Competitive Training Camp

Lecture 10

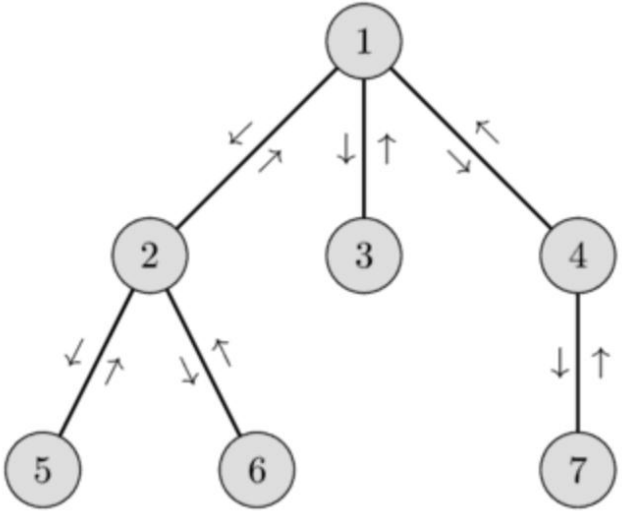
Christian Yongwhan Lim

Wednesday, July 23, 2025

Overview: Graphs, Revisited!

- Lowest Common Ancestor
- Centroid Decomposition
- Heavy-Light Decomposition

Lowest Common Ancestor (LCA): Euler Tour first!



Vertices:	1	2	5	2	6	2	1	3	1	4	7	4	1
Heights:	1	2	3	2	3	2	1	2	1	2	3	2	1

Lowest Common Ancestor (LCA): Example

- The tour starting at vertex 6 and ending at 4 we visit the vertices [6, 2, 1, 3, 1, 4].

- Among those vertices the vertex 1 has the **lowest height**.

- Therefore, LCA(6, 4) = 1!

Lowest Common Ancestor (LCA): Example

- In general, once we pre-process the nodes using Euler tour, you can do a **range minimum query**.

- We know this can be solved using **segment tree** (or, **sparse table**, if the tree is fixed!)

Lowest Common Ancestor (LCA): Implementation

```
struct LCA {
  vector<int> height, euler, first, segtree;
  vector<bool> visited;
  int n;
```

Lowest Common Ancestor (LCA): Implementation

```
LCA(vector<vector<int>> &adj, int root = 0) {
  n = adj.size();
  height.resize(n);
  first.resize(n);
  euler.reserve(n * 2);
  visited.assign(n, false);
  dfs(adj, root);
  int m = euler.size();
  segtree.resize(m * 4);
  build(1, 0, m - 1);
}
```

Lowest Common Ancestor (LCA): Implementation

```
void dfs(vector<vector<int>> &adj, int node, int h=0) {
  visited[node] = true;
  height[node] = h;
  first[node] = euler.size();
  euler.push_back(node);
  for (auto to : adj[node])
    if (!visited[to]) {
      dfs(adj, to, h + 1);
      euler.push_back(node);
    }
}
```

Lowest Common Ancestor (LCA): Implementation

```
void build(int node, int b, int e) {
  if (b == e) {
    segtree[node] = euler[b];
  } else {
    int mid = (b + e) / 2;
    build(node << 1, b, mid);
    build(node << 1 | 1, mid + 1, e);
    int l = segtree[node<<1], r = segtree[node<<1|1];
    segtree[node] = (height[l] < height[r]) ? l : r;
  }
}
```

Lowest Common Ancestor (LCA): Implementation

```
int query(int node, int b, int e, int L, int R) {
  if (b > R || e < L) return -1;
  if (b >= L && e <= R) return segtree[node];
  int mid = (b + e) >> 1;
  int left = query(node << 1, b, mid, L, R);
  int right = query(node << 1 | 1, mid + 1, e, L, R);
  if (left == -1) return right;
  if (right == -1) return left;
  return height[left] < height[right] ? left : right;
}
```

