1. **We typically use the preorder traversal to flatten the tree, because every node appears exactly once in the flattened version, unlike the Euler tour. However, the Euler tour traversal has its own unique features.**

Consider an unrooted tree. Let x be some node, and let E_x be an Euler tour when the tree is rooted at x. Note that the first and last elements of E_x are both x.

Now, let C_x be obtained from E_x by merging the first and last elements of E_x into a single element, thereby turning it into a circular array. We'll call C_x "a cyclic Euler tour of x".

Prove that the i-th cyclic shift of C_x corresponds to a cyclic Euler tour when the tree is rooted at $C_x[i]$. (In particular, for every node y, there is a cyclic shift of C_x that is a cyclic Euler tour when the tree is rooted at node y.

We hold this lemma: Let T be rooted at x with first child y. Remove edge (x,y), forming trees T_x and T_y. Create T' by adding T_x as the last subtree of y in T_y. Then

- T and T' are isomorphic as unrooted trees
- The cyclic Euler tour of T' is the 1st cyclic shift of the cyclic Euler tour of T.

Proof:

- Both trees are formed from T_x and T_y by adding edge (x,y) in different orientations.
- In Euler tours:
    - Let children of x in T_x be $x_1, \ldots, x_k$
    - Let children of y in T_y be $y_1, \ldots, y_m$
    - Let X, Y be cyclic Euler tours of T_x T_y
- In T rooted at x, Euler tour = [x] + Y + [y] + X + [x]
    - Cyclic Euler tour C_T = [x] + Y + [y] + X
- In T 'rooted at y, Euler tour = Y + [y] + X + [x] + [y]
    - Cyclic Euler tour $C_{T'}$ = Y + [y] + X + [x]
- Clearly, $C_{T'}$ is the first cyclic shift of C_T.

Theorem. For any node y, the ith cyclic shift of C_i where i is the position of y in C_x is a cyclic Euler tour rooted at y.

Proof: Apply Lemma 1 repeatedly i times. Each application performs one cyclic shift of the Euler tour and makes the next node the next root. After i shifts, $C_x[i]$ becomes the first element, hence the root.

2. **Consider the following problems:**

Problem 2. You are given an array with n elements [b_0, b_1, …,b_{n-1}]. Process q operations of the following two kinds online:

- **get(i): Return b_i.**
- **inc_range(i,j,v). Increment each of b_i, b_{i+1}, .., b_{j-1} by v.**

Problem 3. You are given an array wth n elements [a_0, a_1, …, a_{n-1}]. Process q operations of the following two kinds online:

- **sum_range(i,j). Find the sum $a_i + a_{i+1} + \ldots + a_{j-1}$.**
- **inc(i,v). Increment a_i by v.**

We say an algorithm that solves any of these problems runs in <f,g> if preprocessing has time complexity O(f), and each operation has time complexity O(g).

The following tasks show a sort of equivalence between Problems 2 and 3.

2.1. Suppose you have an <f,g> solution for Problem 2. Use it to find an <n + f,g> solution for Problem 3.

Let $s_i = a_0 + a_1 + \ldots + a_{i-1}$ be the ith prefix sum ($s_0 = 0$). Then,

- range sum: sum_range(i,j) = $s_j - s_i$
- point increment: inc(i,v) increases s_{i+1}, s_{i+2}, …, s_n by v, a range increment on the prefix sum array.

We preprocess in O(n+f) time by:

- computing prefix sums s_0, s_1, …, s_n in O(n) time using $s_i = s_{i-1} + a_{i-1}$
- initialize Problem 2 solution on array [s_0, s_1, …, s_n] in O(f) time.

Operations O(g) will each:

- sum_range(i,j): return get(j) - get(i)
- inc(i,v): perform inc_range(i+1, n+1, v) on the prefix sum array.

2.2. Suppose you have an <f,g> solution for Problem 3. Use it to find an <n+f, g> solution for Problem 2.

We decompose the range increment into suffix increments:

- inc_range(i,j,v) = inc_suffix(i,v) + inc_suffix(j, -v)
- Let c_k = total sum of all inc_suffix(k,v) operations performed so far.
- Current value: $b_i = b_i^{\text{initial}} + c_0 + c_1 + \ldots\ c_{i-1}$

- This is a range sum on the c array.

