Homework 2

October 9, 2025, Due October 15, 2025

Question 1

The algorithm described in lecture for computing a topological ordering of a DAG repeatedly finds a node with no incoming edges and deletes it. This will eventually produce a topological ordering, provided that the input graph really is a DAG.

But suppose that we're given an arbitrary graph that may or may not be a DAG. Extend the topological ordering algorithm so that, given an input directed graph G, it outputs one of two things: (a) a topological ordering, thus establishing that G is a DAG; or (b) a cycle in G, thus establishing that G is not a DAG. The running time of your algorithm should be $\mathcal{O}(n+e)$ for a directed graph with n nodes and e edges.

Solution: We can extend the topological-ordering procedure so that it either returns a valid topological ordering (if the graph is a DAG) or explicitly outputs a directed cycle (if it is not). The algorithm is based on Kahn's method.

- **Step 1 (Compute in-degrees):** For every vertex v, compute the number of incoming edges indeg[v].
- **Step 2 (Initialize queue):** Place all vertices with indeg[v] $= 0$ into a queue Q. Initialize an empty list `order`.
- **Step 3 (Kahn's loop):** While Q is not empty:
  1. Remove v from Q and append it to `order`, also delete v from the original graph.
  2. For each edge $v \to w$: decrement indeg[w], If it becomes 0, add w to Q. Also delete each $v \to w$ from the original graph.

- **Step 4 (If DAG):** If $|\texttt{order}| = n$, output `order` as the topological ordering and stop.
- **Step 5 (If not DAG):** Let U be the set of vertices that were *never added to the queue* during Kahn's algorithm (equivalently, vertices that never had in-degree 0). These vertices form a subgraph(equivalently, the remaining graph after the deletion in Step 3) in which every node has at least one incoming edge, so a directed cycle must exist within this subgraph.

  To locate a specific cycle, arbitrarily pick a vertex to perform a depth-first search (DFS) on the subgraph induced by U using three colors:

  - **WHITE:** vertex not yet visited,
  - **GRAY:** vertex currently in recursion stack (active path),
  - **BLACK:** vertex fully explored.

  When the DFS encounters an edge $u \to w$ where w is GRAY, a back edge has been detected—indicating a cycle. The cycle can then be reconstructed by tracing parent pointers from u back to w, yielding an explicit directed cycle in G.

Runtime Analysis: Each vertex and edge is processed at most once in the Kahn phase, giving $O(n+e)$. The DFS in the second phase also visits each remaining vertex and edge once, also $O(n+e)$. Thus the overall time complexity is:

$$T(n, e) = O(n + e)$$

Proof of Correctness:

- If G is a DAG: Kahn's algorithm always removes at least one vertex with in-degree 0 in each iteration. After processing all vertices, `order` contains all n vertices, forming a valid topological ordering.
- If G is not a DAG: Kahn's algorithm terminates with a nonempty set U of vertices that were never added to the queue. In the subgraph induced by U, every vertex has in-degree at least 1.

  Why this implies a cycle (proof by contradiction): Suppose, for contradiction, that the subgraph on U is acyclic. Then, because it is a DAG, it must contain at least one vertex v with in-degree 0 (this follows from the property that every DAG has at least one source vertex). However, such a vertex v would have been added to the queue during Kahn's algorithm when its in-degree became 0, contradicting the definition of U as the set of vertices never added to the queue. Therefore, our assumption is false, and the subgraph on U must contain at least one directed cycle.

Once this is established, the DFS restricted to U will necessarily find a back edge and reconstruct an explicit cycle.

Why DFS on U finds a back edge: Since every vertex in U has in-degree at least 1, the subgraph $G[U]$ must contain a directed cycle. When DFS is run on $G[U]$, it explores edges along some cycle $v_0 \to v_1 \to \cdots \to v_k \to v_0$. Let v_i be the first vertex of this cycle discovered by DFS. When DFS later visits v_{i-1} and examines the edge $v_{i-1} \to v_i$, v_i is still on the recursion stack (GRAY), so this edge is identified as a *back edge*. Following parent pointers from v_{i-1} back to v_i reconstructs an explicit directed cycle in $G[U]$.

