ChatGPT

알고리즘 스터디: 9주차 – 그래프 심화 (최단 경로 알고리즘 및 최소 신장 트리)

목차

0. 지난주 학습 내용 되짚기

지난 주 8주차에서는 **그래프 알고리즘의 기초 개념**들을 학습했습니다. 그래프의 **표현 방식**으로 인접 행렬과 인접 리스트 등의 차이를 살펴보고, **깊이 우선 탐색(DFS)**과 **너비 우선 탐색(BFS)**을 구현하며 그래프 **탐색** 방법을 익혔습니다. 이를 통해 그래프의 **사이클 존재 여부**, **연결 요소** 개수, **이분 그래프** 판별 등의 **그래프 주요 속성**을 분석하는 기법도 배웠습니다. 또한 **트리**를 특별한 그래프로 간주하여 트리의 지름, 서브트리 등 **트리의 특성**을 이해하고, **트리 순회 (전위/중위/후위)** 방법과 주어진 순회 결과로부터 **트리 복원**하는 문제를 다루었습니다. 마지막으로 **위상정렬** 알고리즘(Kahn 알고리즘과 DFS 기반 방법)을 통해 **사이클이 없는 방향 그래프(DAG)**에서 선행 순서를 유지하면서 모든 노드를 나열하는 방법을 배웠습니다. 이러한 그래프 기초 내용들은 **최단 경로 문제**와 **MST** 등의 심화 알고리즘을 공부하는 데 밑바탕이 됩니다. **이번 주 9주차에서는** 지난주에 익힌 그래프 개념을 바탕으로, 다익스트라, 벨만-포드, 플로이드-워셜과 같은 **최단 경로 알고리즘**들과 크루스칼/프림의 **최소 신장 트리(MST)** 알고리즘, 그리고 이를 활용하는 **DSU(Union-Find)** 자료구조까지 심화 주제를 학습해보겠습니다.

1. 다익스트라 알고리즘 (3가지 버전)

개요: 다익스트라 알고리즘은 가중치가 **음수가 아닌** 그래프에서 하나의 시작 정점으로부터 다른 모든 정점까지의 **최단 경로**를 찾아주는 대표적인 알고리즘입니다. DFS/BFS와 유사한 탐색 기반이지만, 간선 가중치를 고려하여 **가장 가까운 정점부터 탐색**하는 **그리디 알고리즘**의 일종입니다. 기본 개념은 "현재까지 발견된 가장 짧은 거리의 정점을 선택해, 인접한 간선을 따라 거리를 갱신(relaxation)한다"는 과정을 모든 정점을 방문할 때까지 반복하는 것입니다. **전제 조건:** 모든 간선의 가중치가 0 이상이어야 하며(음수 가중치가 있으면 올바르게 동작하지 않음), 연결 가중치 그래프에서 적용 가능합니다.

알고리즘 동작 원리:
다익스트라는 단일 출발지 최단 경로(Single-Source Shortest Path, SSSP) 문제를 해결하는 알고리즘으로 다음과 같이 진행됩니다:

1. **초기화:** 시작 정점의 거리를 0으로, 그 외 모든 정점의 거리를 무한대(∞)로 설정합니다. 각 정점은 아직 방문되지 않은 상태입니다.
2. **최단 거리 미방문 정점 선택:** 현재 미방문 정점 중에서 **가장 작은 (잠정) 거리값**을 가진 정점을 선택합니다. 처음에는 시작 정점이 선택될 것입니다.

3. **인접 간선 완화(relaxation):** 선택된 정점을 거쳐 다른 인접한 정점으로 가는 경로를 고려하여, 인접 정점들의 거리를 갱신합니다. 즉, 정점 u가 선택되었다면 각 간선 (u, v)에 대해 `dist[v] > dist[u] + w(u,v)` 이면 `dist[v] = dist[u] + w`로 업데이트합니다.
4. **방문 처리:** 한 번 선택된 정점은 최단 경로 거리가 확정되었다고 보고 방문 처리합니다.
5. **반복:** 모든 정점을 방문하거나 더 이상 거리를 갱신할 수 없을 때까지 2~4 과정을 반복합니다.

