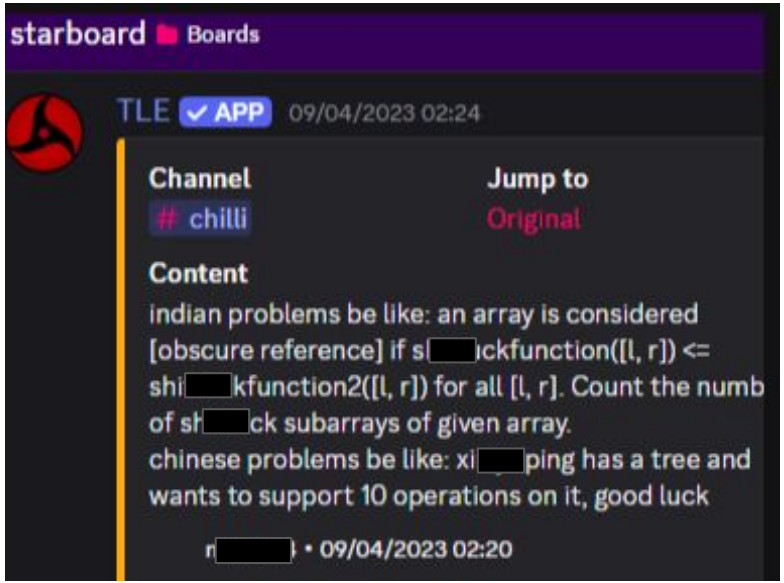
Graph (V)

Nicholas Ko {__jk__}

2026-04-11

Prerequisites and Overview

- Eulerian path and cycle
- Modelling a problem into a setup involving Eulerian paths or cycles
- Centroid Decomposition
- Heavy Light Decomposition
  - aka knowing the first step to deal with ~~Chinese~~ evil data structures tasks
- Some sections of this slides are from last year's Graph (V) lecture, prepared and lectured by `ethening`.



Problems on Graph Decompositions (Challenging)

- If you have attended this lecture in previous years and believe you have fully understood the content, I invite you to try the following problems.
- **Warning**: all of the practice problems below are challenging. It is completely normal if you cannot complete all, or even *any*, of them. (More approachable practice problems are the end of the slides.)
- (Korea TST 2023 Day 1 Problem 3) Taxi
- (NOIP~~lus~~ 2025 Problem 3) 樹的價值
- (JOI Final Stage 2026 Day 4 Problem 2) Festivals in JOI Kingdom 3

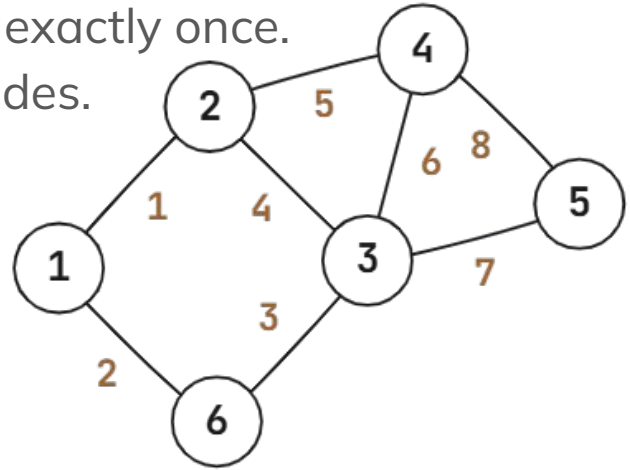
Euler Path

Let's start with some definitions:

Euler Path (also known as Euler Trail)

- A *path* that passes through each **edge** of a graph exactly once.
- Note that we don't care about passing through nodes.
  - It is allowed to revisit the same node multiple times.

For example, the undirected graph on the right has an Euler path: 2 – 1 – 6 – 3 – 2 – 4 – 3 – 5 – 4.

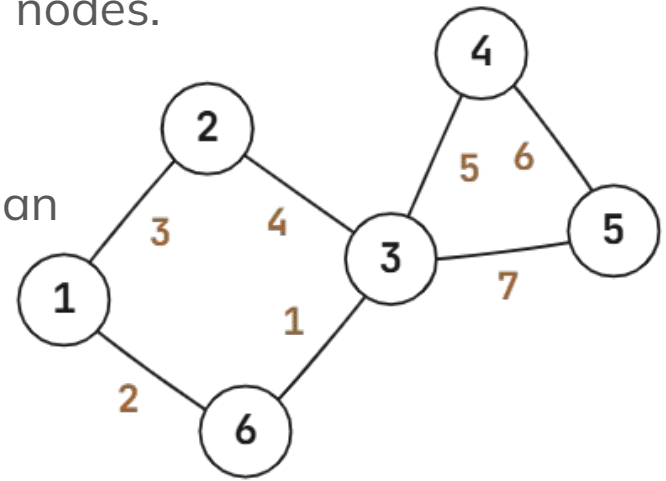


Euler Cycle

Euler Cycle (also known as Euler Circuit)

- A *cycle* that passes through each **edge** of a graph exactly once, **and going back to the same starting node**.
- Note that we don't care about passing through nodes.
  - It is allowed to revisit the same node multiple times.

For example, the undirected graph on the right has an Euler cycle: 3 – 6 – 1 – 2 – 3 – 4 – 5 – 3.



Existence of Euler Path / Cycle in a Graph

Given an arbitrary undirected graph, can we determine efficiently

- Whether an Euler **path** of the graph exists or not?
- Whether an Euler **cycle** of the graph exists or not?
- If the answer to the questions above is “yes”, can we also provide a construction efficiently?
- If the graph is now ***directed*** instead of undirected, can we answer the above questions too?

Euler Cycle for Directed Graph

Claim: If a directed graph has an Euler cycle, then

- every node in the graph must have **indegree = outdegree**, and
- the graph is **connected**.

Why?

- Consider any node u. The number of times the cycle **enters** node u (*indegree* of u) = number of times the cycle **leaves** node u (*outdegree* of u).
- If the graph is not connected, then we cannot construct a cycle that contains all nodes.

Euler Cycle for Directed Graph

In fact, the converse is true!

Claim: Given any directed graph, if

- every node in the graph has **indegree = outdegree**, and
- the graph is **connected**,

then it has an Euler cycle.

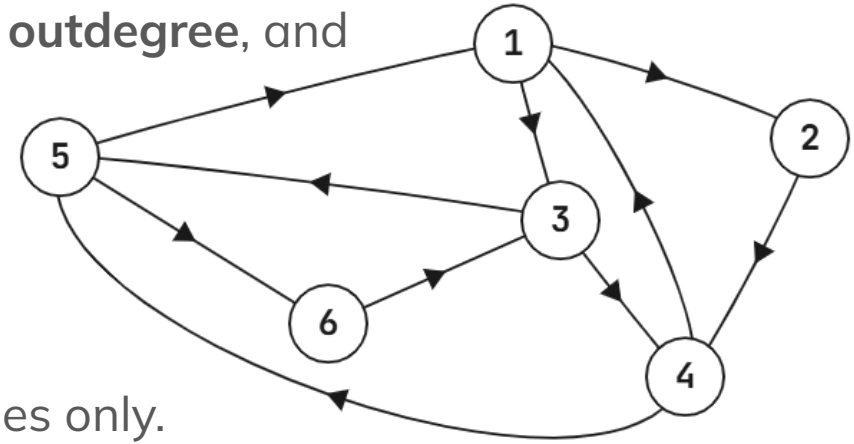
Why?

Euler Cycle for Directed Graph

Claim: Given any directed graph, if

- every node in the graph has **indegree = outdegree**, and
- the graph is **connected**,

then it has an Euler cycle.

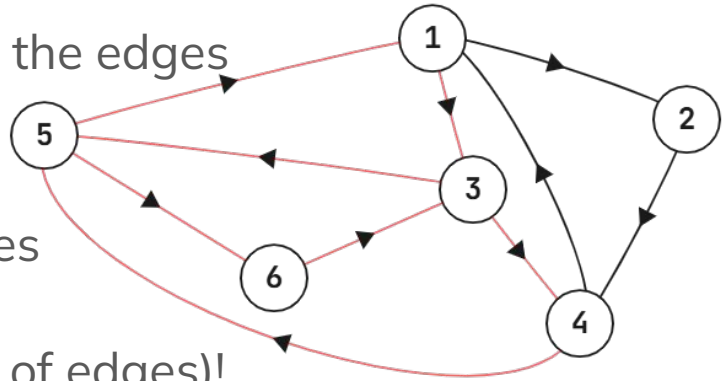


Intuition:

- Suppose you start at an arbitrary node, and you walk arbitrarily via **unused** edges only. When you cannot walk anymore, you must be back at the node where you started at. (Why?)

Euler Cycle for Directed Graph

Suppose we started at node 5, and walked using the edges $5 \to 6 \to 3 \to 5 \to 1 \to 3 \to 4 \to 5$.

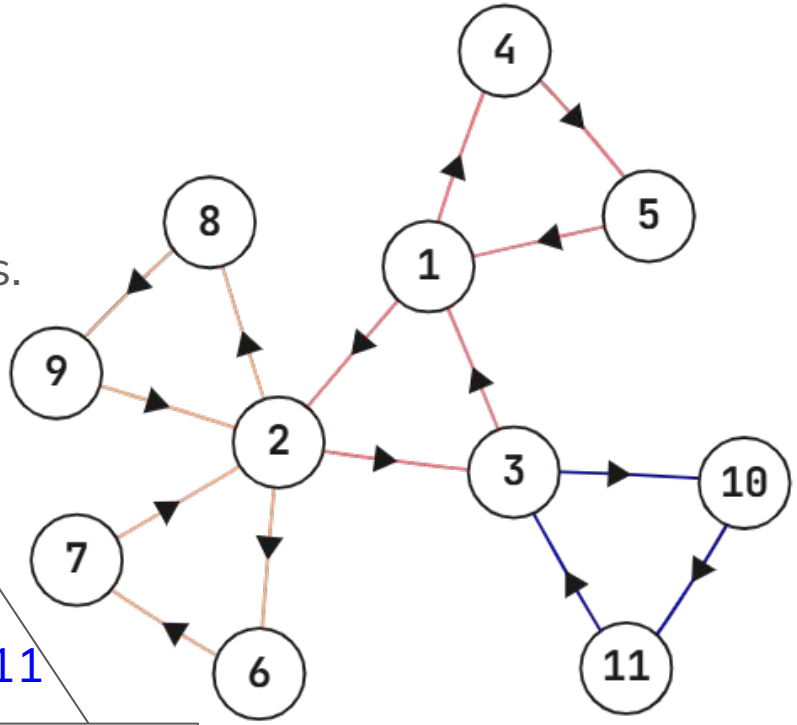


What next? How could we walk through the edges that have yet to be visited?