Lowest Common Ancestor (LCA): Implementation

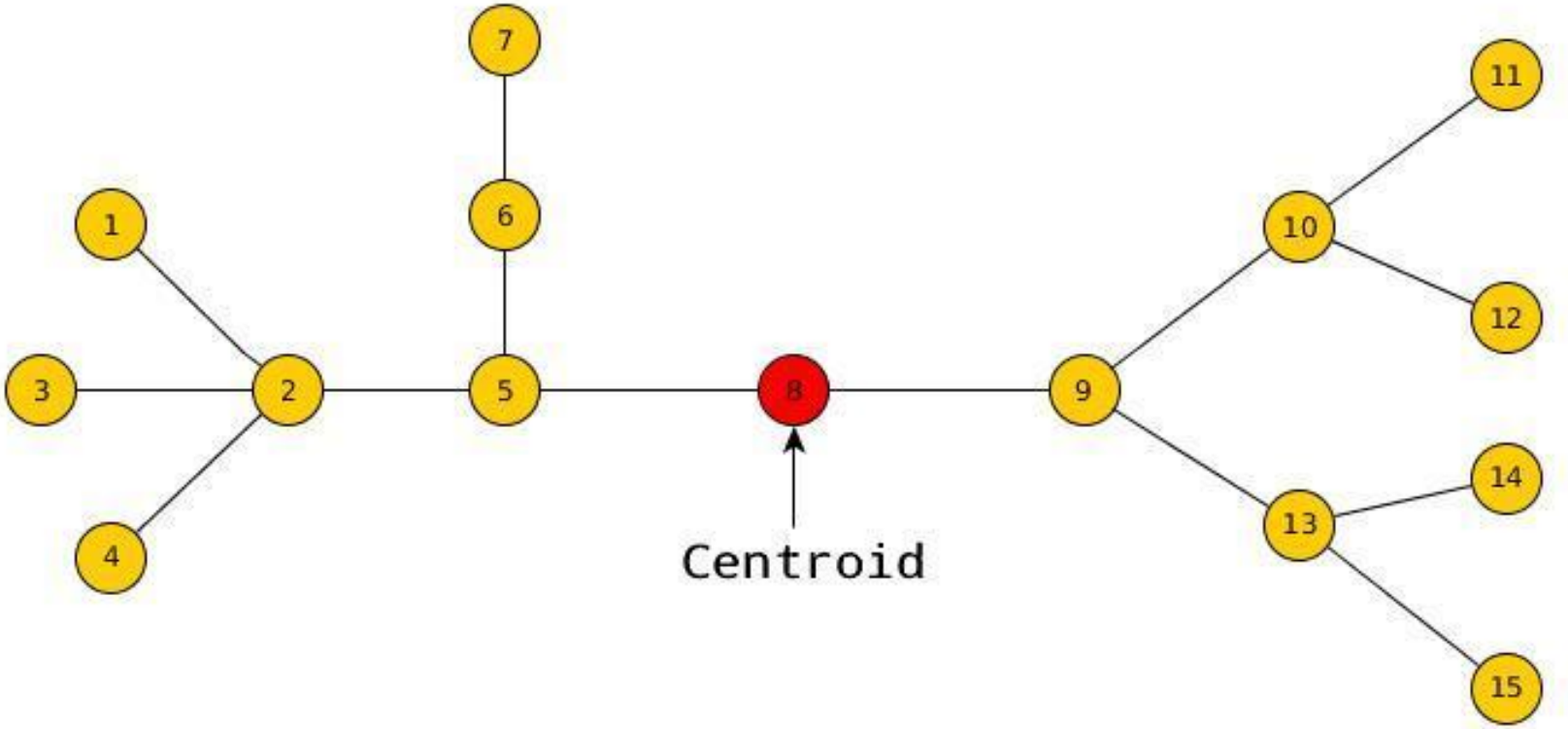
```
  int lca(int u, int v) {
    int left = first[u], right = first[v];
    if (left > right)
      swap(left, right);
    return query(1, 0, euler.size() - 1, left, right);
  }
};
```

Centroid

- **Centroid**: a node such that when the tree is rooted at it, no other nodes have a subtree of size greater than n/2.

- We can find a centroid in a tree by starting at the root.
- Each step, loop through all of its children.
- If all of its children have subtree size less than or equal to n/2, then it is a centroid.
- Otherwise, move to the child with a subtree size that is more than n/2 and repeat until you find a centroid.