위 과정에서 선택되지 않은 정점들의 거리는 시작점으로부터의 **최단 경로 길이**를 지속적으로 유지하며, 알고리즘 종료 시 이 값들이 최종 최단 거리 결과가 됩니다. 다익스트라 알고리즘은 **음수 간선이 없는 가중치 그래프**에서 동작하며, 음수 가중치가 있으면 잘못된 결과를 낳게 됩니다.

다익스트라 알고리즘에는 구현 방법에 따라 **세 가지 버전**이 존재하며, 사용된 **자료구조와 구현 방식에 따라 시간 복잡도**가 달라집니다. 각각의 버전을 살펴보고, 상황에 맞는 활용 방법과 장단점을 비교해보겠습니다.

1.1 기본 구현 버전 (배열 사용, O(N^2))

구현 방식: 가장 단순한 형태의 다익스트라 알고리즘은 **배열이나 선형 탐색**을 통해 미방문 정점 중 최단 거리를 가진 정점을 찾는 방식입니다. 인접 리스트 대신 인접 행렬 또는 간선 리스트를 사용하고, 우선순위 큐를 사용하지 않고 매 단계마다 **선형 탐색**으로 최소 거리를 지닌 정점을 선택합니다. 구현이 비교적 간단하며, 우선순위 큐를 다루기 어려운 초보 단계에서 종종 교육용으로 사용됩니다.

시간 복잡도: 선택 단계에서 매번 남은 정점들 중 최소를 찾는 데 O(N)이 걸리고 이를 N번 반복하므로 전체로 O(N^2) 시간이 걸립니다 [1]. 간선 개수가 적더라도 최악의 경우 정점 수 N에 의해 시간 복잡도가 결정되므로, **밀집 그래프**(간선이 매우 많은 그래프)나 N이 큰 경우에는 비효율적입니다. 예를 들어 N=5,000일 때 O(N^2)=25백만, N=50,000이면 25억 번의 연산으로 현실적으로 수행이 불가능해집니다. 다만 **간선 수 M에 크게 영향받지 않기 때문에** 그래프가 **매우 작은 경우나 밀집한 경우**에는 단순 구현으로도 사용할 수 있습니다.

장점: 우선순위 큐 등의 고급 자료구조 없이도 이해하고 구현하기 쉬워, 알고리즘 학습 초기에 원리를 파악하기 좋습니다.
단점: 시간 복잡도가 높아, 정점 수가 많은 그래프에는 부적합합니다. 또한 구현 시 2중 루프를 사용해야 하므로 코드상으로도 PQ 버전에 비해 덜 직관적일 수 있습니다.

예시: 아래는 기본 구현 버전의 다익스트라 알고리즘을 **Java**와 **Python**으로 나타낸 코드 예시입니다. 인접 행렬 또는 리스트로 그래프가 주어졌다고 가정하고, 배열 `dist[]`를 사용하여 거리를 저장합니다.

```
// Java: 다익스트라 기본 구현 (배열 사용)
int N = 5;  // 정점 개수 (예시)
int[][] graph = new int[N][N];
// graph[u][v] = 가중치 (간선 없으면 어떤 큰 값 or INF)
// ... (그래프 초기화: graph 배열 채우기)
int[] dist = new int[N];
boolean[] visited = new boolean[N];
int INF = Integer.MAX_VALUE;
Arrays.fill(dist, INF);
int start = 0;
dist[start] = 0;

for (int i = 0; i < N; i++) {
  // 1. 미방문 정점 중 최단 거리 정점 찾기
  int u = -1;
  int minDist = INF;
```

```
  for (int v = 0; v < N; v++) {
    if (!visited[v] && dist[v] < minDist) {
      u = v;
      minDist = dist[v];
    }
  }
  if (u == -1) break;  // 더 이상 방문할 정점이 없음 (그래프가 연결되지 않은 경우 등)
  // 2. 선택 정점 u 방문 처리
  visited[u] = true;
  // 3. 거리 갱신 (relaxation)
  for (int v = 0; v < N; v++) {
    if (!visited[v] && graph[u][v] != INF) {  // u의 인접 정점
      if (dist[v] > dist[u] + graph[u][v]) {
        dist[v] = dist[u] + graph[u][v];
      }
    }
  }
}
// 결과: dist[]에 start로부터 각 정점까지의 최단 거리 저장
```

```
# Python: 다익스트라 기본 구현 (리스트 사용)
N = 5  # 정점 개수 (예시)
INF = float('inf')
# 인접 행렬 그래프 예시 (가중치 없으면 INF)
graph = [[INF]*N for _ in range(N)]
# ... (그래프 초기화: graph[u][v] = 가중치)
start = 0
dist = [INF]*N
visited = [False]*N
dist[start] = 0

for i in range(N):
    # 1. 미방문 정점 중 최단 거리 정점 선택
    u = -1
    min_dist = INF
    for v in range(N):
        if not visited[v] and dist[v] < min_dist:
            u = v
            min_dist = dist[v]
    if u == -1:
        break
    # 2. 선택된 정점 u 방문 처리
    visited[u] = True
    # 3. 인접 정점 거리 갱신
    for v in range(N):
        if not visited[v] and graph[u][v] != INF:
            if dist[v] > dist[u] + graph[u][v]:
                dist[v] = dist[u] + graph[u][v]
# 결과: dist 리스트에 start로부터의 최단 거리들이 저장
```