Preprocessing O(n+f):
- initialize array $[c_0, c_1,..., c_{n-1}]$ to all zeroes
- initialize Problem 3 solution on this array in O(f) time

Operations O(g) each:
- get(i): return $b_i^{initial}$ + sum_range(0,i)
- inc_suffix(i,v): perform inc(i,v) on the c array
- inc_range(i,j,v): perform inc_suffix(i,v) and inc_suffix(j, -v).

2.3. Take advantage of the previous results to find an <n, log n> solution for Problem 2.b

We can solve Problem 3 using a segment tree in <n, log n> time, therefore, we can use 2.2. to obtain an<n+n,logn> = <n, log n> solution to Problem 2.

3. **Consider the following problems:**

Problem 4. You are given a rooted tree with n nodes numbered 0 to n-1. Suppose that each node i contains the value a_i. Process q operations of the following two kinds online:
- **get_node(i). return a_i.**
- **inc_subtree(i,v). for each node j in the subtree rooted at node i, increment a_j by v.**

Problem 5. You are given an unrooted tree with n nodes numbered 0 to n-1. Suppose that each node i contains the value b_i. Process q operations of the following two kinds online:
- **sum_path(i,j). Find the sum of b_k across all nodes k in the path from node i to node j.**
- **inc_node(i,v). Increment b_i by v.**

The following tasks show a sort of equivalence between Problem 4 and Problem 5.

3.1. Suppose you have an <f,g> solution for Problem 4. Use it to find an <n+f, g> solution for Problem 5.

Root the tree arbitrarily at r = 0. Let sum_root(k) = sum_path(r,k), and let a_i = current value of sum_root(i).

For nodes i, j with LCA l:
sum_path(i,j) = sum_root(i) + sum_root(j) - 2 * sum_root(l) + value(l), which is O(1) sum_root queries assuming O(1) LCA lookup.

inc_node(i,v) increases sum_root(j) for every j in subtree of i, which is exactly a subtree increment on the a array.

Computing initial a_i = sum_root(i) using DFS = $a_{parent(i)} + b_i$ with $a_r = b_r$. We initialize Problem 4 solution on array $[a_0, a_1, \ldots, a_{n-1}]$ in O(f) time.

Operations O(g) each:

- sum_root(k): return get_node(k)
- sum_path(i,j): compute using formula above with O(1) LCA
- inc_node(i,v): perform inc_subtree(i,v) on the a array

3.2. Suppose you have an <f,g> solution for Problem 5. Use it to find an <n+f,g> solution for Problem 4.

Root the tree arbitrarily at r = 0. Let c_j be the total sum of all inc_subtree(j,v) operations performed so far. This is a path sum from r to i on the c array.

We preprocess in O(n + f) by initializing array $[c_0, c_1, \ldots, c_{n-1}]$ to all zeroes. Then, initialize Problem 5 solution on this array in O(f) time.

So, operations O(g) each:

- get_node(i): return $b_i^{initial}$ + sum_path(r,i)
- inc_subtree(i,v): perform inc_node(i,v) on the c array.

3.3. Find <n, log^{2}n> solutions to Problem 4 and Problem 5.

For Problem 5:
We root the tree arbitrarily at r = 0, and perform Heavy-Light Decomposition to decompose tree into O(log n) heavy paths. Each path maps to a contiguous segment in an array. This preprocessing takes O(n) time.

We can make operations run in O(log^{2}n) by maintaining a segment tree on the flattened array. Then,

- point increment: update single position in O(log n)
- path sum: decompose path into O(log n) heavy path segments, querying each in O(log n), resulting in O(log n) * O(log n) = O(log^{2}n).

For Problem 4 via 3.1:

We reduce the solution via reduction to Problem 5, applying the transformation in 3.1, and using the <n, $\log^2$n> solution for Problem 5.

3.4. Find <n, log n> solutions to Problem 4 and Problem 5.

For Problem 4:

Fist, we preprocess, performing DFS traversal to get the Euler tour order. Each subtree maps to a contiguous range [in[i], out[i]) in the flattened array. Then, we increment subtree at node i to increment range on [in[i], out[i]), point query at node i to point query at position in[i].

We now use the Problem 2.3 solution to handle range increment + point query in <n, log n>, and apply directly to the flattened array. Thus <n, log n>.

For Problem 5 using 3.1:

We apply the 3.1. transformation to get the <n+f,g> reduction. And with Problem 4 solved in <n,logn>, we have <n+n,logn> = <n, log n>,