- We can use recursion / induction (on number of edges)!
- The remaining edges form disjoint graphs which satisfy indegree = outdegree.
- By induction hypothesis, each component has an Euler cycle.
- We “insert” the Euler cycle of a component the first time we meet that component in our arbitrary walk. So, we are done.

Euler Cycle for Directed Graph

- Suppose we started at node 1, and walked 1 → 4 → 5 → 1 → 2 → 3 → 1.
- The remaining edges form disjoint graphs.
  - One component with nodes 2, 6, 7, 8, 9.
  - Another component with nodes 3, 10, 11.
- Then an Euler cycle for our graph is 1 → 4 → 5 → 1 → (Euler cycle for the subgraph with nodes 2, 6, 7, 8, 9 which starts and ends at node 2) → (Euler cycle for the subgraph with nodes 3, 10, 11 which starts and ends at node 3) → 1.



Euler Path for Directed Graph

We now turn our attention to finding an Euler **path** instead of a Euler cycle.

Obviously, if a graph has an Euler cycle, then it has an Euler path.

- Can we use our previous experience to help us?

Claim: If a directed graph has an Euler path, then

- there is *at most one node* with **indegree - outdegree = 1**, and
- there is *at most one node* with **outdegree - indegree = 1**, and
- *all other nodes* have **indegree = outdegree**, and
- the graph is **connected**. (Why?)

Euler Path for Directed Graph

Similar to the case for Euler cycle, the converse turns to be true as well.

Claim: Given any directed graph, if

- there is *at most one node* with **indegree - outdegree = 1**, and
- there is *at most one node* with **outdegree - indegree = 1**, and
- *all other nodes* have **indegree = outdegree**, and
- the graph is **connected**,

then it contains an Euler path.

Euler Path for Directed Graph

Proof:

- If all nodes have indegree = outdegree, we are done.
  - Simply apply the proof for finding Euler cycle (since all cycles are paths).

- Otherwise, denote the node with indegree - outdegree = 1 as u, and the node with outdegree - indegree = 1 as v.

- Consider adding a fictional edge u → v to the graph.
- We can also apply the procedure for finding Euler cycle now.

Euler Cycle for Undirected Graph

We now go back to finding an Euler cycle, this time in an undirected graph.

Claim: If an *un*directed graph has an Euler cycle, then

- every node has **even degree**, and
- the graph is **connected**.

Proof: Suppose the cycle starts at node u.

- For every node except u, every time we enter it, we leave it immediately.
- For node u, every time we leave u, we eventually enter it back again.

Euler Cycle for Undirected Graph

It is unsurprising to us now that the converse holds true:

Claim: Given any *un*directed graph, if

- every node has **even degree**, and
- the graph is **connected**,

then it contains an Euler cycle.

Proof: Exercise.

- Very similar to the proof of finding Euler cycle for a directed graph.

Euler Path for Undirected Graph

Finally, the remaining scenario behaves the same as what we expect.

Claim: Given any *un*directed graph, it contains an Euler path if and only if

- there are *at most two nodes* with ***odd*** degree
  (and so *all other nodes* have **even** degree), and
- the graph is **connected**.

Proof: Exercise.

- You already have all the ingredients required to prove it.

Constructing an Euler Path / Cycle

Suppose we have already checked (using our previous criteria) that an Euler path or cycle exists in a graph. How can we find a construction?

- Recall our trick of adding a fictional edge u → v to find an Euler path.
  - Node u is the unique node with indegree - outdegree = 1.
  - Node v is the unique node with outdegree - indegree = 1.
- So, to find an Euler path, it suffices to know how to find an Euler cycle. This is because we can simply delete the fictional edge back to get an Euler path from v to u.

Constructing an Euler Cycle: Hierholzer's Algorithm

Given a graph which we know (by our previous criteria) that it has an Euler cycle, how can we construct it efficiently? The **Hierholzer's Algorithm** does so:

function Hierholzer(u):

- Find any non-empty cycle starting (and hence ending) at node u. Denote it as C.
  - If no such C exist, return {u} directly.
- Remove the edges in C.
- For every node x in C, replace the x in the list C by Hierholzer(x).
- Return C.

When Hierholzer(u) is called, it returns a list of nodes of an Euler cycle.

Constructing an Euler Cycle: Hierholzer's Algorithm

Assume our graph is directed. (Question: what extra information do we need to store if our graph was *undirected*?)

Hierholzer's algorithm is essentially what we did previously for the proof of showing existence of Euler cycle.

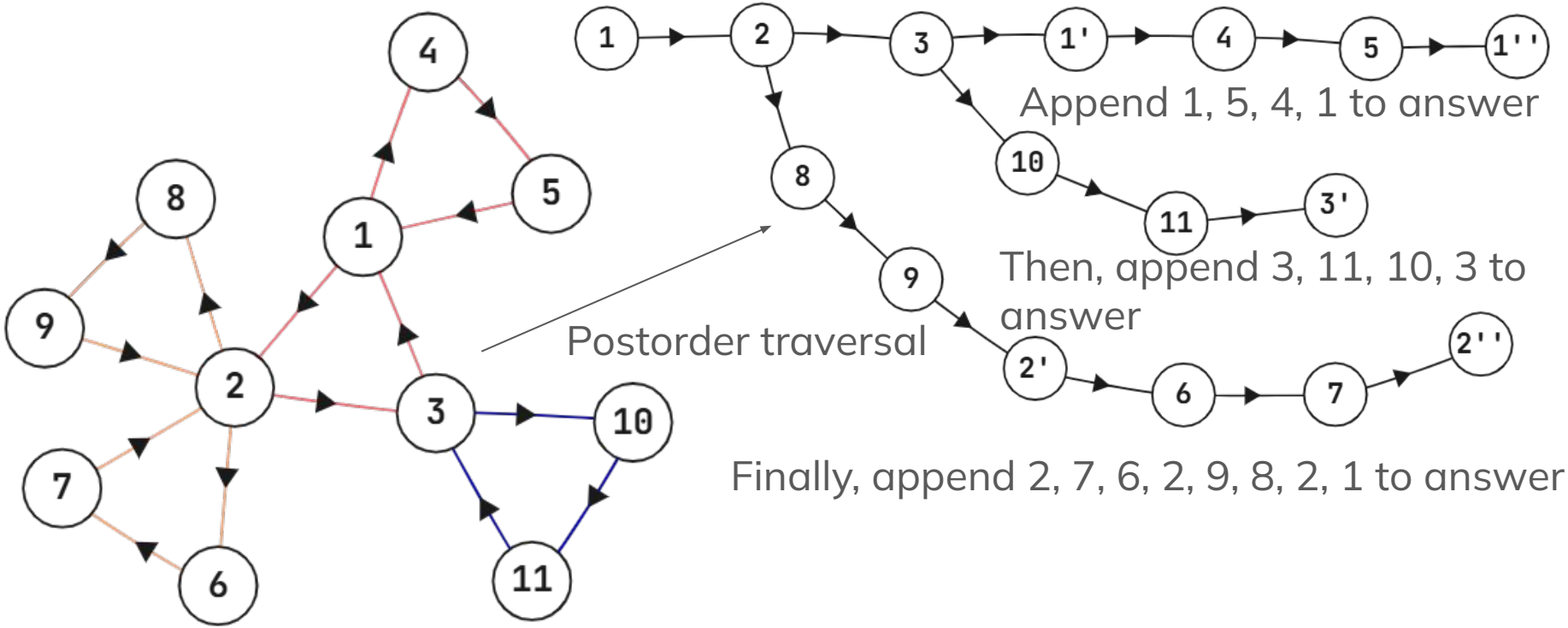
However, it seems like the naive implementation of Hierholzer is O(NM). Can we find a solution with a better time complexity?

Constructing an Euler Cycle: Hierholzer's Algorithm

Instead of doing DFS directly, we try to find the **postorder traversal** of Hierholzer's DFS procedure. It turns out that the solution below works:

- Maintain a stack S. Initially, push any arbitrary node u into it.
  Also, maintain a list L, which is postorder traversal *in reversed order*.
- While the stack S is not empty,
  - If there are still unused outgoing edges from S.top, pick any such edge, mark it as "used", and push the destination of the edge to the stack S.
  - Otherwise, if S.top has no unused edges, we append S.top to the list L, and pop the stack S.
- The reversed version of L gives an Euler cycle of the directed graph.

Constructing an Euler Cycle: Hierholzer's Algorithm



Constructing an Euler Cycle: Hierholzer's Algorithm

Why would this work?

- The only reason that S.top has no unused edges is that we are "stuck".
- But the only situation where you might get "stuck" is when you have completed a cycle.
- By popping a node u and adding it to the list L, you are declaring that everything reachable from this node has already been completely explored.

```
vector <int> adj[MXN];
vector <int> get_cycle() {
  stack <int> s;
  vector <int> cycle;
  s.push(1);
  while (!s.empty()) {
    int u = s.top();
    if (!adj[u].empty()) {
      s.push(adj[u].back());
      adj[u].pop_back();
    } else {
      cycle.push_back(u);
      s.pop();
    }
  }
  reverse(cycle.begin(), cycle.end());
  return cycle;
}
```

Off-topic: Distinct Euler cycles

- In general, if there exists some Euler cycle in a graph, it is not necessary (and generally is not) unique.

- **Disclaimer**: the following section is definitely off-syllabus for IOI and NOI.

- Counting the number of distinct Euler cycles in a general *undirected* graph is #P-complete (at least as hard as NP-complete).
- Intuitively, you might expect the number of distinct Euler cycles in a *directed* graph to be difficult to calculate too, at least not in polynomial time.

Off-topic: Distinct Euler cycles

- Surprisingly, computing the number of distinct Euler cycles in a directed graph can be done in polynomial time!

BEST Theorem: For a directed graph,

number of Euler cycles = (number of root-directed spanning trees with u as root) * product {i = 1 to N} (indegree[i] - 1)!.

- It is also a surprising result that regardless of which node u we take as root, the number of root-directed spanning trees is always the same.
- To find the number of root-directed spanning trees, one can calculate the a minor of the Laplacian matrix of the graph in $O(N^3)$ time.
- A worthwhile read if you are interested.

Graph Modelling: Euler Cycle

General Advice

It is not as rare as you think that problems whose solutions involve Euler cycles doesn't even give you a graph to start working with.