Centroid Illustration



Centroid

```
const int maxn = 200010;
int n;
vector<int> adj[maxn];
int subtree_size[maxn];
```

Centroid

```
int get_subtree_size(int node, int par = -1) {
  int &res = subtree_size[node];
  res = 1;
  for (int i : adj[node]) {
    if (i == par) continue;
    res += get_subtree_size(i, node);
  }
  return res;
}
```

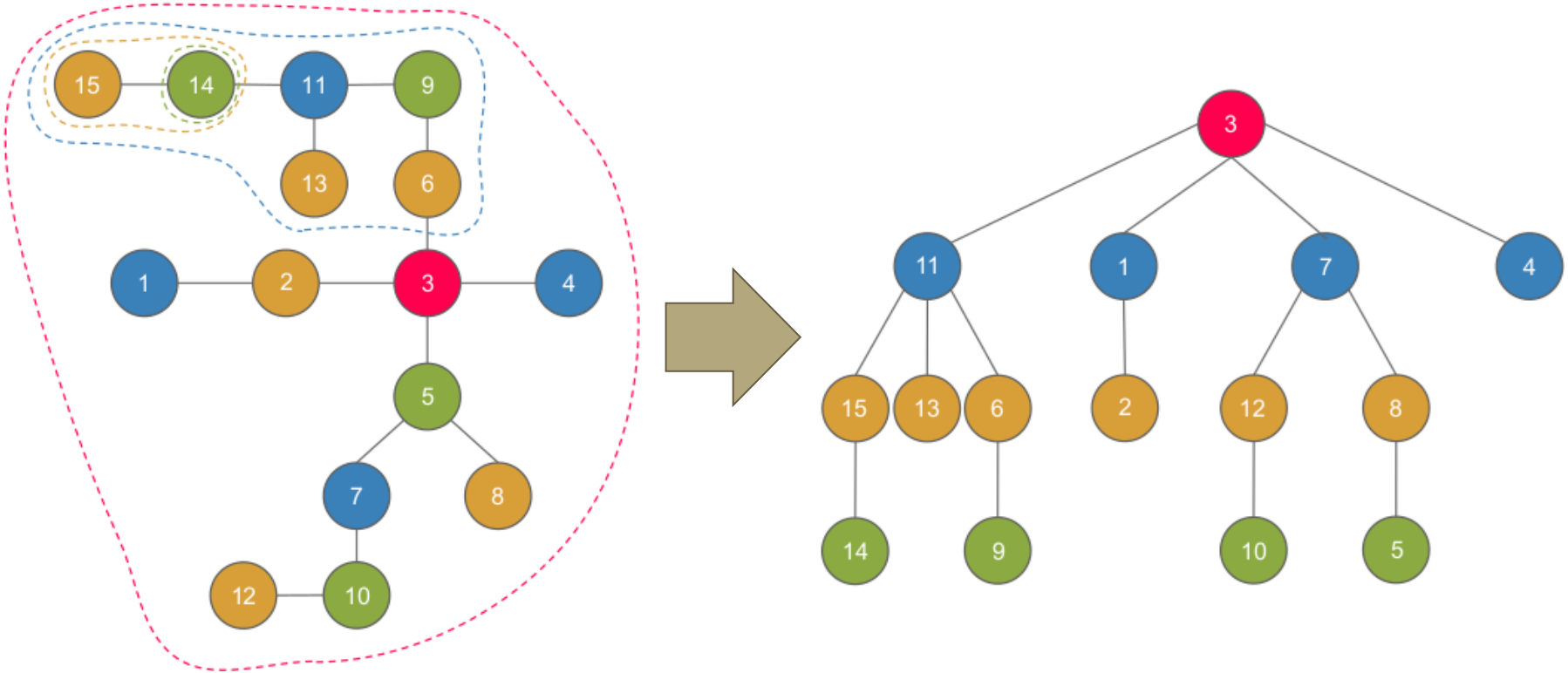
Centroid

```
int get_centroid(int node, int par = -1) {
  for (int i : adj[node]) {
    if (i == par) continue;
    if (subtree_size[i] * 2 > n)
        return get_centroid(i, node);
  }
  return node;
}
```

Centroid Decomposition

- **Centroid Decomposition** works by repeated splitting the tree and each of the resulting subgraphs at the centroid, producing O(log n) layers of subgraphs.

