Q1 Solution Outline

Part (a): Algorithm Design

Goal: To compute the minimum total cost required to ensure that Planet 1 and Planet n become connected by adding at most two edges.

Step 1 — Identify Connected Components:
Represent the network as an undirected graph $G(V, E)$. Use either *Depth-First Search (DFS)* or a *Union–Find (Disjoint Set Union)* data structure to label each planet with its connected component number. Let `comp[i]` denote the component index of planet i.

$$\text{If } \texttt{comp[1]} = \texttt{comp[n]}, \text{ then cost} = 0.$$

Step 2 — Store Component Nodes:
After labeling, group planets belonging to the same component:

$$\text{components}[c] = \{\, i \mid \texttt{comp[i]} = c \,\}.$$

These component arrays are inherently sorted since the nodes are numbered from 1 to n.

Step 3 — Compute Minimum Distance Between Components:
For any two components A and B, define:

$$\text{dist}(A, B) = \min_{i \in A, j \in B} |i - j|, \quad \text{cost}(A, B) = (\text{dist}(A, B))^2.$$

To compute this efficiently:

- **Two-pointer method:** Traverse the sorted lists A and B simultaneously to find the minimal $|i - j|$. Complexity: $O(|A| + |B|)$.
- **Binary search method:** For each $i \in A$, find the nearest element in B using `lower_bound`. Complexity: $O(|A| \log |B|)$.

Step 4 — Evaluate Connection Options:
Let C_1 be the component containing Planet 1, and C_n be the component containing Planet n. Consider:

(i) **Direct connection (One edge):**

$$\text{ans}_1 = \text{cost}(C_1, C_n)$$

(ii) **Via intermediate component (Two edges):**

$$\text{ans}_2 = \min_k \Big(\text{cost}(C_1, C_k) + \text{cost}(C_k, C_n) \Big)$$

The final answer is:

$$\boxed{\text{Answer} = \min(\text{ans}_1, \text{ans}_2)}.$$

Part (b): Implementation - Two-Pointer Approach

The following pseudocode outlines the steps using the two-pointer method for the distance calculation in Step 4. It uses 0-based indexing for implementation clarity (i.e., nodes 0 to $N-1$ instead of 1 to n). The flow is as follows:

- Identifying connected components using DFS/BFS
- Compute distances to start component
- Compute distances to end component
- Compute minimum sum of squared distances

Part (b'): Alternate Implementation - Binary Search Approach

An alternate pseudocode using binary search within a dedicated COST function to find the minimum distance between two specific components, as suggested by the alternative C++ implementation.

Data Structure Justification

- `vector<vector<int>> adj`: Represents the adjacency list of the graph. Efficient for storing sparse graphs and iterating over neighbors.
- `vector<int> vis`: Stores the component label for each node. Initialized to -1 to indicate unvisited nodes.
- `stack<int> s`: Used for iterative depth-first search (DFS) to avoid recursion and stack overflow.
- `vector<vector<int>> components`: Groups nodes by their connected component label for later distance calculations.
- `vector<int> dist1, dist2`: Stores minimum distances from each component to the start and end components respectively. Initialized with `INT_MAX` to simulate infinity.
- `vector<int> adj[MAX_N]`: Static adjacency list for each node. Avoids dynamic allocation overhead and is suitable for large input sizes.
- `vector<int> comps[MAX_N]`: Stores nodes belonging to each connected component. Maintains sorted order for efficient binary search.
- `int comp[MAX_N]`: Maps each node to its component index. Initialized to -1 to mark unvisited nodes.

Correctness:

- **Optimal Substructure:** The minimal cost between any two components depends only on the minimal distance between their node indices, since cost grows monotonically with $|i-j|$.