- You have to construct a graph yourself, so that the condition given in the problem translates to
  - "indegree = outdegree for all nodes", if you are working with a directed graph
  - "degree of all nodes are even", if you are working with an undirected graph

All this theory is quite abstract, so let's demonstrate it with an example problem.

(ICPC North American Championship 2026 Problem B) Boss Rush

- Input gives you $1 \le N \le 3 \times 10^5$, $0 \le A[1], A[2], ..., A[N] < D \le 10^5$ and $1 \le W \le 10^9$. (Original problem has $D \le 10^9$ instead, we just do 10^5 here.)

- This means there are N monsters.
  The i-th monster appears at seconds A[i], A[i] + D, A[i] + 2D, ...
- You can kill a monster at some time t if that monster appears at time t.
- Your killing ability has a W second cooldown (so if you killed a monster at x^{th} second, the earliest time you can kill another monster is x + W).

- Find the earliest time you can kill all monsters.

(ICPC North American Championship 2026 Problem B) Boss Rush

At first glance, you might try some greedy(s) and realise none of it works.

The most difficult part is to realise this problem, which seems completely unrelated to graphs at all, actually involves using Euler cycle.

Let’s unpack the problem step-by-step gradually.

(ICPC North American Championship 2026 Problem B) Boss Rush

Observation: If we can kill all monsters within t seconds, then we can kill all monsters within t + 1 seconds.

- This is very obvious. However, this is useful because it implies we can do binary search on answer.

How can we check whether we can kill all monsters in t seconds or not?

- For reasons which will be apparent later, it is actually more convenient to check whether we can kill all monsters in (t - W) seconds or not.

(ICPC North American Championship 2026 Problem B) Boss Rush

How can we check whether we can kill all monsters in (t - W) seconds or not?

Observation: If the answer is yes, then for every $0 \le m < D$, among the seconds $M := \{m, m + D, m + 2D, \ldots \text{ until } t\}$, there are exactly

$$\text{sum } \{i = 1 \text{ to } N\} \text{ of } |M \cap [A[i], A[i] + W - 1]|$$

seconds of them when you are experiencing cooldown.

Proof: This is because the cooldown time intervals are disjoint.

(ICPC North American Championship 2026 Problem B) Boss Rush

This naturally leads to the following claim.

Fact: We can kill all monsters in (t - W) seconds iff for every $0 \le m < D$, among the seconds $M := \{m, m + D, m + 2D, \ldots \text{ until } t\}$, there are exactly

$$\text{sum } \{i = 1 \text{ to } N\} \text{ of } |M \cap [A[i], A[i] + W - 1]|$$

seconds of them when you are experiencing cooldown.

- Does this remind you of any graph construction techniques now?

(ICPC North American Championship 2026 Problem B) Boss Rush

This setup looks very much like Modded Shortest Path (同餘最短路)!

- We consider a graph with D nodes labelled 0, 1, 2, ..., D - 1, indicating the times after taking mod D.

- We can rephrase the above condition as:
  - For every $0 \le m < D$, there are exactly
    $$|M| - \text{sum } \{i = 1 \text{ to } N\} \text{ of } |M \cap [A[i], A[i] + W - 1]|$$
    seconds when you are *not* experiencing cooldown and denote this quantity as cnt[m].
  - The values cnt[0], cnt[1], ..., cnt[D - 1] can be computed together in O(N + D) time.

(ICPC North American Championship 2026 Problem B) Boss Rush

- We consider a graph with D nodes labelled 0, 1, 2, ..., D - 1, indicating the times after taking mod D.

- We construct the following two types of edges in the graph:
  - For every $0 \le m < D$, construct cnt[m] copies of the directed edge $m \to (m + 1) \bmod D$.
  - For every $1 \le i \le N$, construct a directed edge $A[i] \bmod D \to (A[i] + W) \bmod D$.

This leads to the following fact.

Important Observation: We can kill all monsters in (t - W) seconds iff the above directed graph has an Euler cycle.

(ICPC North American Championship 2026 Problem B) Boss Rush

Unfortunately, if we explicitly built the graph, it will clearly lead to TLE.

- However, recall that an equivalent condition for an Euler cycle to exist is that **indegree = outdegree for all nodes**.
- We don't need to build the graph; we only need to maintain the indegree and outdegree of each node.

Therefore, we have a O((N + D) * log ans) solution.

(ICPC North American Championship 2026 Problem B) Boss Rush

To solve the original version of the problem (D up to 10^9), observe that a lot of nodes in the middle are redundant. The only non-redundant nodes are those A[i] mod D and (A[i] + W) mod D.

- Discretise those values to speed up the graph construction part to O(N).

Takeaways:

- After considering the cooldown and non-cooldown times modulo D, we realised that the information we want can be maintained with a graph.
- After describing the condition in Euler cycles, we only need to care about the in- and out-degrees of each node, speeding up our solution.
- Euler cycles can appear in rather unexpecting places. Keep an open mind!

Centroid Decomposition

Introductory Problem

Problem:

- Given a tree of N nodes, where the i-th node has value A[i].
  - A[i] can be positive, negative, or zero.
- Count the number of pairs (u, v) such that the sum of node values along the simple path u → v is exactly K (some fixed constant).

Some initial comments:

- The trivial solution is $O(N^2)$.
  - Exhaust the starting node u and DFS starting from it, then check for each node v naively.
- How do we achieve a sub-quadratic solution?

Introductory Problem (Chain Case)

It is almost always helpful to think of special cases for tree problems first.

For now, consider a subtask where the tree is a chain:

Problem-:

- Given an *array* of size N, where the i-th element is A[i].
- Count the number of pairs of indices (u, v) such that the elements between them (including u and v) sum to exactly K (some fixed constant).

Introductory Problem (Chain Case)

Note that we can further simplify the problem to counting the number of pairs $u < v$ where $A[u] + A[u + 1] + \ldots + A[v] = K$.

- This is because we can multiply our new count by 2, and check the singletons (pairs with $u = v$) naively to get our answer to our original problem.

Problem--:

- Given an *array* of size N, where the i-th element is A[i].
- Count the number of pairs $u < v$, where $A[u] + A[u + 1] + \ldots + A[v] = K$.

Introductory Problem (Chain Case)

Problem--:

- Given an *array* of size N, where the i-th element is A[i].
- Count the number of pairs u < v, where A[u] + A[u + 1] + … + A[v] = K.

Solution: There is a O(N log N) solution which is more straightforward, but let's use a divide-and-conquer approach here.

- Let solve(l, r) be the number of such pairs with $l \le u < v \le r$.
  - Our answer is just solve(1, N).
- If we let m = (l + r) / 2, then solve(l, r) = solve(l, m) + solve(m + 1, r) + (number of such pairs with $l \le u \le m < v \le r$).

Introductory Problem (Chain Case)

How shall we efficiently count the number of such pairs with $l \le u \le m < v \le r$?

Consider the partial sum array of A, given by ps[i] = sum of A[1..i].

- Our condition becomes ps[v] - ps[u - 1] = K.
- We can use a std::map to store the number of occurrences of ps[u - 1] where $l \le u \le m$.
- To compute the count, we iterate over v from m + 1 to r, and add the number of occurrences of ps[v] - K in the map to our answer.

Introductory Problem (Chain Case)

In fact, if you have thought about the O(N log N) solution, you might realise that our “merging” step is very similar to it.

(Spoiler: this would be helpful for us very soon!)

What is our time complexity?

- We have the recurrence T(N) = 2 * T(N / 2) + O(N log N).
  - The 2 * T(N/2) is because we used a divide-and-conquer approach to divide solve(l, r) into two smaller subproblems.
  - The O(N log N) is because we used std::map which incurs a log factor.

We thus have a **O(N log^2 N)** solution to the chain case.

Introductory Problem

We go back to our original problem on a tree.

Problem: *"Count the number of pairs (u, v) such that the sum of node values along the simple path u → v is exactly K (some fixed constant)."*

Inspired by our solution to the chain case, it seems like a good idea to try breaking the problem into multiple subproblems as well.

Introductory Problem

Root the tree arbitrarily at some node r, and we denote its children to be c[1], c[2], ..., c[x].

- Let solve(l) be the number of such pairs (u, v), where
  - **both nodes u and v are in the subtree of node l**

- Then, solve(l) = solve(c[1]) + solve(c[2]) + ... solve(c[x]) + (number of such pairs with LCA(u, v) = l).

Introductory Problem

Again, we continue to mimic our solution from the chain case.

We want to count the number of such pairs with LCA(u, v) = l.

Let ps[i] be the sum of values along the path l → i.

- Our condition becomes

  (ps[u] - ps[parent[i]]) + (ps[v] - ps[i]) = K

  where u and v are nodes in the subtree of l which are **not from the same children of l.**

Introductory Problem

In other words, we want to count (u, v) where

$$ps[u] + ps[v] = K + ps[i] + ps[parent[i]]$$

(u and v are nodes in the subtree of l, but **not from the same children of l**)

- We can again use a std::map to maintain ps[u] for all nodes u in the subtree of l.
- We then iterate over all nodes v in the subtree of l and add the number of occurrences of K + ps[i] + ps[parent[i]] - ps[v] to the answer.

Introductory Problem

In other words, we want to count (u, v) where

$$ps[u] + ps[v] = K + ps[i] + ps[parent[i]]$$

(u and v are nodes in the subtree of l, but **not from the same children of l**)

- However, we would have counted those (u, v) coming from the same children of l.
- We can simply do the above procedure again from each subtree of c[1], c[2], ... c[x] separately, but we subtract the occurrence from the answer (instead of adding).

Introductory Problem

So, we can see that our solution from the chain case generalises well to the general tree case...?

Let's see what our time complexity is now:

- We have the recurrence T(N) = sum of T(size of children) + O(N log N).
  - The sum of T(size of children) is because we used a divide-and-conquer approach to divide solve(l) into multiple subproblems.
  - The O(N log N) is because we used std::map which incurs a log factor.

Introductory Problem

... wait no, this is O(N^2 log N) in the worst case!

- If we rooted at one end of a chain, then our time complexity becomes sum {i = 1 to N} i log i = O(N^2 log N).
  - This is **even worse than our trivial O(N^2) solution**.

Why? Because our very first assumption was fatal: **we cannot root the tree at any arbitrary node.**

- We have to find some way to make sure that we can *reasonably* control "sum of T(size of children)" over all nodes to be not too large.

Introductory Problem

- We have to find some way to make sure that we can *reasonably* control "sum of T(size of children)" over all nodes to be not too large.

How might we be able to achieve this?