Centroid Decomposition Illustration



Centroid Decomposition

```
bool r[MN];  // removed
int s[MN];   // subtree size
int dfs(int n, int p = 0) {
  s[n] = 1;
  for (int x : a[n])
    if (x != p && !r[x]) s[n] += dfs(x, n);
  return s[n];
}
```

Centroid Decomposition

```
int get_centroid(int n, int ms, int p = 0) {
// n = node, ms = size of tree, p = parent
    for (int x : a[n])
        if (x != p && !r[x])
            if (s[x] * 2 > ms)
                return get_centroid(x, ms, n);
    return n;
}
```

Centroid Decomposition

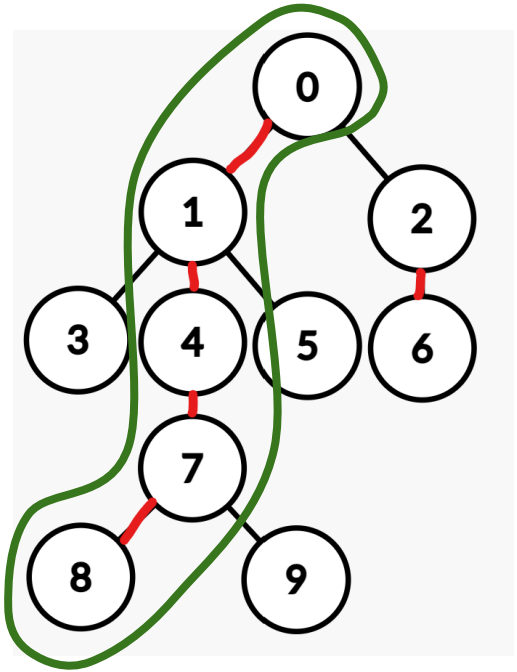
```
void centroid(int n = 1) {
  int C = get_centroid(n, dfs(n));

  // do something

  r[C] = 1;
  for (int x : a[C])
    if (!r[x]) centroid(x);
}
```

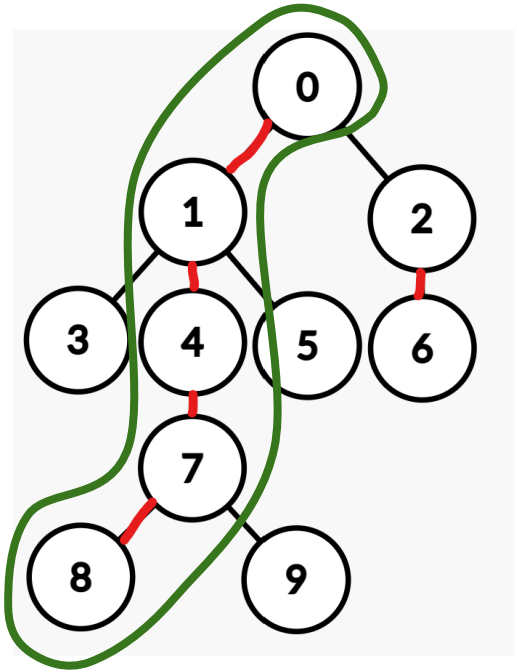
Heavy edge vs Light edge

- **Heavy edge**: the edge that connects a node to its maximum size subtree (break ties arbitrarily)
- **Light edge**: an edge that's not heavy



Heavy edge vs Light edge

- **Heavy edge**: the edge that connects a node to its maximum size subtree (break ties arbitrarily)
- **Light edge**: an edge that's not heavy
- Considering only heavy edges (highlighted in **red**), we obtain our heavy chains (circled in **green**)
- This gives us "groupings" of nodes on a tree but can we prove that this makes updates and queries efficient?



Heavy edge

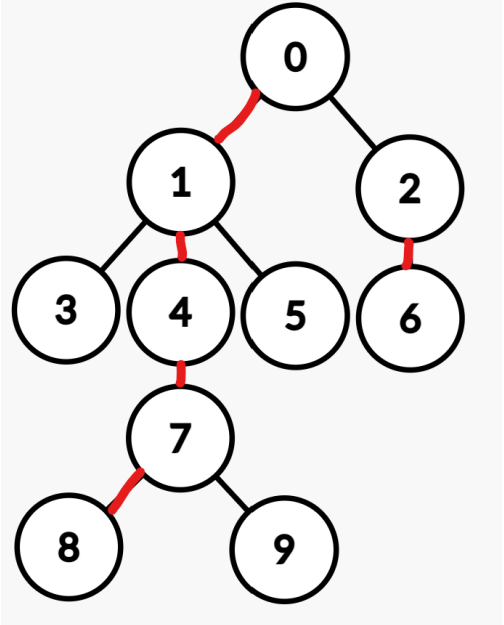
- Let G be a rooted tree of n vertices, with an arbitrary root.
- Let's calculate for each vertex v, the size of its subtree s(v), the number of vertices in the subtree of the vertex v including itself.