위 코드에서 사용한 `graph`는 인접 행렬 형식으로, 간선이 없는 경우 `INF` 값으로 표현했습니다. 방문하지 않은 정점 중 최단 거리를 가진 `u`를 찾아 방문한 뒤, 그 이웃들의 거리를 차례로 갱신하는 구조를 갖습니다.

1.2 우선순위 큐(Priority Queue) 활용 버전 (O((N+M) log N))

구현 방식: 다익스트라 알고리즘의 효율을 높이기 위해 **우선순위 큐(힙)**를 사용한 구현이 일반적입니다. C++에서는 `priority_queue`, Java에서는 `PriorityQueue`, Python에서는 `heapq` 모듈 등을 이용해 최소 힙(min-heap)을 만들어, 항상 현재 **가장 작은 거리**를 갖는 정점을 신속히 꺼낼 수 있습니다. 우선순위 큐를 이용하면 최단 거리 정점을 찾는 작업이 선형 탐색(O(N)) 대신 **로그 시간**(O(log N))에 수행되므로 전체 시간 복잡도가 크게 줄어듭니다 [1].

시간 복잡도: 우선순위 큐 버전에서는 각 정점을 한 번씩 큐에 넣고 뺄 때 O(log N) 비용이 들고, 각 간선에 대해 거리 갱신(간선 relaxation 시 우선순위 큐에 새로운 경로 삽입/갱신)을 수행하므로 **총 연산 수는 O((N + M) log N)** 정도가 됩니다 [2] [3]. 일반적으로 **희소 그래프**(간선 개수가 적은 그래프)나 N, M이 매우 큰 경우 이 방식이 훨씬 효율적입니다. 예를 들어 N=1만, M=5만인 그래프에서 기본 버전이 O(N^2)=1e8 번 연산이 필요한 반면, PQ 버전은 대략 O((N+M) log N) ≈ 60만여 번 정도의 연산으로 끝낼 수 있어 실용적입니다. 우선순위 큐를 활용한 다익스트라 알고리즘은 **모든 간선 가중치가 양수 또는 0일 때** 최단 거리를 정확히 찾아내며, 이는 **일반적인 그래프에서 최단 경로를 구하는 가장 효율적인 알고리즘**으로 오랫동안 여겨져 왔습니다 [2].

장점: 시간 복잡도가 크게 개선되어 **대부분의 현실적인 그래프에서 빠르게 동작**합니다. 표준 라이브러리로 제공되는 우선순위 큐를 활용하면 코드 구현도 비교적 간결합니다.
단점: 우선순위 큐를 사용함에 따라 코드의 복잡도가 약간 증가하며, `decrease-key` 연산이 없는 언어(Python 등)에서는 기존 값을 새 값으로 추가 삽입하고 불필요한 항목을 무시하는 방식으로 구현해야 합니다. 또한 **간선 가중치가 모두 동일**한 특수한 그래프(예: 모든 가중치=1)에서는 BFS를 이용한 **0-1 BFS** 등의 알고리즘이 더 효율적일 수 있습니다.