- Recall that in the chain case, we picked m = (l + r) / 2, the midpoint of the subarray. This is the "key ingredient" that guarantees our time complexity.

- Perhaps we can pick somehow the "midpoint" of a tree now?

Centroid of a Tree

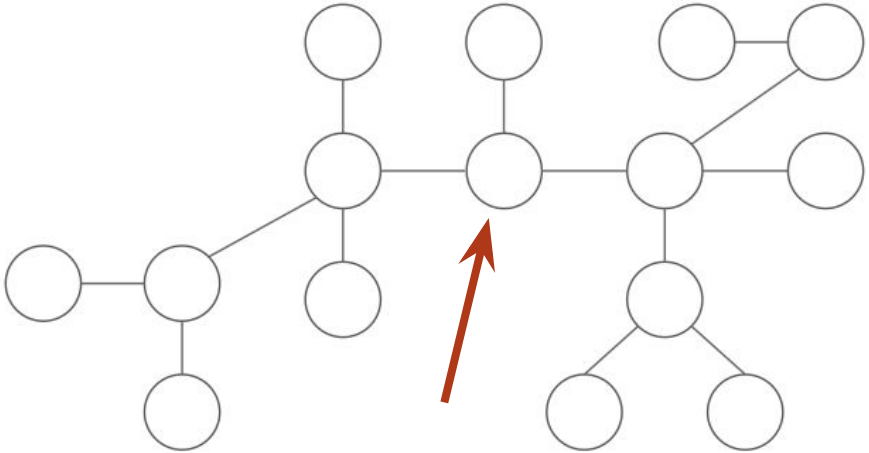
A **centroid** of a tree is defined as:

- A node such that when the tree (of size N) is rooted at it, no other nodes have a subtree of size greater than N / 2.

Fact: The number of centroids of any tree is always either 1 or 2. (Why?)

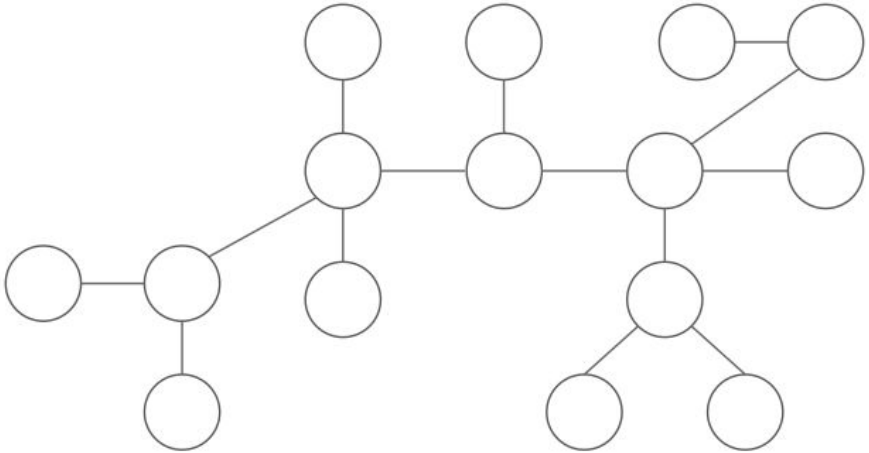
Centroid on Tree

A visual example: which is centroid?



Centroid on Tree

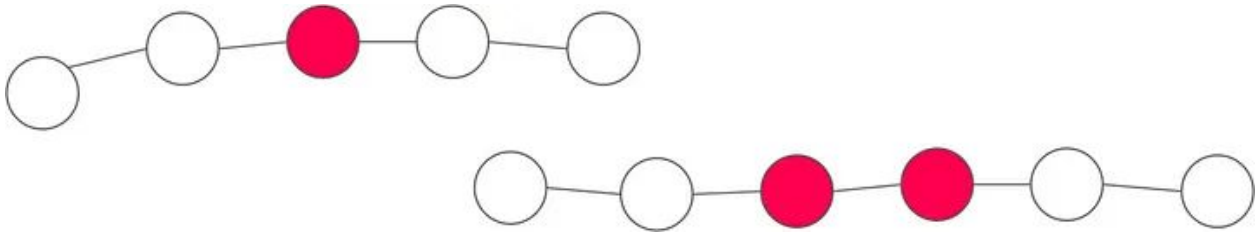
Answer:



Centroid vs Center

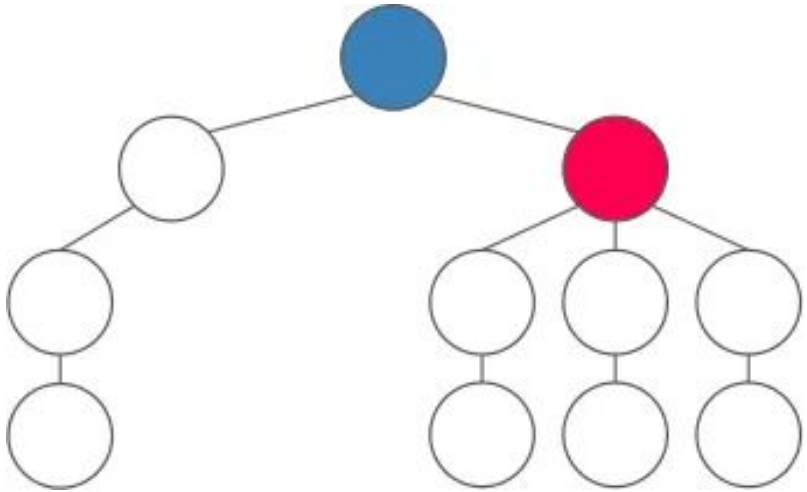
Centroid might not be the same as the **center** of a tree:

- the middle node(s) (either 1 or 2) in every longest path of the tree.



Centroid vs Center

Example: the red node is the centroid but the blue node is the center.



Finding a Centroid in a Tree

Given a node u, how could we check whether it is a centroid or not?

- Root the tree at node u, and compute (with a DFS) the subtree size of every node.
- Node u is a centroid if and only if every children of it has subtree size ≤ N / 2.

Note that there can be at most one child of a node with subtree size > N / 2.

- This fact gives us a way to find a centroid using DFS.

Finding Centroid

Claim: the following algorithm finds a centroid.

- Root the tree arbitrarily.
- Calculate the subtree size of every node.
- Start considering from root node:
  - If any of its children has subtree size > N/2, then the centroid must be within the subtree of that children, so further consider that node.
  - Otherwise, the current node is a centroid.

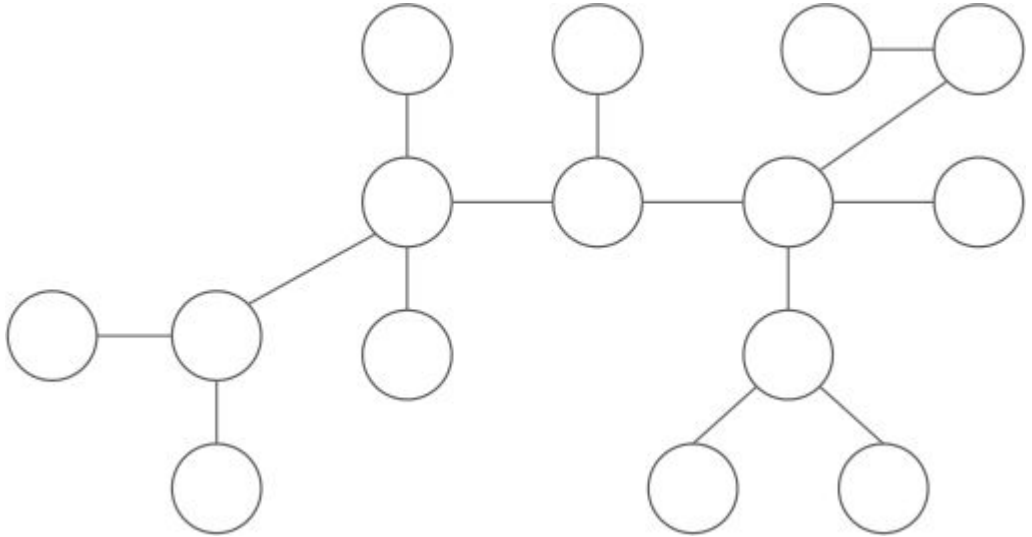
```
int dfs(int cur, int prv) {
  siz[cur] = 1;
  for (int i: adj[cur]) {
    if (i == prv) continue;
    siz[cur] += dfs(i, cur);
  }
  return siz[cur];
}

int find_centroid(int cur, int prv) {
  for (int i: adj[cur]) {
    if (i == prv) continue;
    if (2 * siz[i] > N)
      return find_centroid(i, cur);
  }
  return cur;
}

// call dfs(r, 0) first, then call
find_centroid(r, 0) to find the
centroid of a tree.
```

Finding Centroid

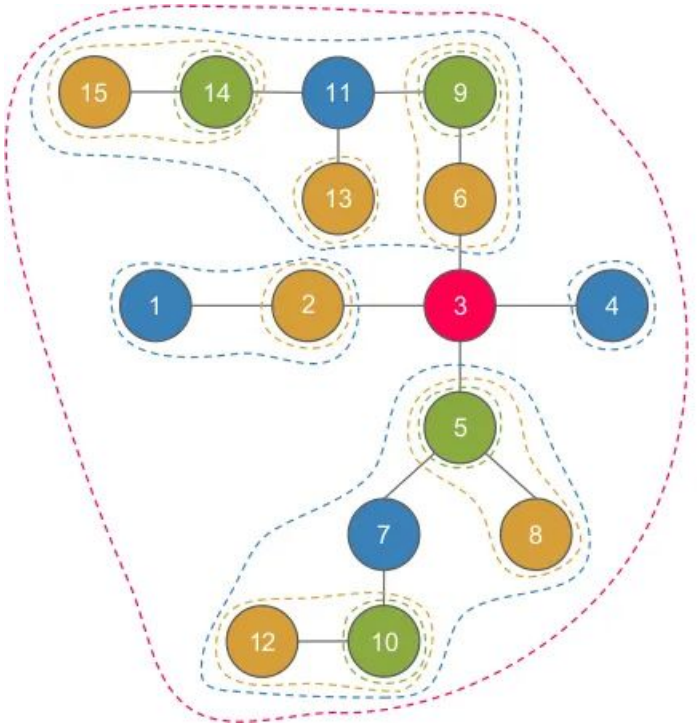
- Root the tree arbitrarily.
- Calculate the subtree size of every node.
- Start considering from root node:
  - If any of its children has subtree size > N/2, then further consider that node.
  - Otherwise, the current node is a centroid.



