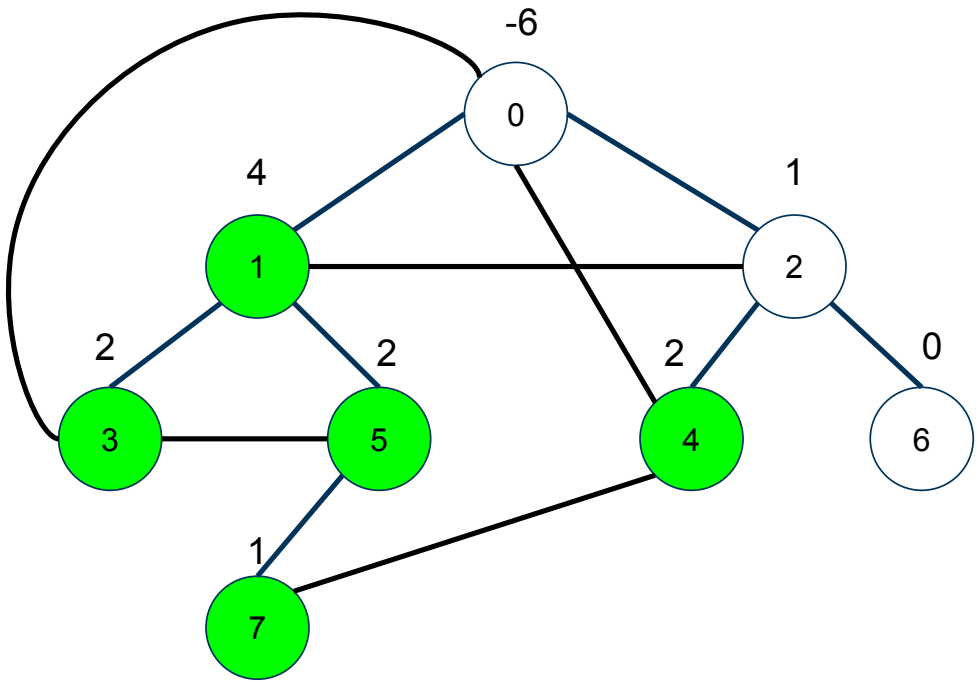
Post-Order Traversal



d[1] += d[3] + d[5] => d[1] = 4

d[1] = 4 > 0

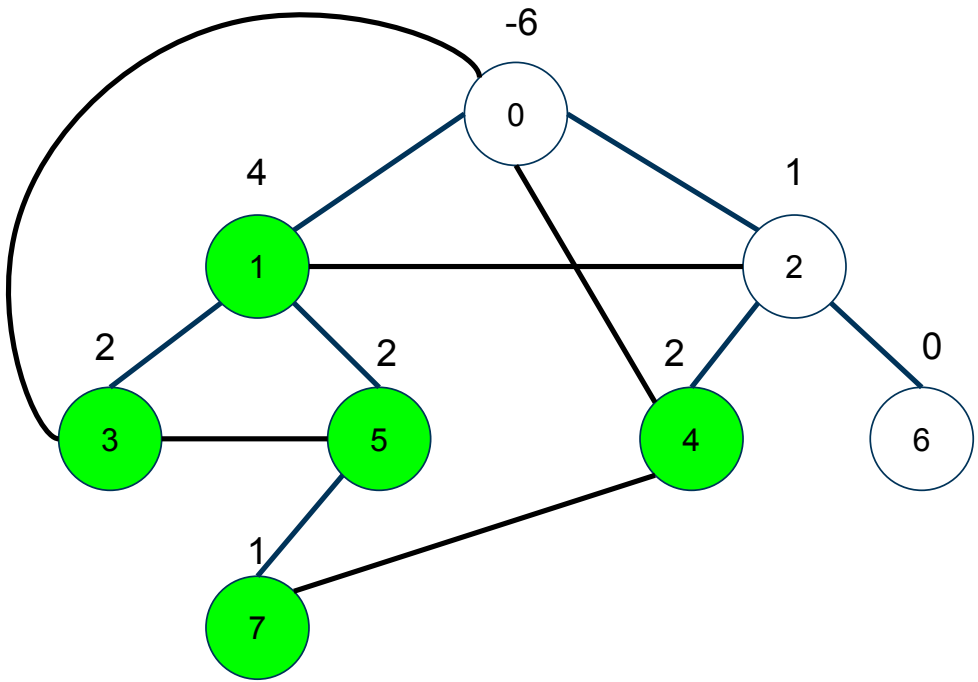
Post-Order Traversal



d[4] has no children

d[4] = 2 > 0

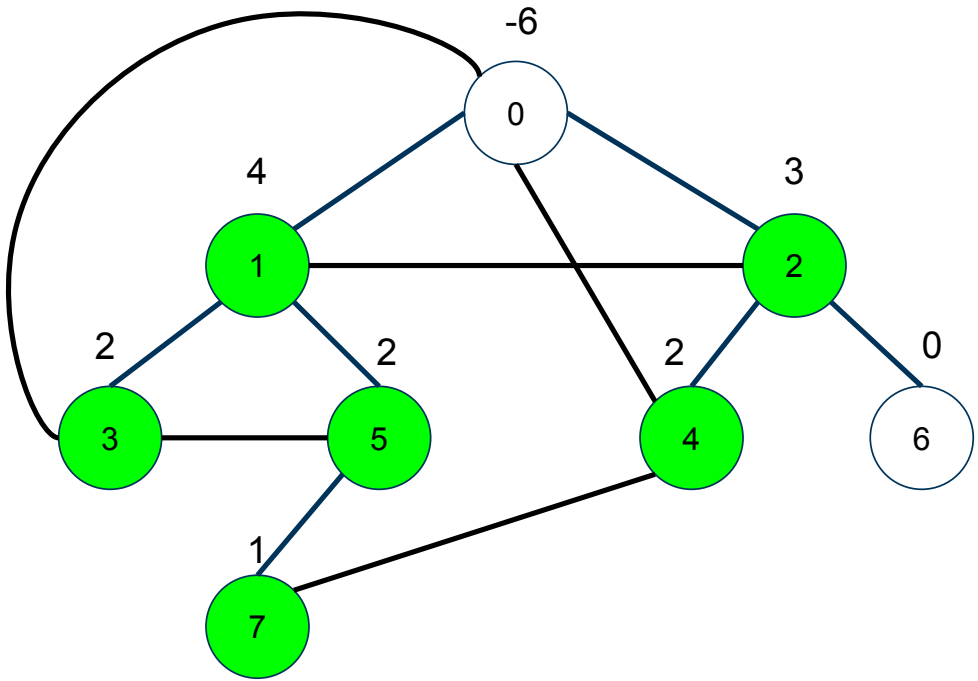
Post-Order Traversal



d[6] has no children

d[6] = 0

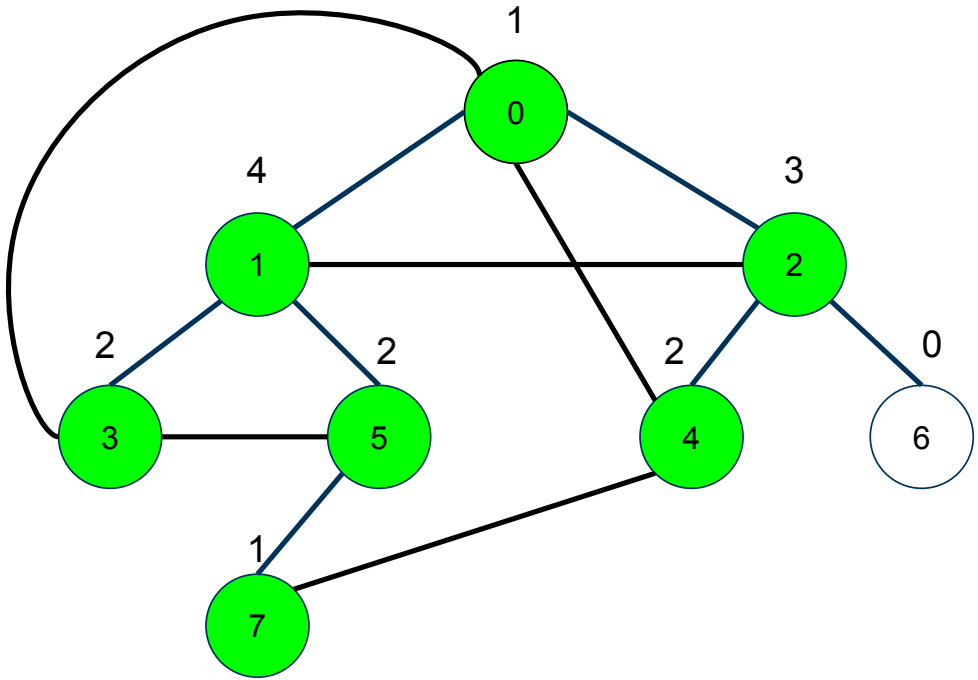
Post-Order Traversal



d[2]+=d[4]+d[6] => d[2] = 3

d[2] = 3 > 0

Post-Order Traversal



d[0]+=d[1]+d[2] => d[0] = 1

d[0] = 1 > 0

PART II - Precompute for Queries

We now need to handle up to 10^6 new edges.

Adding a new undirected edge (x,y) creates one new cycle: **The tree-path between x and y**

We must count how many **previously acyclic** vertices lie on that path.

- Let $T[v]$ = number of acyclic vertices on the path root$\longrightarrow v$.
- Compute it with a simple BFS/DFS on the tree.

For query (x,y): Let a=LCA(x,y)

- Let p=parent(a)
- Number of new cycle vertices: new=$T[x]+T[y]-T[a]-T[p]$
  - This is the usual “two root paths minus shared prefix” pattern on trees.

