Hey everyone! Quick update before we start—Han have an emergency, so he won't be able to join us today. It's just my part tonight!

I'll walk you through two topics:

- First, a quick optimization in Union Find called **Union by Rank**.
- Then we'll move on to one of the most classic graph algorithms: **Dijkstra's algorithm**, both the basic version and the one optimized with a heap.

Union Find

Let's start with a quick recap of last week. We talked about the **Union Find** data structure — which basically helps us group nodes and check if two things are connected.

It's got two main functions:

- `union(x, y)` → puts x and y in the same group
- `find(x)` → tells us which group x belongs to — or who the "boss" of that group is

We also learned a neat trick called **path compression** that makes the `find` operation a lot faster.

And here's the general template we usually follow — you'll see it over and over in graph problems. So it's good to get used to how we build it and when to use it.

```
class UnionFind:
    def __init__(self, n):
        self.parent = [i for i in range(n + 1)]  # Each node starts as its own parent
        self.count = n  # Number of connected components

    def find(self, x):
        '''
        Find the root of x with path compression
        '''
        if self.parent[x] != x:
            self.parent[x] = self.find(self.parent[x])  # Path compression: point x directly
to its root
        return self.parent[x]

    def union(self, x, y):
        '''
        Union the sets that x and y belong to
        '''
        rootX = self.find(x)
        rootY = self.find(y)

        if rootX == rootY:
            return False  # x and y are already in the same set — union fails (cycle
detected)

        self.parent[rootY] = rootX  # Attach rootY under rootX (or the other way around)
        self.count -= 1
        return True  # Union succeeded

    def connected(self, x, y):
        '''
        Check if x and y are in the same set
        '''
        return self.find(x) == self.find(y)

    def count(self):
        '''
        Return the number of connected components
        '''
        return self.count
```

But we skipped another key optimization: **Union by Rank** — which is all about making `union()` more efficient.

What is Union by Rank?

To put it simply — Union by Rank is a smart rule for deciding who becomes the parent when merging two sets.

When we do `union(x, y)`, we're connecting two trees. If we're not careful, the merged tree can become tall and slow down future `find()` operations.

So the idea is: **Always attach the shorter tree to the taller one** — in other words, let the smaller group join the bigger one. This helps keep the tree flat and efficient.

What Is "Rank"?

In Union Find, rank is just a number we use to estimate how "tall" or "large" a tree is. It's not the exact height — just a **estimation** to guide our union decisions.

At the beginning, each node is its own root, so rank = 1.

Why Do We Need This?

Let me show you a worst-case scenario without Union by Rank. Say we have 7 nodes, and we merge like this:

```
union(1, 2)
union(2, 3)
union(3, 4)
union(4, 5)
union(5, 6)
union(6, 7)
```

And we always attach the new node to the previous one. You'll end up with a long, skinny chain like this:

```
1   # root in the group
 \
  2
    \
      3
        \
          4
            \
              5
                \
                  6
                    \
                      7
```

What Happens With Union by Rank?

If we use ranks to control how we merge, we'll attach smaller trees to bigger ones — and over time, the structure becomes much more balanced, like this:

```
        4
      /   \
     2      6
    / \   / \
   1   3 5   7
```

And once we add path compression, the structure becomes almost flat. That's how we make both `find()` and `union()` run really fast — almost like O(1) in practice.

How Do We Code It?

We add a `rank[]` array to track the rank of each root. Then inside `union()`, we compare ranks:

- If `rank[rootX] < rank[rootY]`, attach X under Y.

- If `rank[rootX] > rank[rootY]`, attach Y under X.
- If equal, pick one as root, and increase its rank by 1.

Here's the code with Union by Rank + Path Compression:

```
class UnionFind:
    def __init__(self, n):
        self.parent = [i for i in range(n + 1)]
        self.rank = [1] * (n + 1)
        self.count = n

    def find(self, x):
        if self.parent[x] != x:
            self.parent[x] = self.find(self.parent[x])  # Path compression
        return self.parent[x]

    def union(self, x, y):
        rootX = self.find(x)
        rootY = self.find(y)

        if rootX == rootY:
            return False  # Already in the same group

        # Union by rank
        if self.rank[rootX] < self.rank[rootY]:
            self.parent[rootX] = rootY
        elif self.rank[rootX] > self.rank[rootY]:
            self.parent[rootY] = rootX
        else:
            self.parent[rootY] = rootX
            self.rank[rootX] += 1

        self.count -= 1
        return True

    def connected(self, x, y):
        return self.find(x) == self.find(y)

    def count(self):
        return self.count
```

When you combine path compression + union by rank, the time complexity becomes **O(α(n))** — which is so close to constant time that it's basically instant for all practical input sizes.