예시: 아래는 우선순위 큐를 이용한 다익스트라 구현의 **Java**와 **Python** 코드 예시입니다. 그래프는 인접 리스트 형태(`graph[u]`에 `(v, w)` 튜플이나 객체를 저장)로 표현하며, 각 정점의 인접 목록을 탐색하여 거리 갱신을 수행합니다.

```
// Java: 다익스트라 (우선순위 큐 사용) 구현
import java.util.*;
// ...
int N = 5;
List<int[]>[] graph = new ArrayList[N];
for(int i=0;i<N;i++){
   graph[i] = new ArrayList<>();
}
// 그래프 초기화: graph[u].add(new int[]{v, w}); 형태로 간선 추가
int start = 0;
int INF = Integer.MAX_VALUE;
int[] dist = new int[N];
Arrays.fill(dist, INF);
dist[start] = 0;
boolean[] visited = new boolean[N];
// 최소힙 (거리 기준)
PriorityQueue<int[]> pq = new PriorityQueue<>(Comparator.comparingInt(a -> a[0]));
pq.offer(new int[]{0, start});  // {현재까지 거리, 정점} 형태

while(!pq.isEmpty()){
```

```
    int[] cur = pq.poll();
    int d = cur[0];
    int u = cur[1];
    if(visited[u]) continue;        // 이미 방문 확정된 정점은 건너뜀
    visited[u] = true;
    if(d > dist[u]) continue;       // 더 긴 경로이면 무시
    for(int[] edge : graph[u]){     // u의 모든 인접 간선에 대해
      int v = edge[0];
      int w = edge[1];
      if(dist[v] > dist[u] + w){    // 더 짧은 경로 발견 시
        dist[v] = dist[u] + w;
        pq.offer(new int[]{ dist[v], v });
      }
    }
  }
}
// 결과: dist 배열에 start로부터 각 정점까지의 최단 거리
```

```
# Python: 다익스트라 (우선순위 큐 사용) 구현
import heapq
N = 5
graph = [[] for _ in range(N)]
# 그래프 초기화: graph[u].append((v, w)) 형태로 간선 추가
start = 0
INF = float('inf')
dist = [INF]*N
dist[start] = 0
visited = [False]*N
pq = [(0, start)]  # (현재까지 거리, 정점)
heapq.heapify(pq)

while pq:
    d, u = heapq.heappop(pq)
    if visited[u]:
        continue
    visited[u] = True
    if d > dist[u]:
        continue
    for v, w in graph[u]:
        if not visited[v] and dist[v] > dist[u] + w:
            dist[v] = dist[u] + w
            heapq.heappush(pq, (dist[v], v))
# 결과: dist 리스트에 start부터 각 정점까지의 최단 거리
```

위 코드에서는 우선순위 큐(`PriorityQueue` 혹은 `heapq`)를 사용하여 항상 현재 **가장 가까운 후보 정점**을 빠르게 꺼내 처리합니다. Python 구현에서는 `heapq`를 사용하며, 기존 보다 더 작은 거리가 발견될 때마다 `(새 거리, 정점)` 튜플을 힙에 추가합니다. 이미 확정된 정점을 `visited`로 표시해 다시 처리하지 않고, 혹시 힙에 남아 있는 이전 경로 정보(`d > dist[u]`)는 무시하여 효율을 높입니다. 이러한 구현 기법으로 **시간 복잡도 O(M log N)**에 다익스트라를 구현할 수 있습니다 [2].

1.3 최신 연구 개선 버전 (이분화 + 부분 정렬, O(M · log^{2/3} N))

다익스트라 알고리즘은 오랫동안 **정렬 단계의 한계(sort barrier)** 때문에 O(M log N)이 최선이라고 여겨져 왔습니다 [4]. 그러나 **2025년**에 이르러 중국 칭화대 연구진에 의해 이 복잡도를 개선한 새로운 알고리즘이 발표되어 이목을 끌었습니다 [5] [6]. 이 알고리즘은 다익스트라의 핵심인 "가장 가까운 정점 선택" 작업을 **완전 정렬(full sorting)**하지 않고도 수행하는 기법으로, 이론적으로 **O(M · log^{2/3} N)**의 시간 복잡도를 달성합니다 [6]. 이는 전통적인 O(M log N)보다도 느리게 증가하는 (log N의 2/3 제곱) 복잡도로, **65년간 깨어지지 않던 한계를 돌파**한 사례로 평가받습니다.