Centroid Decomposition Tree

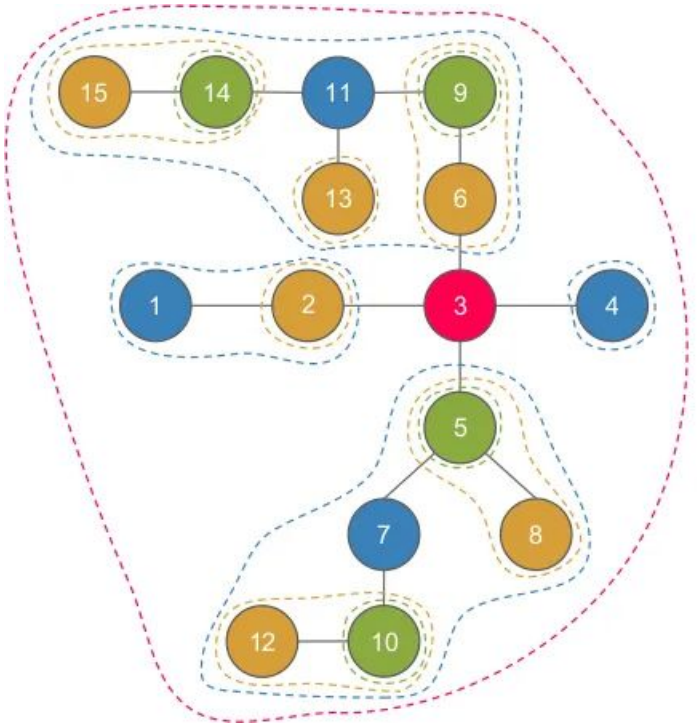
We introduce a concept that will help us make the concept of centroid decomposition clearer: the ***centroid decomposition tree***.

- The centroid of the whole tree is 3.
  Node 3 partitions the remaining tree into four components.
- For each of the four separate components, we find a centroid of each of them.
  - A centroid of {1, 2} is 1.
  - A centroid of {4} is 4.
  - A centroid of {5, 7, 8, 10, 12} is 7.
  - A centroid of {6, 9, 11, 13, 14, 15} is 11.



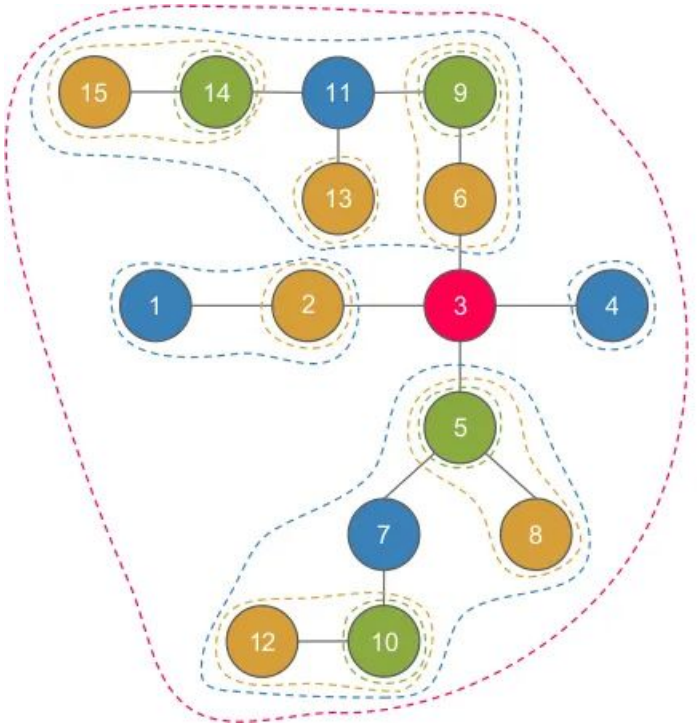
Centroid Decomposition Tree

- A centroid of {1, 2} is 1.
  - Node 1 partitions the remaining subtree into 1 component: {2}.
- A centroid of {4} is 4.
  - The remaining subtree is empty.
- A centroid of {5, 7, 8, 10, 12} is 7.
  - Node 7 partitions the remaining subtree into 2 components: {5, 8} and {10, 12}.
- A centroid of {6, 9, 11, 13, 14, 15} is 11.
  - Node 11 partitions the remaining subtree into 3 components: {6, 9}, {13}, {14, 15}.



Centroid Decomposition Tree

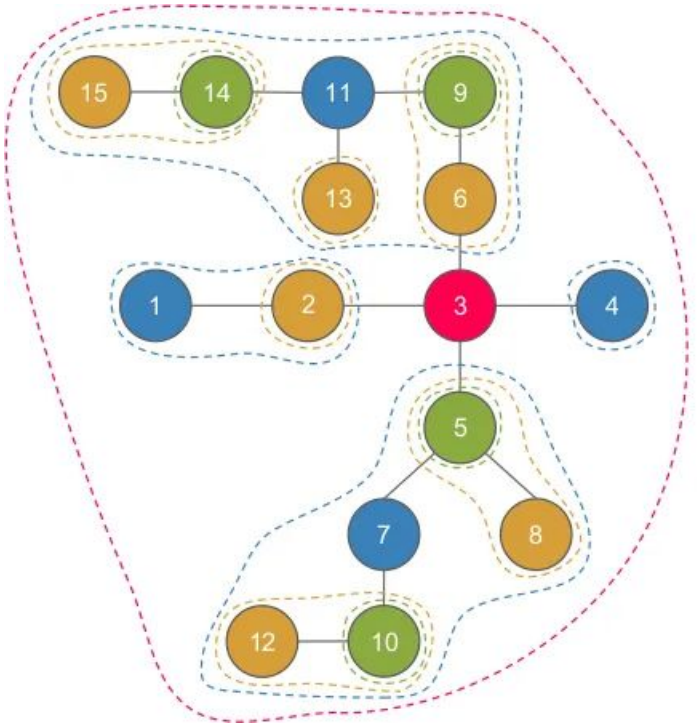
- For each of the 1 + 0 + 2 + 3 = 6 remaining components, we find a centroid of each of them.
  - A centroid of {2} is 2.
  - A centroid of {5, 8} is 8.
  - A centroid of {10, 12} is 12.
  - A centroid of {6, 9} is 6.
  - A centroid of {13} is 13.
  - A centroid of {14, 15} is 15.
- ... and so on. We do this recursively for each component until all remaining subtrees are empty.



Centroid Decomposition Tree

- After decomposing the subtrees, we can form a new tree by adding an edge between the centroid of the current component, and each of the centroids of the subtrees.

- The root of the new tree is the centroid of original tree that we found.

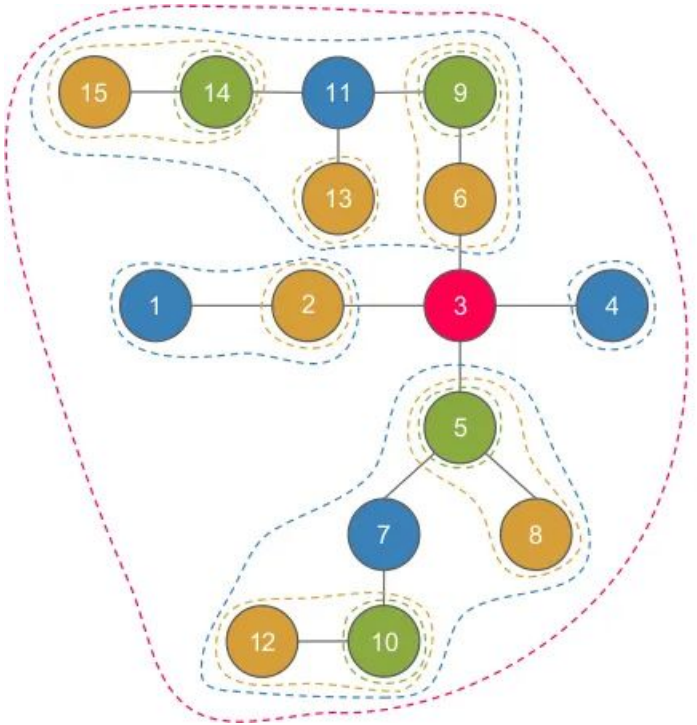


Centroid Decomposition Tree

- For the first layer, we add edges between 3 and 1, 4, 7, 11.

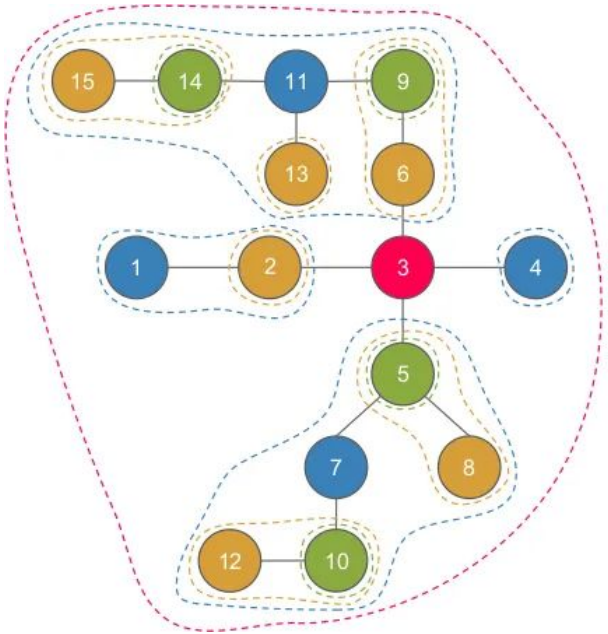
- For the second layer, we add edges
  - Between 1 and 2.
  - Between 7 and 8, 12.
  - Between 11 and 6, 13, 15.

- ... and so on.

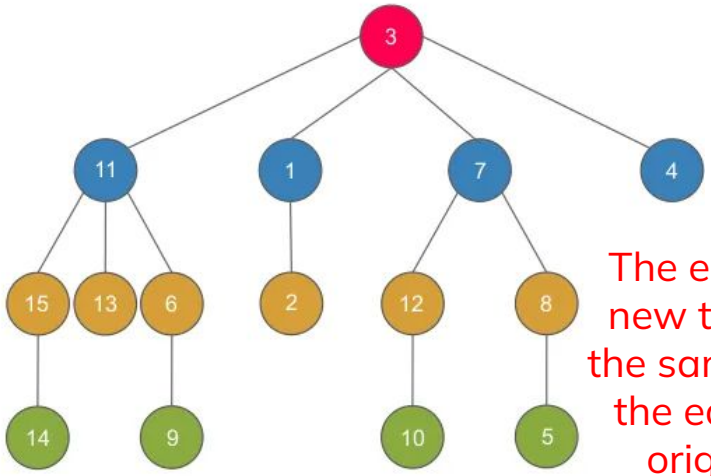


Centroid Decomposition Tree

Original Tree



Centroid Decomposition Tree



The edges in the new tree are **not** the same edges as the edges in the original tree.