Algorithm 1 Minimum Squared Distance Between Components

Input: Integers n, m
Initialize adjacency list `adj` of size n
for $i = 0$ to $m - 1$ **do**
    Read u, v
    Decrement u, v by 1
    Add v to `adj[`u`]`, and u to `adj[`v`]`
Initialize `vis` of size n with -1
$num\_components \leftarrow 0$
for $i = 0$ to $n - 1$ **do**
    if `vis[`i`]` $\neq -1$ **then**
        continue
    Initialize stack s, push i
    while s is not empty **do**
        $u \leftarrow s$.`top()`
        s.`pop()`
        if `vis[`u`]` $\neq -1$ **then**
            continue
        `vis[`u`]` $\leftarrow num\_components$
        for each v in `adj[`u`]` **do**
            if `vis[`v`]` $= -1$ **then**
                s.`push`(v)
    $num\_components \leftarrow num\_components + 1$
Initialize `components` as list of lists of size $num\_components$
for $i = 0$ to $n - 1$ **do**
    Append i to `components[vis[`i`]]`
`start_component` $\leftarrow$ `components[vis[0]]`
`end_component` $\leftarrow$ `components[vis[`$n - 1$`]]`
Initialize `dist1`, `dist2` of size $num\_components$ with ∞
$a \leftarrow 0$
for $i = 0$ to $n - 1$ **do**
    $dist \leftarrow |start\_component[a] - i|$
    while $a < |start\_component| - 1$ and $|start\_component[a + 1] - i| < dist$ **do**
        $a \leftarrow a + 1$
        $dist \leftarrow |start\_component[a] - i|$
    `dist1[vis[`i`]]` $\leftarrow \min($`dist1[vis[`i`]]`$, dist)$
$b \leftarrow 0$
for $i = 0$ to $n - 1$ **do**
    $dist \leftarrow |end\_component[b] - i|$
    while $b < |end\_component| - 1$ and $|end\_component[b + 1] - i| < dist$ **do**
        $b \leftarrow b + 1$
        $dist \leftarrow |end\_component[b] - i|$
    `dist2[vis[`i`]]` $\leftarrow \min($`dist2[vis[`i`]]`$, dist)$
$res \leftarrow \infty$
for $i = 0$ to $num\_components - 1$ **do**
    $res \leftarrow \min(res, dist1[i]^2 + dist2[i]^2)$
Output: res

Algorithm 2 Minimum Cost to Connect Farm Components

Input: Integers n, m
Initialize adjacency list `adj[0..n-1]`
Initialize component list `comps[0..n-1]`
Initialize array `comp[0..n-1]` with -1
function DFS(cur, c)
    if $comp[cur] \neq -1$ **then**
        return
    $comp[cur] \leftarrow c$
    for each u in $adj[cur]$ **do**
        DFS(u, c)
function Cost(a, b)
    $dist \leftarrow MAX\_N$
    for each u in $comps[a]$ **do**
        $i \leftarrow$ lower bound index of u in $comps[b]$
        if $i > 0$ **then**
            $dist \leftarrow \min(dist, |comps[b][i-1] - u|)$
        if $i < |comps[b]|$ **then**
            $dist \leftarrow \min(dist, |comps[b][i] - u|)$
    return $dist^2$
function Solve
    Read n, m
    for $i = 0$ to $n - 1$ **do**
        $comp[i] \leftarrow -1$
        $adj[i]$.`clear()`
        $comps[i]$.`clear()`
    for $i = 0$ to $m - 1$ **do**
        Read a, b
        Decrement a, b by 1
        Add b to $adj[a]$, and a to $adj[b]$
    $cur \leftarrow -1$
    for $i = 0$ to $n - 1$ **do**
        if $comp[i] = -1$ **then**
            DFS(i, $++cur$)
    for $i = 0$ to $n - 1$ **do**
        Append i to $comps[comp[i]]$
    $res \leftarrow$ Cost($comp[0]$, $comp[n-1]$)
    for $c = 1$ to $cur - 1$ **do**
        $res \leftarrow \min(res,$ Cost(c,comp[0]) + Cost(c,comp[n-1]))
    Output: res

- **Completeness:** The algorithm examines all possibilities involving either one or two new edges, ensuring that every valid configuration is considered.
- **Greedy Minimality:** Adding more than two edges cannot yield a smaller cost, as each new edge adds a non-negative cost and connectivity is already achieved with at most two.