- Since exactly one of these two cases occurs, the algorithm always outputs either:
  - a valid topological ordering (if G is acyclic), or
  - a concrete directed cycle (if G contains one).

  Therefore, the algorithm is correct.

Question 2

Inspired by the example of that great Cornellian, Vladimir Nabokov, some of your friends have become amateur lepidopterists (they study butterflies). Often when they return from a trip with specimens of butterflies, it is very difficult for them to tell how many distinct species they've caught—thanks to the fact that many species look very similar to one another.

One day they return with n butterflies, and they believe that each belongs to one of two different species, which we'll call A and B for purposes of this discussion. They'd like to divide the n specimens into two groups—those that belong to A and those that belong to B—but it's very hard for them to directly label any one specimen. So they decide to adopt the following approach.

For each pair of specimens i and j, they study them carefully side by side. If they're confident enough in their judgment, then they label the pair (i, j) either *same* (meaning they believe both come from the same species) or *different* (meaning they believe they come from different species). They also have the option of rendering no judgment on a given pair, in which case we'll call the pair *ambiguous*.

So now they have the collection of n specimens, as well as a collection of m judgments (either "same" or "different") for the pairs that were

not declared ambiguous. They'd like to know if this data is consistent with the idea that each butterfly is from one of species A or B. More concretely, we'll declare the m judgments to be *consistent* if it is possible to label each specimen either A or B so that:

- **For each pair (i, j) labeled "same," specimens i and j receive the same label.**
- **For each pair (i, j) labeled "different," specimens i and j receive different labels.**

They're in the middle of tediously checking whether their judgments are consistent, when one of them realizes that you probably have an algorithm that could answer this question right away.

Task: Give an algorithm with running time $O(m + n)$ that determines whether the m judgments are consistent.

Solution. Interpret the n specimens as vertices $\{1, \ldots, n\}$. Each "same" judgment is a constraint $i \equiv j$ (same species), and each "different" judgment is $i \not\equiv j$ (different species). Ambiguous pairs are ignored.

We first collapse all vertices that must be equal (by "same" constraints) into *equivalence classes*; then we check whether the remaining "different" constraints between classes form a *bipartite* graph. If they do, assign one bipartition to species A and the other to B.

Algorithm.

1. **Group by "same" judgments.** Create n initially separate groups, one for each specimen. For every judgment (i, j) labeled *same*, merge the groups containing i and j into a single group. After this step, every group represents specimens that are believed to come from the same species.
2. **Handle all "different" judgments and build the relationship graph.** After grouping specimens that were judged "same" into groups (as in Step 1), process each judgment (i, j) labeled *different* as follows:
   - If specimens i and j already belong to the same group, the data are immediately Inconsistent, since two specimens said to be "different" would be into the different groups.
   - Otherwise, let G_i and G_j be the two distinct groups that contain specimens i and j. Draw an undirected edge between G_i and G_j in a new graph H, which records all "different" relationships between groups. Each edge in H means that its two endpoints must belong to different species.

3. **Check whether the graph is bipartite.** Traverse each connected component of H using either breadth-first search (BFS) or depth-first search (DFS), trying to color its nodes with two colors (say, 0 and 1) so that every edge connects nodes of different colors. If an edge ever connects two nodes of the same color, the judgments are Inconsistent, since that would require two "different" groups to belong to the same species. If all components can be colored successfully, the graph is bipartite and the judgments are Consistent. Assign one color to species A and the other to species B, and label every specimen according to the color of its group.

Runtime Analysis. In Step 1, we process all "same" judgments by repeatedly merging the groups that should belong together. Using an efficient grouping structure, each merge and lookup takes almost constant time on average, so this phase runs in total $O(n + m)$ time.