Building the Centroid Decomposition Tree

```
bool deleted[MXN];
int siz[MXN];
int dfs(int cur, int prv) {
  siz[cur] = 1;
  for (int i: adj[cur]) {
    if (i == prv || deleted[i]) continue;
    siz[cur] += dfs(i, cur);
  }
  return siz[cur];
}
int find_centroid(int cur, int prv,
                  int treeSize) {
  for (int i: adj[cur]) {
    if (i == prv || deleted[i]) continue;
    if (2 * siz[i] > treeSize)
      return find_centroid(i, cur, treeSize);
  }
  return cur;
}
```

Remember to not visit deleted nodes, and recalculate subtree sizes every time.

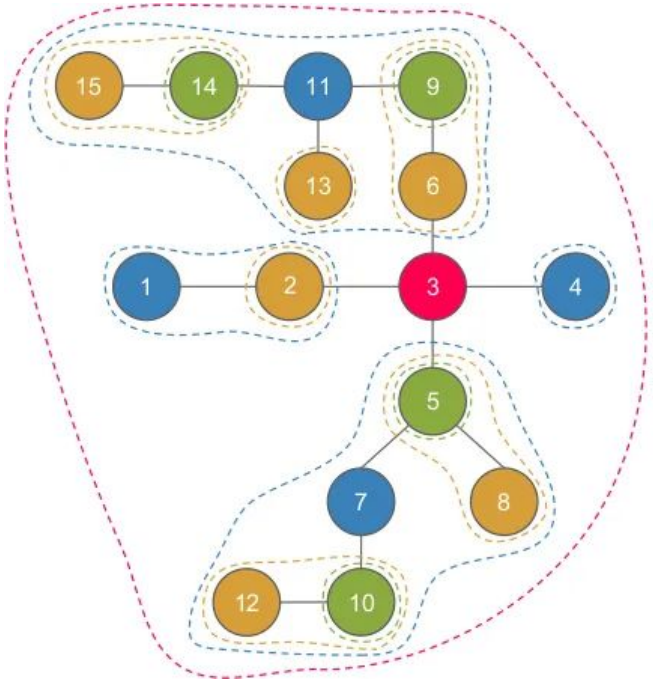
```
int decomposition(int cur){
  int r = find_centroid(cur, 0, dfs(cur, 0));
  deleted[r] = true;
  // calculate things
  for (int i: adj[r]) {
    if (!deleted[i]){
        int c = decomposition(i);
        /* add an edge in the
           centroid decomposition tree
           between nodes r and c */
    }
  }
  return r;
}
// call decomposition(u) for any arbitrary node u
```

Properties of Centroid Decomposition Tree

Fact 1: The height of the centroid decomposition tree is at most log N.

- This is because by the property of centroids, the subtree size of any node in the i^{th} layer (0-based) is at most $N / 2^i$.

Corollary 2: The sum of subtree sizes of the centroid decomposition tree is at most $N \times (\log N + 1)$.

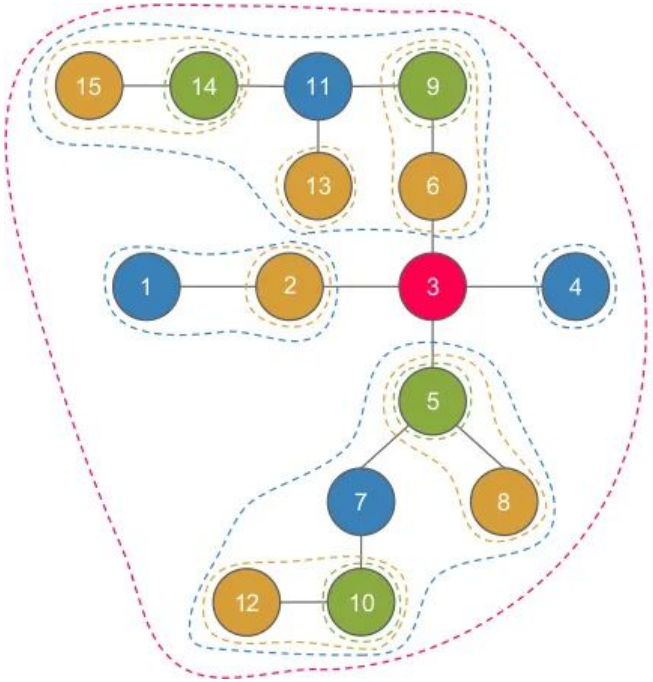


Properties of Centroid Decomposition Tree

Fact 3: A node belongs to the component of all its ancestors (in the decomposition tree).

- Here, the component of a node u is the remaining tree connected to it when node u is the centroid of the remaining tree.
  - e.g. the component of node 11 is {6, 9, 11, 13, 14, 15}.

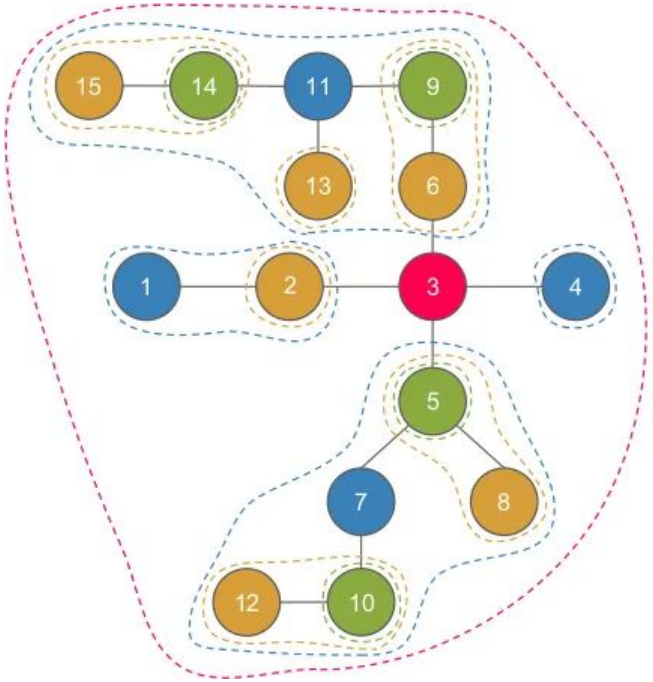
- For example, the node 14 belongs to the component of all its ancestors in the centroid decomposition tree: 3, 11, 15, 14.



Properties of Centroid Decomposition Tree

Fact 4: For any two nodes u and v, the LCA of u and v **in the centroid decomposition tree** always lies on the simple path u → v of the **original tree**.

- For example, if u = 14 and v = 6, then the LCA of u and v in the decomposition tree is 11, which lies on the simple path 14 – 11 – 9 – 6.
- This means we can always decompose a simple path u → v into u → $LCA_{\text{centroid tree}}(u, v)$ → v.

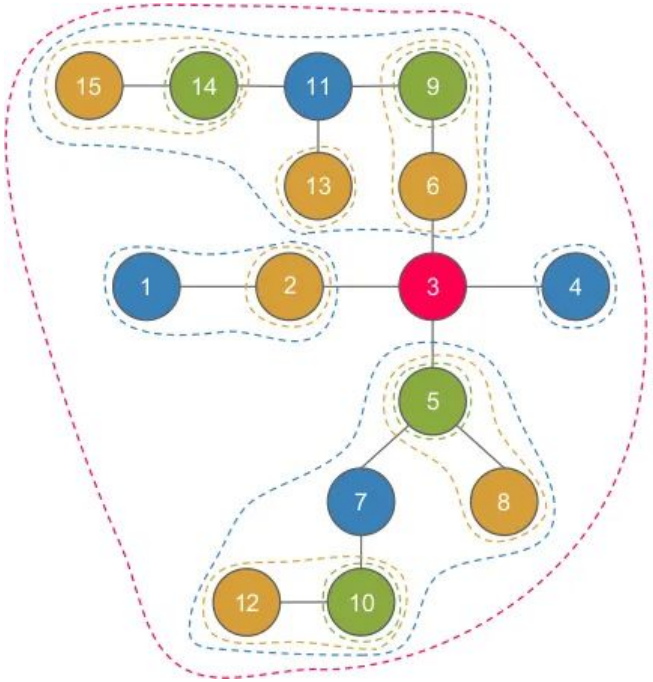


Properties of Centroid Decomposition Tree

Corollary 5: Consequently, every pair of nodes u ≠ v can be ***uniquely*** described using a triplet of nodes (l, a, b), where:

- l is any arbitrary node
- both a and b are in the component of node l
- node a and node b are not in the same component of some children of node l.

(In other words, $LCA_{\text{centroid tree}}(a, b) = l$.)



Solution to Introductory Problem

Problem: Count the number of pairs (u, v) such that the sum of node values along the simple path u → v is exactly K.

The first step is, of course, build the centroid decomposition tree. Then, for each node l in the centroid decomposition tree,

- For every node x in the component of l, compute the sum of values on the simple path l → x in the ***original tree***, and denote this as sum[x].
  - This can be done in O(size of component) directly with a DFS.
  - By **Corollary 2**, this takes only O(N log N) time over all u from 1 to N.

Solution to Introductory Problem

For each node l in the centroid decomposition tree,

- For every node x in the component of l, compute the sum of values on the simple path l → x in the ***original tree***, and denote this as sum[x].

- Count the number of pairs of (u, v), where u and v are nodes in the component of l, such that sum[u] + sum[v] - A[l] = K, and add it to the answer.
  - This can be done in O(component size * log (component size)) easily.

- However, we *should not have counted* the pairs of (u, v) where u and v are in the same component of some child of l.

Solution to Introductory Problem

However, we *should not have counted* the pairs of (u, v) where u and v are in the same component of some child of l.

- To fix this, iterate through each child c of node l, and count the number of pairs of (u, v) where u and v are nodes in the component of c that satisfies the same condition (sum[u] + sum[v] - A[l] = K) above.
  - For a node l, this whole process (across all children c) is still only O(component size of node l * log (component size of node l)).
- We *subtract* the count from the answer now instead of adding it.

Therefore, we have a working solution that works in **O(N log^2 N)**.

Example Problem: (Codeforces 342E) Xenia and Tree

Almost everyone does this as their first centroid decomposition task, so it would be criminal to exclude it here.

