1.1. Prove or Disprove: For every pair of nodes a and b (not necessarily distinct), node a is an ancestor of node b if and only if lca(a,b) = a. Note that a node a is considered an ancestor of itself.

[=>] If lca(a,b) = a, then a is the lowest common ancestor of a and b, then in particular, node a is an ancestor of node b.

[<=] If node a is an ancestor of node b, then a is a candidate for the lowest common ancestor, and since lca(a,b) is the actual lowest ancestor, it is either a or something "lower" than a, but anything lower than a is not an ancestor of a. Therefore, lca(a,b) = a.

1.2. Prove or Disprove: For every triple of nodes a,b, and c, which are not necessarily distinct, the set {lca(a,b), lca(b,c), lca(c,a)} has cardinality at most 2.

True. Let the three LCA values be sorted by depth such that depth(X) <= depth(Y) <= depth(Z) for {X,Y,Z} = {lca(a,b), lca(b,c), lca(c,a)}.

Observe that Y and Z are both ancestors of c since each pair is the LCA of a pair involving c, so Y is an ancestor of Z.

Since Z is an ancestor of a, Y is also an ancestor of a. Thus, Y is a common ancestor of a and b. Since X is the LCA of a and b, we have depth(X) <= depth(Y), but with depth(X) = depth(Y), and both being ancestors of b at the same depth, they must be the same node X = Y.

Hence, the set {lca(a,b), lca(b,c), lca(c,a)} contains at most two distinct nodes. This equality holds if lca(a,b) != lca(c,a), in which case the cardinality is 1.

Problem 1. There are n attractions in a city and m routes that connect these attractions. Each route connects two different attractions, is bidirectional, and takes a certain number of minutes to go through. There is exactly one way to go from any attraction to any other attraction via routes, if you're not allowed to reuse a route.

You want to plan a tour around this city, but you want to be as time-efficient as possible. Answer q questions of the following form:
"What is the shortest amount of time it will take to go from attraction A to attraction B?
Note that the problem is *online*, you must answer each query without knowing the future queries.

2.1. Find an algorithm that solves Problem 1 where preprocessing takes O(1) time, and each query takes O(n) time.

We observe that we can model the problem as a graph, a tree with unique paths.

For each query(A,B), we run:

1. BFS/DFS from A until reaching B, tracking the unique path.
2. Summing the edge weights along this path using DP.

```
function query(A,B):
    visited = array of size n, initialized to false
    parent = array of size n,initialized to -1
    cost_to = array of size n, initialized to 0

    queue = [A]
    visited[A] = true

    while queue not empty:
        u = queue.pop()
        if u == B:
            return cost_to[B]
        for each neighbor v of  with weight w:
            if not visited[v]:
                visited[v] = true
                parent[v] = u
                cost_to[v] = cost_to[u] + w
                queue.push(v)
```

The time and space complexity is O(n) because running DFS/BFS costs O(n) and it takes O(n) to initialize the required arrays.

2.1. Find an algorithm that solves Problem 1 where preprocessing takes O(n^2) time, and each query takes O(1) time.

For each node u:

- Run BFS/DFS from u to compute the shortest path distance to every other node.
- Use DP: dist[u][v] = dist[u][parent] + w(u, parent)
- Store all distances in an n x n table.

```
Pseudocode:
preprocess():
    dist = 2D array of size n x n
    for each node u in V:
        run BFS from u:
            dist[u][u] = 0
            for each node v visited:
                dist[u][v] = dist[u][parent[v]] + weight(parent[v], v)
    return dist

query(A,B,dist):
    return dist[A][B]
```

Preprocessing takes n * O(n) = O(n^2), with table lookups being O(1).

2.3. Find an algorithm that solves Problem 1 where preprocessing takes O(n log n) time, and each query takes O(log n) time.