In Step 2, we examine each "different" judgment exactly once. For each such pair (i, j), we either detect a contradiction (if i and j are already in the same group) or add one undirected edge between their groups in the relationship graph H. This step therefore also runs in $O(m)$ time.

In Step 3, we perform a bipartite check on H using either breadth-first or depth-first search. Each vertex and edge of H is visited at most once, taking $O(n + m)$ time.

Hence, all steps together run in overall linear time $O(n + m)$, and the space usage is also linear in the number of specimens and judgments.

Proof of Correctness.

- If the algorithm reports Consistent, it has successfully assigned two colors (representing species A and B) to all groups in H such that every edge connects nodes of opposite colors. Each "same" pair lies within a single group and thus shares the same color, and each "different" pair corresponds to an edge between nodes with opposite colors. Therefore, the resulting labeling of specimens satisfies all judgments.
- Suppose there exists a consistent way to label all specimens as species A or B. Then all "same" pairs belong to the same group, and all "different" pairs connect groups with opposite labels. This labeling induces a valid two-coloring of H, meaning the bipartite check will succeed and the algorithm will output Consistent.
- If H were not bipartite, it would contain an odd cycle of "different" edges. Following this cycle alternates the assigned colors A and B, but returning to the starting point would require assigning both colors to the same node—a contradiction. Hence, a consistent labeling exists if and only if the relationship graph H is bipartite.

Question 3

A number of stories in the press about the structure of the Internet and the Web have focused on some version of the following question: *How far apart are typical nodes in these networks?* If you read these stories carefully, you find that many of them are confused about the difference between the *diameter* of a network and the *average distance* in a network; they often jump back and forth between these concepts as though they were the same thing.

As in the text, we say that the *distance* between two nodes u and v in a graph $G = (V, E)$ is the minimum number of edges in a path joining them; we'll denote this by $\mathrm{Dist}(u, v)$. We say that the *diameter* of G is the maximum distance between any pair of nodes, and we'll denote this quantity by $\mathrm{Diam}(G)$. Let's define a related quantity, which we'll call the *average pairwise distance* in G, denoted $\mathrm{APD}(G)$. We define $\mathrm{APD}(G)$ to be the average, over all $\binom{n}{2}$ sets of distinct nodes u and v, of the distance between u and v. That is,

$$\mathrm{APD}(G) = \left[\sum_{\{u,v\} \subseteq V} \mathrm{Dist}(u, v) \right] / \binom{n}{2}.$$

Here's a simple example to convince yourself that there are graphs G for which $\mathrm{Diam}(G) \neq \mathrm{APD}(G)$. Let G be a graph with three nodes u, v, w, and with the two edges $\{u, v\}$ and $\{v, w\}$. Then

$$\mathrm{Diam}(G) = \mathrm{Dist}(u, w) = 2,$$

while

$$\mathrm{APD}(G) = \frac{\mathrm{Dist}(u, v) + \mathrm{Dist}(u, w) + \mathrm{Dist}(v, w)}{3} = \frac{4}{3}.$$

Of course, these two numbers aren't all that far apart in the case of this three-node graph, and so it's natural to ask whether there's always a close relation between them. Here's a claim that tries to make this precise:

Claim: There exists a positive natural number c so that for all connected graphs G, it is the case that

$$\frac{\mathrm{Diam}(G)}{\mathrm{APD}(G)} \leq c.$$

Decide whether you think the claim is true or false, and give a proof of either the claim or its negation.