핵심 아이디어: 새로운 알고리즘은 그래프 탐색 중 **프론티어(경계)**에 있는 정점들을 일일이 정렬하지 않고, **여러 군집으로 묶어 부분 정렬**하는 전략을 취합니다 [7]. 구체적으로: - 아직 방문되지 않은 후보 정점들을 일부 그룹으로 모아 **"대표 노드"**들을 선별합니다. - 대표 노드들만 우선적으로 처리하여 거리를 갱신한 후, 다시 세분화하는 **분할 정복** 접근을 사용합니다. - 이 과정에서 **벨만-포드**와 유사한 **반복 완화(relaxation)**를 섞어 사용함으로써, 전체 정점을 완전히 정렬하지 않고도 최단 경로 계산이 가능합니다 [6].

장점 및 단점: 이 알고리즘은 **이론적으로** O(M log^{2/3} N)이라는 더 빠른 성능을 보장하므로, 노드와 간선 수가 매우 큰 **희소 그래프**에서 잠재적으로 유리합니다 [8]. 그러나 알고리즘이 상당히 복잡하고 재귀적이며, 구현 상 크게 개선된 상수 시간 요소가 없으면 실제 성능은 기존 알고리즘보다 느릴 수 있습니다 [9]. 작은 그래프에서는 오히려 부가 비용으로 인해 다익스트라보다 느리며, 구현 난이도도 높습니다 [9] [10]. 따라서 현 시점에 **실제 서비스**(예: 지도 길찾기 등)에 즉시 응용되기보다는, **이론적 진전**으로서의 의미가 큽니다 [11]. 그럼에도 불구하고 이 성과는 **정렬 장벽(sort barrier)**을 깨뜨렸다는 점에서 중요하며, 향후 더 단순하고 실용적인 개선 알고리즘의 등장 가능성을 열었다는 평가를 받고 있습니다 [12].

비고: 해당 알고리즘은 최신 연구 주제로, **입문자 수준에서는 상세 구현을 다루지 않아도** 무방합니다. 다만 알고리즘 분야에서도 지속적인 개선이 일어나고 있음을 소개하기 위해 언급하였습니다. 실제 코딩 테스트나 대회에서는 여전히 전통적인 우선순위 큐 활용 다익스트라(O(M log N))가 사용되며, 0-1 BFS처럼 특정 조건에서 최적화된 알고리즘도 자주 활용됩니다.

2. 벨만-포드 알고리즘

개요: 벨만-포드 알고리즘은 다익스트라와 마찬가지로 **단일 출발지 최단 경로(SSSP)** 문제를 해결하는 알고리즘이지만, **음수 가중치(edge weight)**를 허용한다는 점에서 더 **범용적**입니다 [13]. 즉, 그래프에 **음의 가중치 간선**이 포함되어 있어도 최단 경로를 찾을 수 있으며, 만약 **음수 사이클(사이클의 가중치 합이 음수인 경우)**이 존재할 경우 이를 **검출**할 수 있습니다 [14]. 이러한 유연성 때문에 다익스트라로 풀 수 없는 문제(음수 간선 존재)를 벨만-포드로 풀 수 있지만, **시간 복잡도** 측면에서는 다익스트라보다 느립니다.

알고리즘 원리: 벨만-포드는 동적 계획법(DP)의 원리에 기반하여, **간선 완화(relaxation)** 과정을 그래프에 대해 **V-1번 반복**합니다. 여기서 V는 정점의 개수입니다. 완화 과정이란 모든 간선 (u, v)에 대해 `dist[v] = min(dist[v], dist[u] + w(u,v))` 형태로 거리를 갱신하는 것입니다. 이론적으로 최단 경로는 최대 (V-1)개의 간선을 포함할 수 있으므로, V-1번의 반복으로 모든 최단 경로를 발견할 수 있습니다. 주요 과정은 다음과 같습니다:

- **초기화:** 시작 정점의 거리만 0으로 설정하고, 나머지 정점들의 거리를 무한대로 둡니다.
- **간선 완화 반복:** 그래프의 **모든 간선**에 대해 (V-1)번 루프를 돌면서 매 반복마다 모든 간선을 검사하여 거리를 갱신합니다. 이때 어떠한 반복에서도 갱신이 일어나지 않으면 조기에 종료할 수 있습니다.
- **음수 사이클 검사:** V-1번 반복 후에도 한 번 더 **전체 간선 완화**를 시도하여, 갱신이 발생하면 음수 사이클이 존재함을 알 수 있습니다. 음수 사이클이 존재하면 해당 사이클에 도달 가능한 정점들의 최단 거리는 **무한히 작아질 수 있으므로** (문제에 따라) 별도의 처리가 필요합니다. 보통은 음수 사이클 존재 여부를 출력하거나 경고합니다.