Thus, the algorithm always finds the globally minimal total cost.

Part (c):Time and Space Complexity

Time Complexity Analysis:

Let $N = n$ (number of planets) and $M = m$ (number of existing connections). K is the number of connected components.

Step	Description	Two-Pointer Time	Binary Search
1	Finding connected components (DFS/Union–Find)	$O(N + M)$	$O(N + M)$
2	Collecting component nodes	$O(N)$	$O(N)$
3	Distance calculation:		
	Total work in CALCULATE_MIN_DIST_ARRAY	$O(N)$	N/A
	Total work in COST calls	N/A	$O(\sum_A \lvert A\rvert \log N)$
4	Final evaluation:	$O(K)$	$O(K \cdot \text{COST})$

Note on Binary Search Complexity: The total time complexity for the Binary Search approach is dominated by Step 3 and Step 4's COST function calls, which could potentially reach $O(N \cdot K \cdot \log N)$ in the worst case, making the **Two-Pointer approach** ($O(N + M)$) significantly faster.

Total Time per Test Case: $O(N + M)$ (For Two-Pointer Approach).

Space Complexity: $O(N + M)$ due to adjacency lists and component storage.

Key Insights:

- Graph connectivity reduces to analyzing component indices along a number line (1 to n).
- The minimal squared distance between components determines the lowest connection cost.
- At most two edges are sufficient to guarantee optimal connectivity.

Common Mistakes:

- Inefficient pseudo code with code's time complexity is $O(N^2)$, not $O(N+M)$. Using 2-pointer, iterating through components and pre-computing minimum distances to component 1 and component n separately will give $O(m + n)$ time complexity (or) can be done in $O(n.logn)$ time by using lower bound (i.e., binary search on components array).

- Not showing complete implementation i.e., for the last (final) loop in you code. (which is crucial for computing time complexity)
- Giving incorrect algorithm i.e., logic to compute cost is wrong ; Checking only for endpoint indices is not enough.

Q2 Solution Outline

Part (a)

Design an efficient algorithm to decide whether a valid royal postal route exists in the kingdom of Valoria. If it exists, also provide such a route. Explain your approach.

Solution: This problem is well known as finding an Eulerian path/ealk. Conditions required for a valid postal route, i.e., an eulerian path to exist and have different start and end points is -

- Graph should be connected (ignoring the isolated vertices).
- Exactly two nodes should have and odd degree.

No need to provide proof of correctness.

Constructing the Eulerian walk : There are two approaches, (i) using a recursive dfs-like approach, (ii) Heirhozler's Algorithm which simulates a dfs iteratively using a stack

```
DFS(u){
  while adj[u] is not empty{
    take an edge (u,v) from adj[u];
    remove (u,v) and (v,u) from the adj[u] and adj[v] respectively;
    DFS(v);
  }
  path.append(u);
}
// Call DFS(start), where start is one of the odd-degree vertices.
// path contains the Eulerian walk in the reverse order
```

We need to remove the edges from the graph in O(1) time, there can be multiple ways to do so. Removing (u,v) from adj[u] in O(1) time is trivial, we simply maintain a tail pointer of the adjacency list, choos the last element then remove the last element. For (v,u) one way is to keep a pointer to the reverse edge in each node of the linked list, and use the pointer in (u,v) to delete (v,u) in O(1) time. Another solution is to keep a boolean array to mark the edges as used, and then lazily remove the (v,u) edge when we encounter it.

Grading scheme for part(a) : 5 marks for conditions. 10 marks for path construction algorithm.

Part (b)

The royal messengers have now requested you to find a route that starts and ends in the same town. Mention the changes you need to make in your algorithm in part (a).

Solution: There are two changes - (i) All vertices must be even and (ii) We can start from any vertex.

Part (c)

Analyze the time and space complexity of your algorithm for both the parts (a) and (b).