K_m (clique: every node connected to every node)

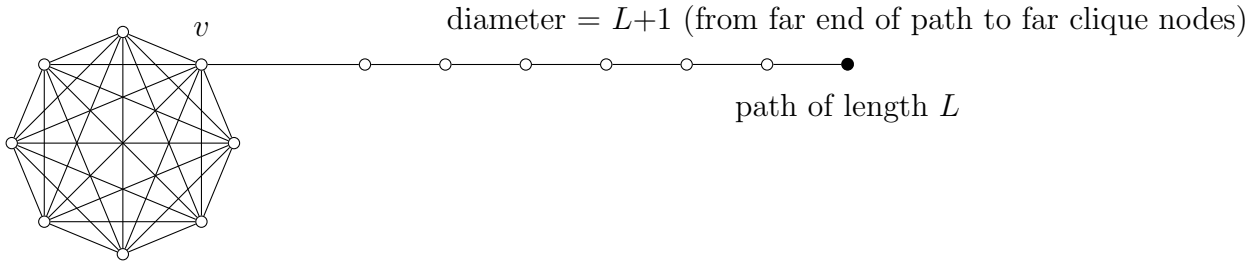

Figure 1: **Figure for the proof (negation of the claim).** For $m = L^3$ and $L \to \infty$, the graph $G_{m,L}$ has $\text{APD}(G_{m,L}) \to 1$ and $\text{Diam}(G_{m,L}) \geq L$. Thus $\text{Diam}(G_{m,L})/\text{APD}(G_{m,L}) \to \infty$, so no universal constant c can bound the ratio for all connected graphs.

Proof (negation of the claim) We show that there is **no universal constant** c such that

$$\frac{\text{Diam}(G)}{\text{APD}(G)} \leq c$$

for all connected graphs G.

Construction: Consider the *lollipop graph* $G_{m,L}$ formed by attaching a complete graph (clique) K_m to a path of length L at one vertex v, as illustrated in Figure 1. Thus, $G_{m,L}$ consists of:

- a large clique K_m where every pair of nodes is connected (distance 1);
- a simple path $x_1, x_2, \ldots, x_L$ attached to K_m via the vertex v.

Diameter: The farthest pair of nodes is between the endpoint x_L of the path and any non-attached clique vertex. Hence

$$\text{Diam}(G_{m,L}) = L + 1.$$

Average Pairwise Distance: For $m \gg L^3$, almost all pairs of vertices lie inside the clique and have distance 1. Pairs that cross between the clique and the path contribute an $O(1)$ average distance, since each path vertex connects quickly through v. Thus,

$$\text{APD}(G_{m,L}) \approx 1.$$

Conclusion: As $L \to \infty$, we have

$$\frac{\text{Diam}(G_{m,L})}{\text{APD}(G_{m,L})} \approx L + 1 \ \to \ \infty.$$

Question 4

You're helping some security analysts monitor a collection of networked computers, tracking the spread of an online virus. There are n computers in the system, labeled $C_1, C_2, \ldots, C_n$, and as input you're given a collection of *trace data* indicating the times at which pairs of computers communicated. Thus the data is a sequence of ordered triples (C_i, C_j, t_k); such a triple indicates that C_i and C_j exchanged bits at time t_k. There are m triples total.

We'll assume that the triples are presented to you in sorted order of time. For purposes of simplicity, we'll assume that each pair of computers communicates at most once during the interval you're observing.

The security analysts you're working with would like to be able to answer questions of the following form:

> **If the virus was inserted into computer C_a at time x, could it possibly have infected computer C_b by time y?**

The mechanics of infection are simple: if an infected computer C_i communicates with an uninfected computer C_j at time t_k (in other words, if one of the triples (C_i, C_j, t_k) or (C_j, C_i, t_k) appears in the trace data), then computer C_j becomes infected as well, starting at time t_k. Infection can thus spread from one machine to another across a sequence of communications, provided that no step in this sequence involves a move backward in time.

Thus, for example, if C_i is infected by time t_k, and the trace data contains triples (C_i, C_j, t_k) and (C_j, C_q, t_r) where $t_k \leq t_r$, then C_q will become infected via C_j. (Note that it is okay for $t_k = t_r$; this would mean that C_j had open connections to both C_i and C_q at the same time, and so a virus could move from C_i to C_q.)