시간 복잡도: 벨만-포드 알고리즘은 **총 V-1번의 루프** 안에 **모든 M개의 간선**을 검사하므로 **O(V * M)** 시간, 빅오 표기법으로 흔히 **O(N * M)**의 시간 복잡도를 가집니다 [15] [16]. 이 복잡도는 최악의 경우 다익스트라(O(M log N))보다 훨씬

느리며, 특히 **밀집 그래프**(M ≈ N^2)에 대해서는 O(N^3)에 근접해 실용적이지 않습니다. 따라서 벨만-포드는 **그래프에 음수 간선이 존재**하는 특수한 경우에만 사용하는 것이 일반적입니다. 또한 음수 간선이 있더라도 **음수 사이클**이 없고, **N이 작을 때**(예: N 수백 이하) 현실적으로 사용 가능합니다. 그 외의 경우에는 보통 다익스트라 등 더 효율적인 알고리즘을 택하게 됩니다.

장점: 음수 가중치를 처리할 수 있으며, 구현이 비교적 단순합니다. 또한 **음수 사이클 존재 여부**까지 검사할 수 있어, 그래프에 **해답이 정의되지 않는 경우**(무한히 감소하는 경로)도 감지 가능합니다 [14]. 이러한 특성 때문에 통화 환율 문제(환율 차익: arbitrage)나 특정 그래프 이론 문제 등에서 활용됩니다.
단점: 시간 복잡도가 높아, N이나 M이 큰 경우 실행이 매우 오래 걸립니다. 또한 음수 사이클이 존재하면 경로 가중치가 계속 줄어들기 때문에, 일반적인 최단 경로 문제로는 해답이 없다는 점도 한계입니다.

예시: 아래는 벨만-포드 알고리즘의 구현을 **Java**와 **Python**으로 나타낸 것입니다. 그래프는 간선 리스트(Edge list) 형태로 표현하며, 음수 간선을 허용합니다. 알고리즘 수행 후 음수 사이클 검출까지 구현합니다.

```
// Java: 벨만-포드 알고리즘 구현
int N = 5;
List<int[]> edges = new ArrayList<>();
// 그래프 간선 추가: edges.add(new int[]{u, v, w});
int start = 0;
int INF = Integer.MAX_VALUE;
long[] dist = new long[N];  // 거리 배열 (int 범위 초과 가능성 대비 long 사용)
Arrays.fill(dist, INF);
dist[start] = 0;
// 1. V-1번 반복하여 간선 완화
for(int i = 0; i < N-1; i++){
  boolean updated = false;
  for(int[] e : edges){
    int u = e[0], v = e[1], w = e[2];
    if(dist[u] != INF && dist[v] > dist[u] + w){
      dist[v] = dist[u] + w;
      updated = true;
    }
  }
  if(!updated) break;  // 더 이상 갱신 없으면 조기 종료
}
// 2. 음수 사이클 존재 여부 체크
boolean negCycle = false;
for(int[] e : edges){
  int u = e[0], v = e[1], w = e[2];
  if(dist[u] != INF && dist[v] > dist[u] + w){
    negCycle = true;
    break;
  }
}
if(negCycle){
  System.out.println("음수 사이클이 존재합니다!");
} else {
  // 결과 출력: dist 배열에 start로부터 각 정점까지 최단 거리
```

```
    System.out.println(Arrays.toString(dist));
}
```

```python
# Python: 벨만-포드 알고리즘 구현
N = 5
edges = []
# 그래프 간선 추가: edges.append((u, v, w))
start = 0
INF = float('inf')
dist = [INF]*N
dist[start] = 0

# 1. V-1번 간선 완화 수행
for i in range(N-1):
    updated = False
    for u, v, w in edges:
        if dist[u] != INF and dist[v] > dist[u] + w:
            dist[v] = dist[u] + w
            updated = True
    if not updated:
        break

# 2. 음수 사이클 존재 검사
neg_cycle = False
for u, v, w in edges:
    if dist[u] != INF and dist[v] > dist[u] + w:
        neg_cycle = True
        break

if neg_cycle:
    print("음수 사이클이 존재합니다!")
else:
    print("최단 거리 결과:", dist)
```