We execute the following preprocessing:

1. Root the tree arbitrarily.
2. Compute depth and parent for each node via DFS: O(n)
3. Precompute binary lifting table up[u][k]: 2^kth ancestor of u: O(n log n)
4. Compute distance from root: dist[u] = dist[parent] + w(parent, u): O(n)

Query(A,B):

1. L = LCA(A,B) using binary lifting: O(log n)
2. Return dist[A] + dist[B] - 2 * dist[L]

~~~
```Pseudocode:
preprocess():
    DFS(root): compute parent, depth, dist[root] = -
    for k = 1 to log2(n)
    for each node u:
    up[u][k] = up[up[u][k-1]][k-1]

lca(u,v):
    if depth[u] < depth[v]: swap
    raise u to depth[v] using binary lifting
    if u == v: return u
    for k = log2(n) down to 0:
        if up[u][k] != up[v][k]:
            u = up[u][k]; v = up[v][k]
    return up[u][0]

query(A,B):
    L = lca(A,B)
    return dist[A] + dist[B] - 2 * dist[L]
~~~

Therefore, <O(nlogn), (logn)>.

Problem 2. There are n attractions in a city and m routes that connect these attractions. Each route connecs two different attractions, is bidirectional, and takes a certain number of minutes to go through. There is exactly one way to go from any attraction to any other attraction via routes, if you're not allowed to reuse a route.

You want to plan a tour around this city via electric car, but its battery only lasts for a limited number of minutes. There is a charging station at each attraction, where you can fully charge up your electric car's battery. Answer q questions of the following form:

"If you want to go from attraction A to attraction B, at least how many minutes should your battery be able to last?"

Note that the problem is online; you must answer each query without knowing the future queries.

3.1. Find an algorithm that solves Problem 2 that runs in <O(1), O(n)>.

First, we observe that the graph is a tree, and we can model it as an undirected weighted graph with a node for each attraction and an edge for each route, with cost equal to the amount of time to go through that route.

We do not preprocess anything, resulting in O(1) time complexity.

For query (A,B), we need the maximum edge weight along the unique path from A to B.

1. Run BFS/DFS from A until reaching B.
2. Track the maximum edge weight along the path via DP: max_to[v] = max(max_to[parent], w(parent,v))

```
Pseudocode:
function query(A,B):
    visited = array of size n, initialized to false
    max_to = array of size n, initialized to 0

    queue = [A]
    visited[A] = true

    while queue not empty:
        u = queue.pop()
        if u == B:
            return max_to[B]
        for each neighbor v of u with weight w:
            if not visited[v]:
                visited[v] = true
                max_to[v] = max(max_to[u], w)
                queue.push(v)
```

O(n) per query.

3.2. Find an algorithm that solves Problem 2 that runs in <$O(n^2)$, O(1)>.

This time, we preprocess for each node in V:

- Run BFS/DFS for u to compute the maximum edge weight along the path to every other node.
- Use DP: max_edge[u][v] = max(max_edge[u][parent], w(parent,v))
- Store all values in an n x n table.

```
preprocess():
    max_edge = 2D array of size n x n
    for each node u in V:
        run BFS from u:
            max_edge[u][u] = 0
            for each node v visited:
                p = parent[v]
                max_edge[u][v] = max(max_edge[u][p], weight(p,v))
    return max_edge

query(A,B,max_edge):
    return max_edge[A][B]
```

Preprocessing takes n * O(n) = $O(n^2)$, and queries take O(1) table lookup.

3.3. Find an algorithm that solves Problem 2 that runs in <O(nlogn), O(logn)>

Preprocessing goes as follows:

1. Root the tree arbitrarily. Run DFS to compute depth and parent.
2. Compute binary lifting tables: O(nlogn)
    a. up[u][k]: 2^k-th ancestor of u
    b. max[u][k]: maximum edge weight from u up to its 2^k-th ancestor
3. Recurrence:
    a. max[u][0] = w(u, parent)
    b. max[u][k] = max(max[u][k-1], max[up[u][k-1]][k-1])

Query(A,B):

1. Compute L = LCA(A,B) using binary lifting.
2. Compute maximum edge weight on path A -> L:
    a. Climb from A to L, tracking max using max table
3. Compute maximum on path B -> L similarly as 2

4. Return the maximum of both sides

```Pseudocode:
max_on_path(u,): // v is ancestor of u
    res = 0
    diff = depth[u] - depth[v]
    for k = log2(n) down to 0:
        if diff  >= $2^k$:
            res = max(res, max[u][k])
            u = up[u][k]
            diff -= $2^k$
    return res

query(A,B):
    L = lca(A,B)
    return max(max_on_path(A,L), max_on_path(B,L))
```

Preprocessing: O(n log n)
Querying: O(log n)