Input gives you an unweighted tree with $N \leq 10^5$ nodes. Initially, all nodes are coloured blue. Support these (up to 10^5) operations online:

- Given a node x which is currently coloured blue, colour node x red.
- Given a node y, find the distance from node y to the closest red node.

Example Problem: (Codeforces 342E) Xenia and Tree

The first step of doing a centroid decomposition problem is always to build the centroid decomposition tree.

- We let ans[u] to be distance *(in terms of the original tree)* from u to the closest red node in the component of u.
  - If there are no red nodes in the component of u, we set ans[u] to be infinity.
- Suppose we want to change a blue node x to red now.
  What should we do?

Example Problem: (Codeforces 342E) Xenia and Tree

Suppose we want to change a blue node x to red now.

- Recall **Fact 3:** a node belongs to the component of all its ancestors (in the centroid decomposition tree).

- So, we find all the ancestors of node x in the centroid decomposition tree.
- Denote them as a[1], a[2], ..., a[i], ..., then we set

$$ans[a[i]] = \min(ans[a[i]], dist_{\text{original tree}}(a[i], x))$$

for each i.

Example Problem: (Codeforces 342E) Xenia and Tree

Suppose we want to change a blue node x to red now.

- Denote them as a[1], a[2], ..., a[i], ..., then we set ans[a[i]] = min(ans[a[i]], $dist_{\text{original tree}}$(a[i], x)) for each i.

- To find all the ancestors of a node, maintain the parent of each node while constructing the decomposition tree. Jump up layer-by-layer until we reach the root. By **Fact 1**, there are O(log N) such ancestors.
- Finding distance in the original tree can be done in O(log N) time: see Graph (III).

So, each update takes **$O(\log^2 N)$** time.

Example Problem: (Codeforces 342E) Xenia and Tree

We now know how to handle updates; what about query on a node y?

- Recall **Fact 4:** For nodes u and v, the LCA of u and v **in the centroid decomposition tree** lies on the simple path u → v of the **original tree**.

- Again, we iterate through all the ancestors of node y, and denote that as a[1], a[2], ..., a[i], ...
- By **Fact 4**, we know the answer is the minimum value of

$$\text{dist}_{\text{original tree}}(y, a[i]) + \text{ans}[a[i]]$$

over all ancestors a[i] of y.

Example Problem: (Codeforces 342E) Xenia and Tree

...wait. Won't we "overcount" something?

- It might be the case that node y and the red node with the shortest distance to a[i] are both from the same component of some child of a[i].
- In that case, our answer is invalid!

- Actually, if the above situation happened, then the path y → a[i] → the red node is *not* simple, so our value is larger than the actual distance.
- We are only interested in the *minimum* distance, so this actually doesn't affect our answer at all.

Time complexity is the same as updates: **$O(\log^2 N)$** per query.

Heavy Light Decomposition

Introductory Problem

- HLD is a very blackbox-able algorithm.
  - It is quite rare that you would have to modify the decomposition.

- Our target problem: given a tree with N nodes (rooted at node 1), and each node has a value A[i] which are initially all 0.
- Support multiple queries online:
  - Given u, v, and x, do A[i] += x for all nodes i on the simple path between u and v.
  - Given u and v, query the sum of A[i] over all nodes i on the simple path between u and v.

- Again, if we don't know how to do the general tree case, it never hurts to consider when the tree is just a chain to get a better "feel" of the problem.

Introductory Problem (Chain Case)

- Update: Given u, v, and x, do A[i] += x for all i between u and v.
- Query: Given u and v, query the sum of A[i] over i between u and v.

- This is just segment tree with lazy propagation, which is already covered in Data Structures (IV).

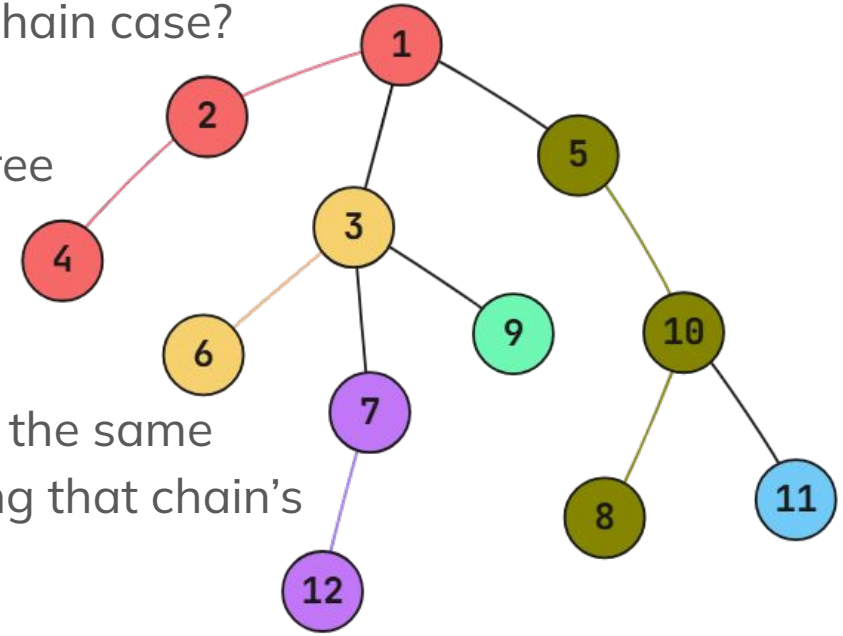
... wow, quite straightforward.

- Since we know how to do the chain case comfortably, can we somehow turn the general case into some instances of the chain case?

Introductory Problem (Chain++?)

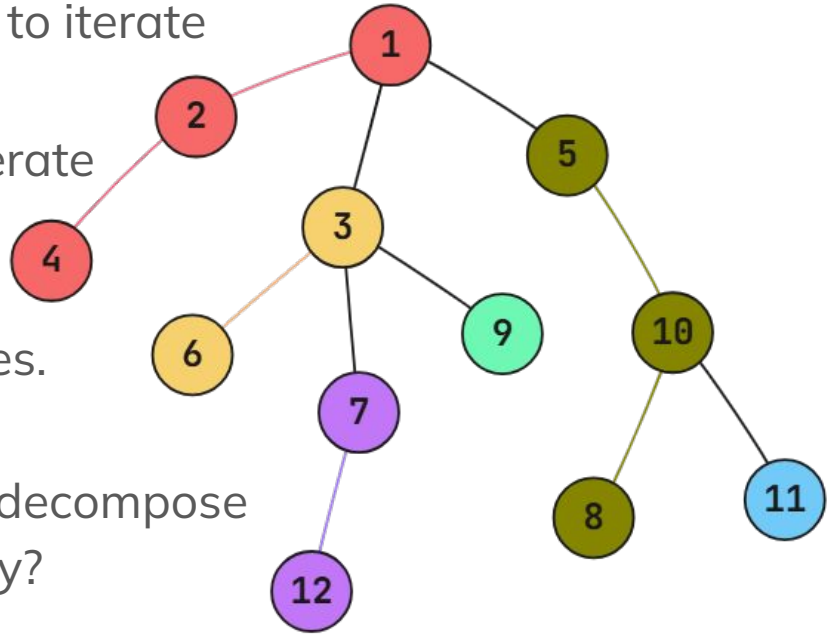
Since we know how to do the chain case comfortably, can we somehow turn the general case into some instances of the chain case?

- Let's try decomposing the nodes in the tree into multiple chains arbitrarily.
- We can maintain a segment tree for the nodes on each chain.
- If the two nodes u and v happen to be in the same chain, then we can directly maintain using that chain's segment tree!



Introductory Problem (Chain++?)

- But what if the nodes u and v are in different chains?
- If u = 11 and v = 12, then we would have to iterate through 11 – 10 – 5 – 1 – 3 – 7 – 12.
  ⇒ in the worst case, we might have to iterate through O(N) different chains, and for each of them we need to update the corresponding nodes on the segment trees.



- Come up with a more systematic way to decompose chains and guarantee our time complexity?

Introductory Problem (Chain++?)

Let's take a step back and think about what we want.

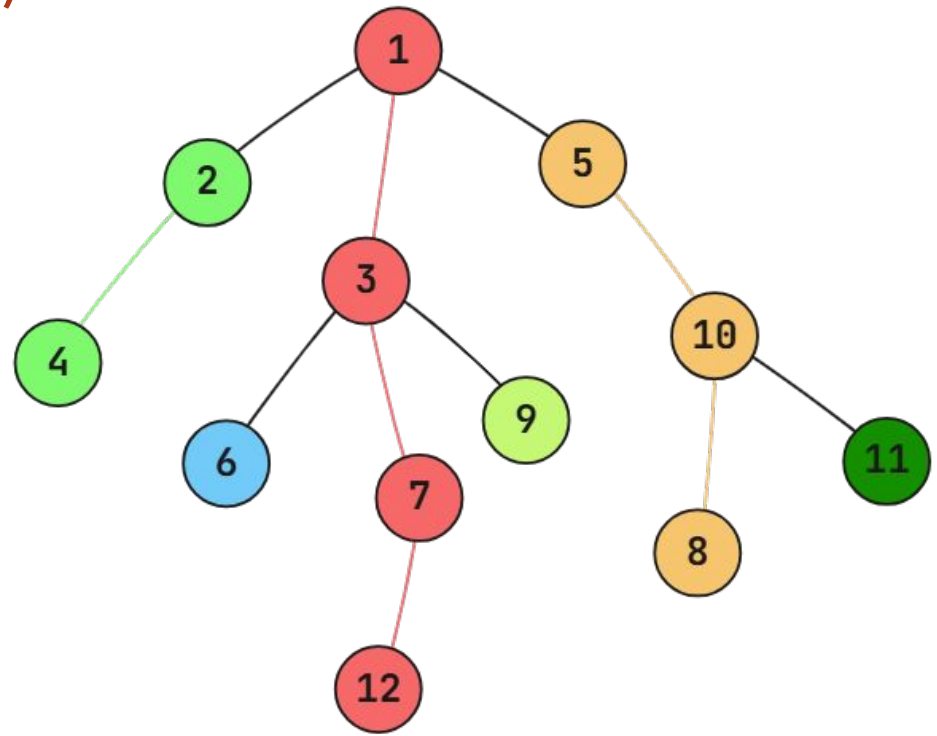
- For every pair of node (u, v), we want the number of distinct chains that the simple path u → v goes through is controllable (not too large).

Consider the following **heuristic:**

- Start at the root and choose a chain from the root to a leaf as follows:
  - If the current node has no children, then we have reach a leaf, so done.
  - Otherwise, choose the child whose subtree size is the largest (tiebreak arbitrarily). Add it to the chain and continue from that child downwards (until it reaches a leaf).
- This partitions the tree into several rooted subtrees. Solve each of them recursively.