- Formally, we can an edge **heavy** if it leads to a vertex c such that:

$$s(c) \geq \frac{s(v)}{2} \iff \text{edge } (v, c) \text{ is heavy}$$

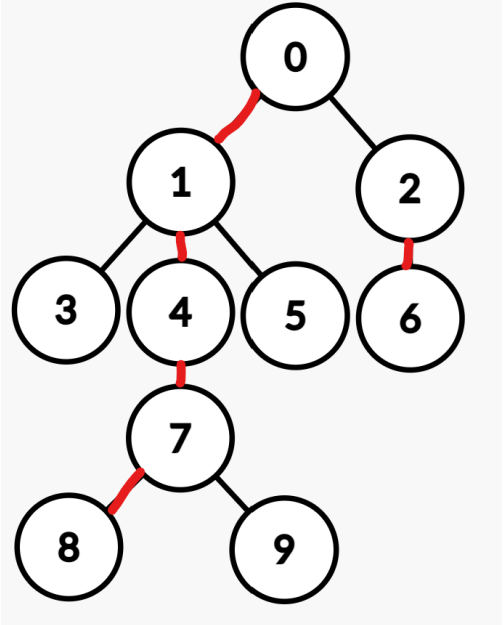
Heavy-Light Decomposition

- Suppose we want to process some query for vertices u and v.
- Without loss of generality, consider the path (u, lca(u, v)).
- Suppose we travel down the tree along the path, every new heavy chain means if we go through a light edge then subtree size is at least halved.
- Then there are O(log n) heavy chains on any path (u, v).



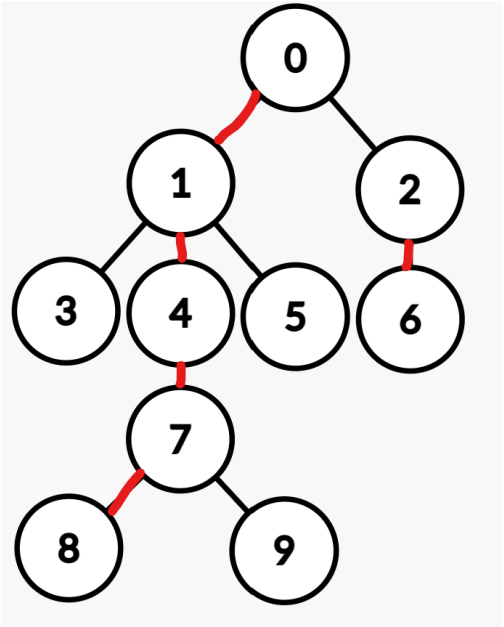
Heavy-Light Decomposition (con't)

- Heavy-light decomposition decomposes the tree into disjoint heavy chains so that every path consists of O(log n) chains.
- Each heavy chain is maintained by a segment tree for efficient range queries.
- Since the chains are disjoint, point updates take O(log n) time.
- Path queries take $O(\log^2 n)$ time.



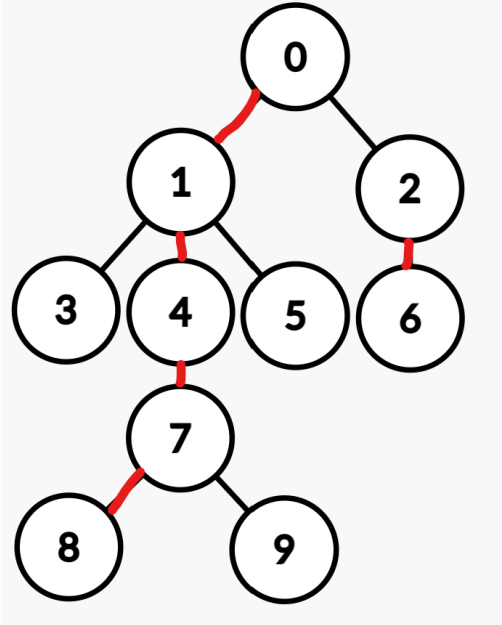
Update

- Suppose we want to update a node's value in the tree:
  - Straightforward **segment tree update**!
- Note that path updating would follow the same approach.