Example 1. Suppose $n = 4$, and the trace data consists of the triples

$$(C_1, C_2, 4), \quad (C_2, C_4, 8), \quad (C_3, C_4, 8), \quad (C_1, C_4, 12).$$

If the virus was inserted into computer C_1 at time 2, then C_3 would be infected at time 8 by a sequence of three steps: first C_2 becomes infected at time 4, then C_4 gets the virus from C_2 at time 8, and then C_3 gets the virus from C_4 at time 8.

Example 2. If the trace data instead were

$$(C_2, C_3, 8), \quad (C_1, C_4, 12), \quad (C_1, C_2, 14),$$

and again the virus was inserted into computer C_1 at time 2, then C_3 would not become infected during the period of observation: although C_2 becomes infected at time 14, we see that C_3 only communicates with C_2 before C_2 was infected. There is no sequence of communications moving forward in time by which the virus could get from C_1 to C_3 in this second example.

Goal: Design an algorithm that answers questions of this type: given a collection of trace data, the algorithm should decide whether a virus introduced at computer C_a at time x could have infected computer C_b by time y. The algorithm should run in time $\mathcal{O}(m+n)$.

Solution.

Graph Construction: Create a directed graph G, with nodes representing each computer at an instant in time when it is communicating (that is, for every triple (C_x, C_y, t), create a node for C_x and C_y at time t). If we have m time-stamped events, this produces $2m$ nodes. Now create two types of directed edges:

- **Communication edges:** For each triple (C_x, C_y, t), add a directed edge $(C_x, t) \to (C_y, t)$ and another edge $(C_y, t) \to (C_x, t)$. These represent the fact that infection can spread between the two computers at that same time. Hence, each triple creates two communication edges.
- **Forward-in-time edges:** For each computer C_x, if it appears at times $t_1 < t_2 < \cdots < t_k$ in the trace data, add directed edges $(C_x, t_1) \to (C_x, t_2), (C_x, t_2) \to (C_x, t_3), \ldots, (C_x, t_{k-1}) \to (C_x, t_k)$. These edges represent that once a computer is infected, it remains infected in later communications.

This construction creates at most $2m$ communication edges (two per triple) and at most $2m$ additional forward-in-time edges (one per node). Therefore, the total number of nodes and edges are both $O(m)$.

Algorithm: Perform a BFS (Breadth-First Search) in G starting from the node (C_a, t) where $t \geq x$ is the first time computer C_a appears after the infection is introduced. If

during BFS we reach any node (C_b, t') such that $t' \leq y$, output **Yes**; otherwise output **No**. Since BFS runs in $O(|V| + |E|)$ time, and our graph has $O(m)$ nodes and edges, the total running time is $O(m + n)$.

Proof of Correctness: To prove correctness, we must show that BFS outputs **Yes** if and only if an infection sequence exists. We prove both directions.

- **(Soundness)** Suppose there exists an infection sequence:

$$(C_a, t_1), (C_p, t_2), \ldots, (C_b, t_k) \quad \text{with } t_1 \leq t_2 \leq \ldots \leq t_k.$$

  By our graph construction, each consecutive pair in this sequence is connected by either:

  - a communication edge $(C_i, t) \to (C_j, t)$, representing infection spread at time t, or
  - a forward-in-time edge $(C_i, t_1) \to (C_i, t_2)$, representing that C_i remains infected.

  Thus, the sequence forms a directed path from (C_a, t_1) to (C_b, t_k) in G. Since BFS explores all reachable nodes, it will find (C_b, t_k), and therefore output **Yes**. Hence, infection sequences imply BFS reachability.