Practice: Leetcode 684

Let’s now practice Union Find with a classic Leetcode problem. I will give you 2 minutes to read this problem.

Here’s what the problem says:

- You're given a graph that *was* a tree — meaning:
  - It has `n` nodes.
  - It has `n - 1` edges.
  - It's connected, and there is **no cycle**.
- But now, **one extra edge** is added.
- That one edge creates **a cycle**.
- Your task is to **find the extra edge that caused the cycle**.

```
'''
original graph -> tree with n nodes and n-1 edges -> undirectional, no cycle
action -> add one edge -> a cycle
return -> which edge is added to produce a cycle?
'''
```

What we should do is to find the edge which causes the cycle. Let’s visualize it using a small example:

```
'''
      3
    /   \
  2       4
 /          \
1             5
'''
```

This is a tree — everything is connected, and no cycles. This is a connected component, and a union-find function can be used to check whether two nodes belong to the same connected component. If a union-find function is used to check, we find that node 1 and node 5 are in the same set.

Now, imagine we add a new edge: `[1, 5]`.

```
'''
      3
    /   \
  2       4
 /          \
1 --------- 5
'''
```

We now have a cycle!

Analyzing the relationship before and after the cycle is formed, we find that if find(1) == find(5) and edge[1, 5] exist at the same time, then a cycle will be formed.

Here's the key idea: When we try to add an edge `[u, v]`, we can use Union Find to check if u and v are already in the same connected component.

- If they are **not** connected yet → we can safely union them.
- If they **are** already connected → adding this edge would create a **cycle**!

And that's exactly how we'll detect the redundant connection.

Code

Next step we will write the code:

Even though the problem is simple enough to solve without a formal class, we'll still follow the **Union Find template** — to get familiar with writing it properly.

There are 3 standard steps we mentioned last week:

1. Initialize the parent array.
2. Write the `find()` function with path compression.
3. Write the `union()` function.

```
# Initialize the parent array.
# Write the find() function with path compression.
# Write the union() function.
```

```
class UnionFind:
    def __init__(self, n):
        self.parent = [i for i in range(n + 1)]

    def find(self, x):
        if self.parent[x] != x:
            self.parent[x] = self.find(self.parent[x])  # path compression
        return self.parent[x]

    def union(self, x, y):
        rootX = self.find(x)
        rootY = self.find(y)

        if rootX == rootY:
            return False  # already connected — would form a cycle
        self.parent[rootY] = rootX  # merge
        return True
```

Because we have mentioned union by rank optimization, we will add this in the class. There are 2 steps:

- Add rank in the initialization part
- Change attach behaviour in the union function.

```
class UnionFind:
    def __init__(self, n):
        self.parent = [i for i in range(n + 1)]  # nodes start at 1
        self.rank = [1 for _ in range(n + 1)]

    def find(self, x):
        if self.parent[x] != x:
            self.parent[x] = self.find(self.parent[x])  # path compression
        return self.parent[x]

    def union(self, x, y):
        rootX = self.find(x)
        rootY = self.find(y)

        if rootX == rootY:
            return False  # already connected

        # Union by rank: attach smaller tree to larger one
        if self.rank[rootX] < self.rank[rootY]:
            self.parent[rootX] = rootY
        elif self.rank[rootX] > self.rank[rootY]:
            self.parent[rootY] = rootX
        else:
            self.parent[rootY] = rootX
            self.rank[rootX] += 1

        return True
```

Now let's solve the actual problem:

```python
class Solution(object):
    def findRedundantConnection(self, edges):
        n = len(edges)
        helper = UnionFind(n)

        for u, v in edges:
            if not uf.union(u, v):
                return [u, v]  # found the redundant edge

        return []
```

This is how we use union find in leetcode problem. The key point is remenber the template and write the necessary part at first. Then use the fucntions to solve the problem.

Next, let’s move on to a new topic — the **Dijkstra Algorithm**. It is one of the most classic algorithms for solving **shortest path problems**.

Dijkstra Algorithm

In general, shortest path problem has two categories:

- Single-source shortest path (SSSP) — From *one* starting node to *all* other nodes.
- Multi-source shortest path — From *every* node to *every* other node.

Dijkstra solves the first one: **Single Source Shortest Path (SSSP)**.

What Problem Does Dijkstra Solve?