Introductory Problem (Chain++?)

- Start at the root and choose a chain from the root to a leaf:
  - Choose the child whose subtree size is the largest. Add it to the chain and continue from that child downwards (until it reaches a leaf).
- Pick 1 – 3 – 7 – 12, then
  - Pick 2 – 4
  - Pick 6
  - Pick 9
  - Pick 5 – 10 – 8, then
    - Pick 11



Heavy Light Decomposition

Our heuristic we use is commonly called “heavy light decomposition”.

It is called “heavy light” because for every node,

- its child with the largest subtree size is known as the “***heavy***” child.
  - the edge connecting to its heavy child is called the ***heavy*** edge.
- all its other children are known as a “***light***” child.
  - edges connecting to its light children are called ***light*** edges.

⇒ for each (non-leaf) node, we extend the current chain onto its heavy child, and start a new chain for each of its light child.

Heavy Light Decomposition

Good news! Our worst-case scenario is actually still quite “controlled”.

Claim: If node v is an ancestor of node u, then node v can be reached from node u by visiting at most ceil(log N) distinct chains.

Proof:

- Denote the nodes on the path as u = p[1] – p[2] – … – p[k] = v.
- Our **Claim** is equivalent to showing that there are at most (ceil(log N) - 1) indices i, such that p[i] and p[i + 1] belong to different chains.
- If p[i] and p[i + 1] belong to different chains, then the subtree size of p[i + 1] must be larger than 2 * (subtree size of p[i]).

Heavy Light Decomposition

Claim: If node v is an ancestor of node u, then node v can be reached from node u by visiting at most ceil(log N) distinct chains.

Proof:

- If p[i] and p[i + 1] belong to different chains, then the subtree size of p[i + 1] must be larger than 2 * (subtree size of p[i]).
  - Because p[i] and p[i + 1] belong to different chains, p[i] is ***not*** the heavy child of p[i + 1].
  - So, there must be some other (heavy) child of p[i + 1] with a larger subtree size than p[i].
- Clearly, this can only happen at most ceil(log N) times. So we are done.

Heavy Light Decomposition

- By the **Claim** above, for every pair of nodes (u, v), the simple path u → v goes through at most 2 * ceil(log N) distinct chains.
  - Going from u to LCA(u, v) goes through at most ceil(log N) chains.
  - Going from LCA(u, v) to v also goes through at most ceil(log N) chains.

Therefore, for each query and update, we go through every chain which intersects the path u → v, and maintain the corresponding values on each chain using a segment tree with lazy propagation.

- Time complexity: $O(\log^2 N)$ per query and update.

Implementing Heavy Light Decomposition

If you try to implement our idea above directly, you would probably quickly find out that it is quite annoying.

There is a common trick that involves relabelling the nodes of the tree. By doing so, you can maintain all the information we need on one *(instead of total number of chains)* segment tree. This makes implementing to be less painful.

Node Relabel Trick--

Let's move slightly aside and consider another problem first:

Problem: Given a tree with N nodes (rooted at node 1), and each node has a value A[i]. Support multiple queries online:

- Given u and x, do A[i] += x for all nodes i **in the subtree** of u.
- Given u, query the sum of A[i] over all nodes i **in the subtree** of u.

Solution: We relabel the nodes according to **DFS order** (starting from the root). This is convenient for us because

- the node labels within a subtree **always form a consecutive interval**.

Node Relabel Trick--

Solution: We relabel the nodes according to **DFS order** (starting from the root). This is convenient for us because

- the node labels within a subtree **always form a consecutive interval**.

So if we precompute dfn[u] to be the DFS order of node u, and siz[u] to be the subtree size of node u, then

- if we want to query / update node u, we just query / update the interval [dfn[u], dfn[u] + siz[u] - 1] on the segment tree.

Can we do something similar here too?

Node Relabel Trick

We relabel the nodes according to **DFS order** (starting from the root).

- Previously, the order in which we iterated through different children of a node was arbitrary.
- Now, we additionally force ourselves to **DFS to the heavy child first**, before visiting the light children of a node.

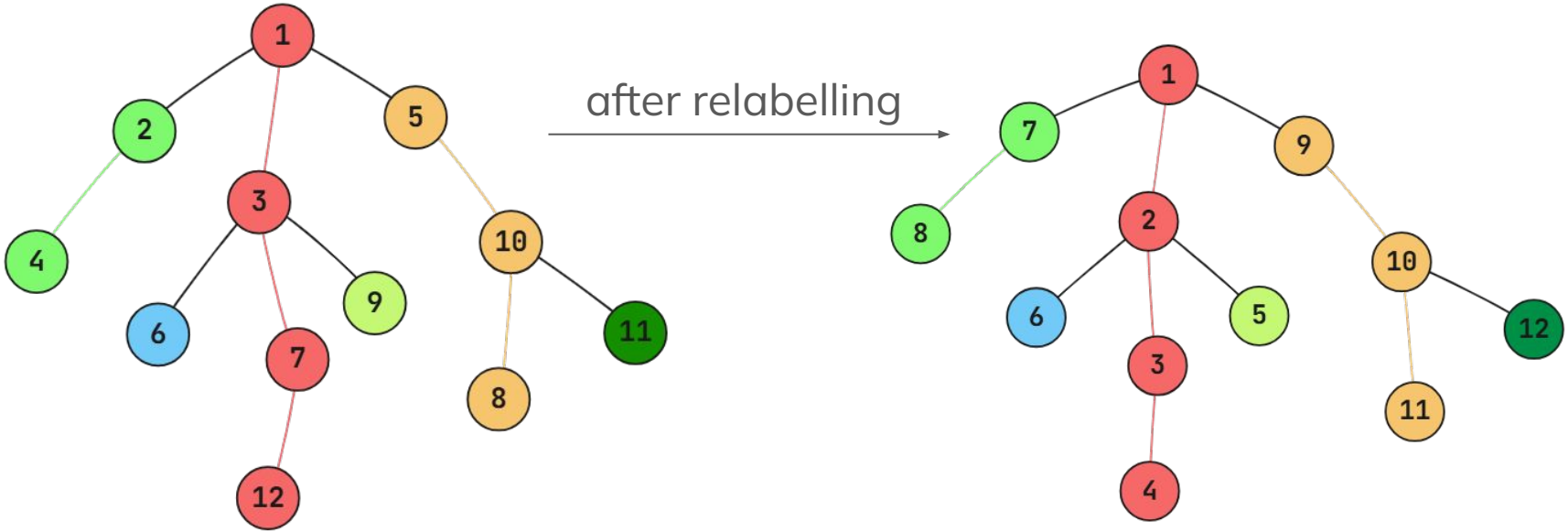
With this, we can achieve the following.

Observation:

- The label of each partitioned chain forms a consecutive interval.
- The label of a node's subtree also forms a consecutive interval.

Node Relabel Trick

DFS to the heavy child first, before visiting the light children of a node.



Implementing Heavy Light Decomposition

We are now ready to implement the whole solution.

```
vector <int> adj[MXN];
int pa[MXN], dep[MXN], siz[MXN], heavy[MXN];

void dfs1(int cur, int prv) {
  pa[cur] = prv;
  dep[cur] = dep[prv] + 1;
  siz[cur] = 1;
  for (int i: adj[cur]) {
    if (i == prv) continue;
    dfs1(i, cur);
    siz[cur] += siz[i];
    if (siz[i] > siz[heavy[cur]]) {
      heavy[cur] = i;
    }
  }
}
```

```
int chainTop[MXN], label[MXN], dfn;

void dfs2(int cur, int from) {
  chainTop[cur] = from;
  label[cur] = ++dfn;
  if (!heavy[cur]) return;
  dfs2(heavy[cur], from);
  for (int i: adj[cur]) {
    if (i != pa[cur] && i != heavy[cur]) dfs2(i, i);
  }
}
```

We pick any arbitrary node u.
Call dfs1(u, 0), and then dfs2(u, u).

Implementing Heavy Light Decomposition

- If nodes u and v are in different chains, we process the one which is currently deeper, maintaining the required information in the chain.
- After processing the interval from chainTop[u] to u, we move up to the parent of chainTop[u] and continue.

```
void add_path(int u, int v, int w) {
  while (chainTop[u] != chainTop[v]) {
    if (dep[chainTop[u]] != dep[chainTop[v]]) swap(u, v);
    SegTree.update(label[chainTop[u]], label[u], w);
    u = pa[chainTop[u]];
  }
  if (dep[u] > dep[v]) swap(u, v);
  SegTree.update(label[u], label[v], w);
}
int get_path_sum(int u, int v) {
  int res = 0;
  while (chainTop[u] != chainTop[v]) {
    if (dep[chainTop[u]] != dep[chainTop[v]]) swap(u, v);
    res += SegmentTree.query(label[chainTop[u]], label[u]);
    u = pa[chainTop[u]];
  }
  if (dep[u] > dep[v]) swap(u, v);
  res += SegmentTree.query(label[u], label[v]);
  return res;
}
```

Remarks

Of course, HLD can support other range operations too, not just sums.

Because we are maintaining things on a segment tree, this basically implies:

- If you can solve a range query / update problem with (only) a segment tree, then you can solve the same problem on a tree with path query / update with an extra log factor using HLD.

General Remarks

- Centroid decomposition is useful to count the number of paths of a tree that satisfy some specified property.
- Heavy light decomposition is useful to maintain information along a given path online.

Neither of the two decomposition techniques would generally require you to modify its inner workings.

- They are more like “tools” rather than “new thinking perspectives”.
- The best way to become proficient with a tool is to **use it**.

Practice Tasks

Euler Path/Cycle

(M2402) Poem of Love
(M2434) Colouriguessr
(M24A1) Balance
(T242) Perfect Tour
(M25AA) Kowloon Walled City

Centroid Decomposition

(CEOI 2019 Day 1 Problem 3) Dynamic Diameter
(JOISC 2026 Day 3 Problem 3) Collecting Stamps 5
(Universal Cup 4-17 Problem G) Traffic Lights

Heavy Light Decomposition

(NOI 2021 Day 1 Problem 1) 輕重邊
(JOISC 2023 Day 3 Problem 3) Tourism
(Korea TST 2024 Day 1 Problem 3) Police
(M2354) Traveler's Mission

(JOISC 2024 Day 3 Problem 2) JOI Tour can be solved using either **Centroid Decomposition** or **Heavy Light Decomposition**.
Try to do it in either (or better, both) ways!