- **(Completeness)** Suppose BFS outputs **Yes**. Then BFS discovered a path

$$(C_a, t_1) \to (C_p, t_2) \to \ldots \to (C_b, t_k)$$

  in G. Because every edge in G goes forward in time or connects two computers at the same time, the path is time-nondecreasing ($t_1 \leq t_2 \leq \ldots \leq t_k$). We can therefore interpret this BFS path as a valid time-respecting infection sequence, implying that infection could have spread from C_a at time x to C_b by time y.

Hence, BFS returns **Yes** if and only if such a time-respecting infection path exists. Therefore, the algorithm is both sound and complete.

Runtime: The graph has $O(m)$ vertices and edges, and BFS runs in $O(|V| + |E|) = O(m + n)$ time. Thus, the algorithm satisfies the required time bound.

Question 5

You're helping a group of ethnographers analyze some oral history data they've collected by interviewing members of a village to learn about the lives of people who've lived there over the past two hundred years.

From these interviews, they've learned about a set of n people (all of them now deceased), whom we'll denote $P_1, P_2, \ldots, P_n$. They've also collected facts about when these people lived relative to one another. Each fact has one of the following two forms:

- **For some i and j, person P_i died before person P_j was born; or**
- **For some i and j, the life spans of P_i and P_j overlapped at least partially.**

Naturally, they're not sure that all these facts are correct; memories are not so good, and a lot of this was passed down by word of mouth. So what they'd like you to determine is whether the data they've collected is at least internally consistent, in the sense that there could have existed a set of people for which all the facts they've learned simultaneously hold.

Give an efficient algorithm to do this: either it should produce proposed dates of birth and death for each of the n people so that all the facts hold true, or it should report (correctly) that no such dates can exist—that is, the facts collected by the ethnographers are not internally consistent.

Solution.

We represent each person P_i with two events: their *birth* b_i and their *death* d_i. For every person we know that life proceeds forward in time:

$$b_i \to d_i.$$

We now encode all reported facts as temporal precedence constraints and build a directed graph $G = (V, E)$ whose vertices are all birth and death events.

- For every person P_i, add nodes b_i and d_i and an edge $b_i \to d_i$.
- For every fact "P_i died before P_j was born," add an edge $d_i \to b_j$.
- For every fact "P_i and P_j overlapped," add edges $b_i \to d_j$ and $b_j \to d_i$ (their lifespans intersect).

If the resulting directed graph G contains a directed cycle, then the facts are contradictory. If G is acyclic, we can assign consistent birth and death times by placing events along the time line according to any topological order.

Algorithm.

1. Construct the directed graph $G = (V, E)$ as described above.
2. Perform a topological sort on G (using either Kahn's algorithm or a DFS-based

cycle detection).

3. If a cycle is detected, report **Inconsistent**; otherwise report **Consistent** and output the topological order as a valid chronological arrangement of all birth and death events.

Runtime analysis. The graph has $2n$ vertices (b_i, d_i for each person) and each fact contributes at most two directed edges. Hence $|V| = O(n)$ and $|E| = O(n+m)$, where m is the number of given facts. Topological sorting runs in $O(|V| + |E|) = O(n+m)$ time. Thus, the algorithm satisfies the required linear-time bound.

Proof of correctness.

(Soundness). If the algorithm finds that G is acyclic, then a topological order $\prec$ exists over all events. Assign each event a distinct time increasing along $\prec$. Every edge $u \to v$ corresponds to a constraint "u occurs before v," and since edges follow $\prec$, all constraints are satisfied. Hence the reported timeline is consistent with all facts.

(Completeness). Conversely, suppose the ethnographers' statements are in fact consistent: there exist times $\{b_i, d_i\}$ satisfying all constraints. Ordering all events by their real times yields a total order that respects each required precedence, so G must be acyclic. Therefore, if G contained a directed cycle, some event would need to occur before itself, which is impossible. Hence a cycle implies inconsistency.