- You're given a graph where all the **edge weights are non-negative**.
- You’re also given a **starting node**.
- Your goal is to find the **shortest distance** from this start node to **every other node** in the graph.

A few things to keep in mind before applying Dijkstra:

- All edge weights must be **non-negative**. If there are **negative weights**, Dijkstra **won’t work**.
- We have **a starting node**.
- It works for **directed** or **undirected** graphs.

Core idea

The core idea of Dijkstra is **greedy**: Each time starting from the "currently known shortest node", try to update its neighbors; repeat this process until the shortest path is determined for all points.

We can break the single process down it into 3 steps:

1. Pick the node that is **closest** to the starting point (among all nodes not yet visited).
2. Tag it as visited.
3. **Update the distances** to its neighboring nodes.

Repeat this process until all nodes have been taged visited with their shortest distances.

Basic Structure

We analyze the core idea to find what we should do when coding the algorithm.

1. Pick the node that is closest to the starting point (among all nodes not yet visited).
2. Tag it as visited.
3. Update the distances to its neighboring nodes.

I find two main things:

- I must keep track of the shortest known distance from the start to each node. So I use a A `minDist[]` array. Before we start, each element in this array is infinity, and the distance from start to itself is 0.
- I must mark if we’ve visited a node. So I use a `visited[]` array to track this information. All values start as `False`.

And there is 2 important operations which will influence the performance of the algorithm.

- How you find the closest node to the starting node?
- How you update the distance to current node's neighboring nodes?

Find the closest node is related to the difference between basic Dijkstra algorithm and heap opetimization Dijkstra algorithm.

Update the distance is related to how we build the graph?

Naive Dijkstra

In the **basic (naive)** version of Dijkstra's algorithm, we do the 3 steps:

1. Find the closest unvisited node — use for loop to find the current shortest point.
2. Mark it as visited.
3. Try to update its neighbors — see if going through this node gives a better path to others.

```python
def dijkstra(n, graph, start):
    INF = float('inf')
    dist = [INF] * n          # min distance from start to each node
    dist[start] = 0           # start node's distance to itself is 0

    visited = [False] * n     # whether the shortest path to this node is already finalized

    for _ in range(n):        # Do this n times — we’ll finalize at most n nodes
        u = -1
        # Find the unvisited node with the smallest distance
        for i in range(n):
            if not visited[i] and (u == -1 or dist[i] < dist[u]):
                u = i

        visited[u] = True  # Lock in this node's distance

        # Try to update all neighbors of u
        for v in range(n):
            if graph[u][v] != INF:  # There's an edge from u to v
                dist[v] = min(dist[v], dist[u] + graph[u][v])

    return dist
```

The time complexity here is **O(n^2)**. Because for each node, we're scanning through all other nodes to find the next "closest" one. That's... fine for small graphs or **dense** graphs (lots of edges).

Dijkstra with Heap Optimization

But for **sparse graphs**, where there are very few edges, this becomes extremely wasteful: You spend a lot of time scanning nodes that aren't connected to the current node.

- Sparse graph: about `O(n)` edges (like a tree or a road map)
- Dense graph: up to `O(n²)` edges (like a complete graph)

So to avoid waste, instead of scanning *every* node to find the closest one, can we just consider those nodes that are **actually connected** to our current node?

This is where a **priority queue** (a min heap) comes in: A heap is a great structure for quickly finding the

This is where a **priority queue** (a min-heap) comes in. A heap is a great structure for quickly finding the smallest value — which is exactly what we need to do repeatedly in Dijkstra.

We follow the same 3 steps in core idea, but change in details:

1. Find the closest unvisited node — Select the node closest to the source point and add the edge directly to the top heap (using the heap to automatically sort) Every time we take out an edge from the top of the heap, it is naturally the edge of the node closest to the source point..
2. Mark it as visited.
3. Try to update its neighbors.

```python
import heapq
from collections import defaultdict

def dijkstra(n, graph, start):
    """
    Heap-optimized Dijkstra algorithm.
    :param n: number of nodes (1-indexed)
    :param graph: adjacency list -> graph[u] = [(v, weight), ...]
    :param start: starting node
    :return: dist[i] = shortest distance from start to node i
    """
    dist = [float('inf')] * (n + 1)
    dist[start] = 0

    visited = [False] * (n + 1)
    heap = [(0, start)]  # (distance, node)

    while heap:
        d, u = heapq.heappop(heap)
        if visited[u]:
            continue
        visited[u] = True

        for v, w in graph[u]:
            if dist[u] + w < dist[v]:
                dist[v] = dist[u] + w
                heapq.heappush(heap, (dist[v], v))

    return dist
```

Here I would like to briefly mention a key part.