Total answer: answer$_i$=base_total+new (all in $O(1)$ per query).

Full Solution Analysis

Preprocessing (O((n+m)logn))

1. Convert directed graph to undirected - *O(m)*
2. Build a spanning tree from 1 - *O(n+m)*
3. Build LCA Preprocessing Structure - *O(nlogn)*
    - *Binary Lifting / Euler Tour + Range Minimum Query - O(nlogn)*
4. For every non-tree edge (u,v): apply tree-difference updates - *O(m)*
5. DFS accumulate ⟶ mark all cycle vertices - *O(n)*
6. Build prefix counts $T[v]$ of acyclic nodes on root paths - *O(n)*

Per Query (O(1))

Given edge (x,y):

- Compute a=LCA(x,y)
- Compute new cycle count: $T[x]+T[y]-T[a]-T[\text{parent}(a)]$
- Add to base_total

Alternative: BCCs

We solved the problem using:

- one spanning tree
- non-tree edges
- LCA + prefix sums on the tree

But there's another way to think about it:

Biconnected Components (BCCs)

- In the undirected graph, compute all biconnected components (BCCs).
- A BCC is a maximal subgraph where removing any single vertex does not disconnect it.
- **Fact:** Every simple cycle lies entirely inside one BCC.

Alternative: BCCs

1. **Base case (no queries):**
   - For each BCC:
     - If it has at least one cycle (e.g. $|E\text{bcc}| \geq |V\text{bcc}|$), then every vertex in that BCC is on some cycle.
   - Count all vertices that belong to such "cyclic" BCCs.
2. **Block tree for queries:**
   - BCCs + articulation points form a tree ("block tree").
   - A new edge (x,y) creates a cycle along the unique path between the BCC-nodes containing x and y in this tree.
   - Just like in the main solution:
     - Precompute prefix sums over this tree,
     - Use LCA on the block tree to count how many previously acyclic vertices lie on that path.

Takeaway: BCCs give a more "theoretical" decomposition of the graph, but algorithmically we end up **doing the same kind of work**:

- **"tree + paths + LCA + prefix sums" — just on the block tree instead of the DFS tree.**
- **Computing BCCs and the block tree is also $O(n+m)$.**