Question 6

Given an array `arr[]` of size n, the task is to find the minimum number of jumps to reach the last index of the array starting from index 0. In one jump you can perform one of the following actions:

- **Move from index i to index $i-1$.**
- **Move from index i to index $i+1$.**
- **Move from index i to index j if $\texttt{arr}[i] = \texttt{arr}[j]$ and $i \neq j$.**

Note: You cannot jump outside of the array at any time.

Example.

Input:

$$\texttt{arr} = [100, -23, -23, 404, 100, 23, 23, 23, 3, 404]$$

Output: 3

Explanation: Valid jump indices are $0 \to 4 \to 3 \to 9$**.**

Solution.

We can interpret the array indices as vertices in an unweighted graph $G = (V, E)$, where $V = \{0, 1, \ldots, n-1\}$ and edges represent valid jumps between indices.

Graph Construction.

- For each index i $(0 \leq i < n)$:
  - Add an edge $(i, i-1)$ if $i > 0$ (move left).
  - Add an edge $(i, i+1)$ if $i < n-1$ (move right).
  - For all $j \neq i$ such that $\texttt{arr}[i] = \texttt{arr}[j]$, add an edge (i, j) (same-value jump). It is good to use a hash map that groups all indices having the same value together. The key will be a value from the array (arr[i]), and the value will be a list of all indices that contain that number.

Thus, G is an undirected, unweighted graph where each edge corresponds to a legal jump.

Algorithm.

1. **Graph Construction.**
   - Let G be an initially empty graph with n vertices $\{0, 1, \ldots, n-1\}$.
   - For each vertex i, initialize $G[i]$ as an empty list of neighbors.
   - Create a hash map M that groups all indices by their array value:

     $$M[v] = \{i \mid \texttt{arr}[i] = v\}.$$

     For example, if $\texttt{arr} = [100, -23, -23, 404, 100]$, then

     $$M = \{100 : [0, 4],\ -23 : [1, 2],\ 404 : [3]\}.$$

   - For each index i:
     - If $i > 0$, add edge $(i, i-1)$ to G.
     - If $i < n-1$, add edge $(i, i+1)$ to G.
     - For all j in $M[\texttt{arr}[i]]$ with $j \neq i$, add edge (i, j) to G.
   - (Optional optimization) After adding same-value edges for value $\texttt{arr}[i]$, clear $M[\texttt{arr}[i]]$ so that this group will not be processed again.
2. **Breadth-First Search (BFS).**

- Perform a Breadth-First Search (BFS) on G starting from node 0.
- During BFS, record each vertex's level (distance) from node 0 in the BFS tree.
- When node $n-1$ is first discovered, its level in the BFS tree is the minimu number of jumps.

Runtime Analysis.

Let n be the length of the array. The graph $G = (V, E)$ has one vertex for each index, so $|V| = n$.

- Each index i has at most two adjacency edges $(i, i-1)$ and $(i, i+1)$.
- All indices sharing the same value `arr`$[i]$ are also connected via same-value edges. Using a hash map to store these index groups and clearing each group once it is used ensures that each index and value group is processed only once.

Thus, every vertex and edge is examined at most a constant number of times during BFS. The total running time is therefore $O(n)$.

Proof of Correctness.

We prove that the BFS algorithm correctly returns the minimum number of jumps needed to reach the last index.

- *Soundness.* Every valid jump operation (to an adjacent index or to another index with the same value) is represented as an undirected edge in G. Breadth-First Search explores all nodes in nondecreasing order of their level (i.e., shortest path length from node 0). Therefore, when node $n-1$ is first discovered by BFS, the path found corresponds to a valid sequence of jumps with the minimum possible number of steps.
- *Completeness.* If there exists any valid sequence of jumps from index 0 to index $n-1$, it corresponds to a path in G. Since BFS explores all reachable vertices, it will necessarily discover node $n-1$. The level of node $n-1$ in the BFS tree equals the minimum number of jumps along this path.
- Hence, the algorithm always terminates, and the BFS level of node $n-1$ equals the optimal (fewest) number of jumps from index 0 to $n-1$.