Solution: Since each edge will be traversed exactly once, path construction will take O(m) time since each edge is traversed exactly once. The checks for connectivity can be done using standard BFS/DFS, will take O(n+m) time. Degree counting will take O(m) time.
Time complexity is O(n+m).
We need O(n) space for counting degrees, O(m) space for storing the path. O(n+m) for adjacency list, O(m) for recursion stack.
Space Complexity is O(n+m).

Some common mistakes

- If no explanation provided for why the path construction is O(m) time, 1 mark is deducted.
- Just writing the time and space complexities without showing the analysis is given partial marks only.
- In some solutions, pseudocode for part (a) accepts both, odd degree count = 0 and 2. This is incorrect. Only partial marks are given for this.
- Edges are not removed from the graph. Note that only keeping a boolean array to mark the edges is not sufficient, since next time you would need to iterate over all the neighbours in the adjacency list to find an unmarked/unused edge. This will not be O(1) time and our argument that each edge is operated on only once will not be correct. Some solutions iterate over the adjacency list for removing the edge. That is also sub optimal. 1 mark is deducted for this.
- Some solutions have only mentioned the name of Heirhozler's Algorithm, but not given any implementation approach or pseudocode have received zero credit for path construction part.
- Some solutions have vaguely explained the Heirhozler's algorithm without pseudocode. Such solutions have received partial credits. If not providing a pseudocode, it must be clearly mentioned, when a vertex must be popped from the stack, when it should be pushed onto the stack, and when it is added to the path. If all these are clearly mentioned, full marks are awarded even if pseudocode is not provided. It is not correct to vaguely mention phrases like "... we will backtrack and (*do something*)..." or "we can then merge these partial paths ..." . You must explicitly mention, what how do you backtrack, till what point do you backtrack, how do you decide when to backtrack, how do you merge the partial paths and how do you do

all this in O(m). Solutions which have mentioned these points have received full credit.

Q3 Solution Outline

(a) Problem Formulation (5 points)

We model the system as a **bipartite graph** $G = (V, E)$ consisting of:

- Island nodes $I = \{1, 2, \ldots, m\}$.
- Tram (route) nodes $R = \{r_1, r_2, \ldots, r_n\}$.

An edge (i, r) exists if tram r visits island i. The messenger may move between an island and a tram only through such edges. Hence, travel from source island s to target island t corresponds to finding the **shortest alternating path** in this bipartite graph. The number of tram transitions on that path gives the minimum trams required.

(b) Algorithm Design (10 points)

We perform a **BFS** starting from the source island, maintaining the minimum number of trams used to reach each island:

- Build a mapping `islandToRoutes`, storing for each island the list of trams visiting it.
- Initialize a distance array `dist[]` for all islands with ∞, and set `dist[source] = 0`.
- Maintain a queue of islands and a set of visited trams.
- When processing island u:
  - For each tram r serving u, if not visited:
    - Mark r visited.
    - For every island v on tram r:
      - If $dist[v] = \infty$, set $dist[v] = dist[u] + 1$ and enqueue v.
- The first time we reach the target island, $dist[target]$ gives the minimum number of trams required.

(c) Pseudocode (10 points)

Algorithm 3 Minimum Trams to Reach Target (BFS with Distance Array)

```
Input: routes, source, target
if source = target then return 0
Build map islandToRoutes
for each route r do
    for each island i in route r do
        islandToRoutes[i].append(r)
Initialize array dist[1..m] ← ∞
Initialize queue Q, set visitedTrams
dist[source] ← 0; Enqueue(source)
while Q not empty do
    u ← Q.pop()
    for each tram r in islandToRoutes[u] do
        if r not in visitedTrams then
            visitedTrams.insert(r)
            for each island v in routes[r] do
                if dist[v] = ∞ then
                    dist[v] ← dist[u] + 1
                    if v = target then return dist[v]
                    Enqueue(v)
return −1
```

(d) Complexity Analysis (5 points)

Let n be the number of trams and k the maximum number of islands per tram.

- **Time Complexity:** $O(n \cdot k)$ — each (island, tram) connection is processed once.
- **Space Complexity:** $O(n \cdot k)$ — for the adjacency mapping and distance array.