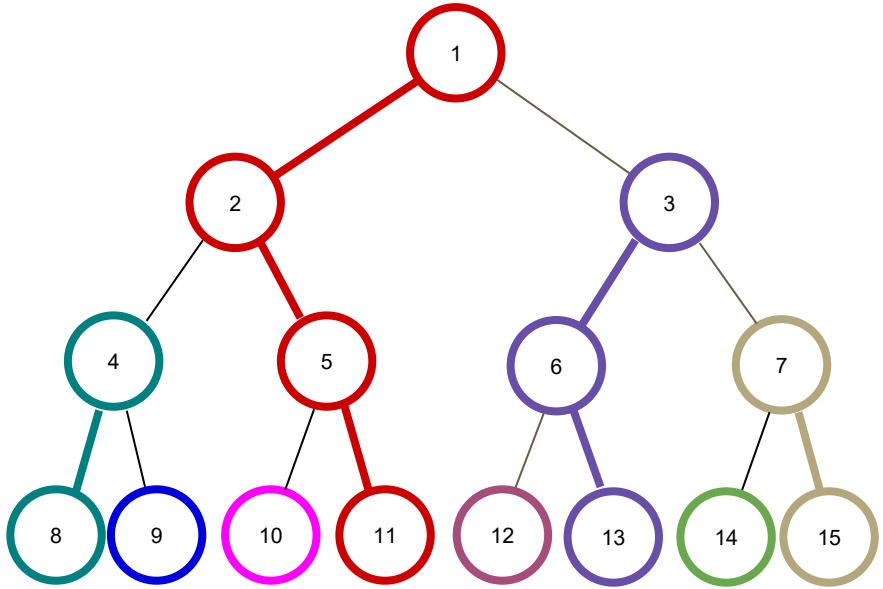
Query

- Suppose we want to compute the sum of a path (u, v).
- Similar to efficient LCA-finding, we start with the respective chains that nodes u and v are in.
- We repeatedly query for range sum in the lower of the two chains until we arrive at the chain containing the LCA, where we query a final time.



Step by Step Example

- Calculate sum along path between two nodes using HLD (values are same as node)
- Query: **Node 10** to **Node 14**
  - Current sum: 10
- Query: **Node 5** to **Node 14**
  - Current sum: 10 + 14
- Query: **Node 5** to **Node 7**
  - Current sum: 10 + 14 + 7
- Query: **Node 5** to **Node 3**
  - Current sum: 10 + 14 + 7 + 3
- Query: **Node 5** to **Node 1**
  - Current sum: 10 + 14 + 7 + 3 + (1 + 2 + 5)



Implementation Details

```
vector<int> parent, depth, heavy, head, pos;
int cur_pos;
```

Implementation Details

```
int dfs(int v, vector<vector<int>> const& adj) {
  int size = 1, max_c_size = 0;
  for (int c : adj[v]) {
    if (c != parent[v]) {
      parent[c] = v, depth[c] = depth[v] + 1;
      int c_size = dfs(c, adj);
      size += c_size;
      if (c_size > max_c_size)
        max_c_size = c_size, heavy[v] = c;
    }
  }
  return size;
}
```

Implementation Details

```
void decompose(int v, int h,
               vector<vector<int>> const& adj) {
  head[v] = h, pos[v] = cur_pos++;
  if (heavy[v] != -1)
    decompose(heavy[v], h, adj);
  for (int c : adj[v]) {
    if (c != parent[v] && c != heavy[v])
      decompose(c, c, adj);
  }
}
```

Implementation Details

```
void init(vector<vector<int>> const& adj) {
  int n = adj.size();
  parent = vector<int>(n);
  depth = vector<int>(n);
  heavy = vector<int>(n, -1);
  head = vector<int>(n);
  pos = vector<int>(n);
  cur_pos = 0;

  dfs(0, adj);
  decompose(0, 0, adj);
}
```

Maximum Query Example

```
int query(int a, int b) {
  int res = 0;
  for (; head[a] != head[b]; b = parent[head[b]]) {
    if (depth[head[a]] > depth[head[b]]) swap(a, b);
    int cur_heavy_path_max = segment_tree_query(pos[head[b]],
                                                pos[b]);
    res = max(res, cur_heavy_path_max);
  }
  if (depth[a] > depth[b]) swap(a, b);
  int last_heavy_path_max = segment_tree_query(pos[a], pos[b]);
  res = max(res, last_heavy_path_max);
  return res;
}
```

Example Uses

- Maximum value on the path between two vertices.
- Sum of the numbers on the path between two vertices.
- Repainting the edges of the path between two vertices.