Build graph

Build graph

Before we even run Dijkstra, we need to build the graph in a way that's efficient to work with.

There are two common ways to represent graphs: **Adjacency Matrix** and **Adjacency List** — and each has its own use case.

Structure	When to Use	Works Well With...	Time Complexity
Adjacency List	Sparse graphs	Heap-optimized Dijkstra	`O(m log n)`
Adjacency Matrix	Dense graphs	Naive Dijkstra	`O(n²)`

Adjacency matrix

This is basically a grid. If you want to know whether node `u` is connected to node `v`, you just check `graph[u][v]`. Good for dense graphs, but **space and time expensive** for sparse ones.

```
graph = [[float('inf')] * (n) for _ in range(n)]

for u, v, w in times:
    graph[u][v] = w
```

Adjacency List

This is a dictionary where each key is a node, and the value is a list of `(neighbor, weight)` pairs. Much more efficient for **sparse graphs** — because we only store edges that actually exist.

```
from collections import defaultdict

graph = defaultdict(list)

for u, v, w in times:
    graph[u].append((v, w))
```

This way, You don't waste time scanning nonexistent edges. You only process actual neighbors — perfect for heap-based Dijkstra!

Practice: Leetcode 743

Alright, now let's walk through a practice problem that's perfect for applying Dijkstra's algorithm. I will give you 2 minutes to read this problem.

Let's break down the key information from the problem statement:

Let's break down the key information from the problem statement.

```
'''
Input:
- graph -> directed, weighted
- times -> list of edges: (u, v, weight)
- n -> total number of nodes (1 to n)
- k -> the source node where the signal starts

Action:
- Send a signal from node k

Goal:
- Return the time it takes for all nodes to receive the signal.
- If any node can’t receive it, return -1.

Important idea:
- The signal can travel in parallel across different paths.
- But the signal finishes only when the last node receives it.
- So we need to find the maximum time among all the shortest paths from k to each node.
'''
```

In other words, we’re looking for the **longest shortest path** from the source node `k` to every other node. Why longest? Because the signal is sent out in parallel — the total time to “finish” is when the **last node** gets the message. That’s the slowest path.

So our final answer is the max of all shortest distances from k and if any node is unreachable, we return `-1`.

We still follow 3 steps to write the code.

```
# build the graph

# initilization

# find the closest unvisited node
# Mark it as visited.
# Try to update its neighbors.
```

```
class Solution(object):
    def networkDelayTime(self, times, n, k):
        graph = defaultdict(list)
        for u, v, w in times:
            graph[u].append((v, w))

        min_dist = [float("inf")] * (n+1)

        visited = [False] * (n+1)
        min_dist[k] = 0

        for _ in range(n):
            u = -1
            for i in range(1, n+1):
                if not visited[i] and (u == -1 or min_dist[i] < min_dist[u]):
                    u = i

            visited[u] = True

            for v, w in graph[u]:
                if min_dist[u] + w < min_dist[v]:
                    min_dist[v] = min(min_dist[v], min_dist[u] + w)

        res = max(min_dist[1:])
        return res if res != float("inf") else -1
```

```
from collections import defaultdict
import heapq

class Solution(object):
    def networkDelayTime(self, times, n, k):
        graph = defaultdict(list)
        for u, v, w in times:
            graph[u].append((v, w))

        min_dist = [float("inf")] * (n+1)
        visited = [False] * (n+1)
        min_dist[k] = 0

        heap = [(0, k)]  # (distance, node)

        while heap:
            dist, u = heapq.heappop(heap)

            if visited[u]:
                continue
            visited[u] = True

            for v, w in graph[u]:
                if min_dist[u] + w < min_dist[v]:
                    min_dist[v] = min_dist[u] + w
                    heapq.heappush(heap, (min_dist[v], v))

        res = max(min_dist[1:])
        return res if res != float("inf") else -1
```

This is a textbook example of applying **single-source shortest path** with Dijkstra — and the heap version is

This is a textbook example of applying **single source shortest path** with Dijkstra — and the heap version is much faster on large inputs.

Alright, that's everything I planned to cover today. Does anyone have any questions?

If not, we'll wrap up the session here. I'll upload all the materials to my repo later — feel free to check them out there if you need a reference.

https://github.com/chichizhang0510/Tech-Crossroad-WeeklyLeetcode