위 코드에서 `edges` 리스트는 (출발 노드, 도착 노드, 가중치)로 이루어진 튜플들의 리스트입니다. 벨만-포드 알고리즘은 매우 단순하게, 매 반복마다 모든 간선을 일괄적으로 확인하며 `dist` 값을 업데이트합니다. 이러한 반복을 최대 V-1번 수행한 뒤에도 갱신이 발생하면, 이는 곧 거리값이 무한히 감소할 수 있는 **음수 사이클**이 존재함을 의미합니다.

활용 예시: 벨만-포드는 음수 간선을 허용하는 상황에서 유용하며, 예를 들어 **환율 차익** 문제(다중 통화를 환전하며 순환할 때 이득이 생기는지 여부 검사)에서 각 통화 간 환율 로그값(음수 간선으로 변환)을 그래프 간선 가중치로 삼아 음수 사이클을 탐지하는 데 사용될 수 있습니다. 또한 **최단 경로를 찾되 정확성을 요하는 경우**(예: 간선 가중치가 작고 음수가 포함된 그래프)에도 쓰입니다.

3. 플로이드-워셜 알고리즘

개요: 플로이드-워셜(Floyd-Warshall) 알고리즘은 그래프 내의 **모든 쌍의 최단 경로(All-Pairs Shortest Path)**를 한꺼번에 계산하는 알고리즘입니다. 즉, **모든 정점 쌍 (i, j)**에 대하여 i에서 j로 가는 최단 거리 값을 구합니다. **동적 계획법(DP)**에 기반한 알고리즘으로, 인접 행렬 형태의 그래프에 대해 작동하며, **가중치가 음수인 간선도 허용**하지만 **음수 사이클은 없다고 가정**합니다 [17] .

알고리즘 원리: 플로이드-워셜은 세 겹의 중첩 루프를 통해 점진적으로 최단 경로를 찾아냅니다. `dist[i][j]`를 정점 i에서 j로 가는 현재까지 찾은 최단 거리라고 할 때, **중간에 거쳐가는 정점의 집합**을 단계적으로 확대합니다. 구체적으로는, k=0부터 N-1까지 순차적으로 고려하면서 "정점 0..k-1만 거쳐가는 경로"와 "정점 0..k까지 거쳐가는 경로"를 비교합니다. 이를 코드로 표현하면 다음과 같습니다:

```
for k in range(N):
    for i in range(N):
        for j in range(N):
            if dist[i][j] > dist[i][k] + dist[k][j]:
                dist[i][j] = dist[i][k] + dist[k][j]
```

여기서 `k`가 **중간 정점**의 범위를 나타내며, 루프가 진행됨에 따라 점점 더 많은 정점을 중간에 사용할 수 있게 되어 최단 경로 거리가 갱신됩니다. 최종적으로 k=N-1 단계까지 거치면, **임의의 i->j 경로상 모든 정점을 활용**한 결과가 `dist[i][j]`에 저장됩니다.

시간 복잡도: 3중 루프를 사용하므로 **O(N^3)**의 시간 복잡도를 가집니다 [18] . 이로 인해 N이 큰 그래프에는 적용이 불가능하지만, N이 작거나 (수백 이하) 또는 **밀집 그래프**의 경우에는 경쟁력 있는 방법입니다. 특히 **그래프가 밀집**해서 간선이 매우 많을 때, 다익스트라를 모든 정점에 대해 수행하면 O(N * (M log N))이 되어 이것이 O(N^3)와 맞먹거나 더 느려질 수 있습니다. 플로이드-워셜은 **구현이 단순**하고, 또한 최단 경로뿐만 아니라 **경로 복원**(path reconstruction)을 위한 정보도 쉽게 유지할 수 있다는 장점이 있습니다. 예를 들어 `next[i][j]` 배열을 별도로 관리하여 i에서 j로의 최단 경로 상 다음 정점을 저장하면, 최단 경로 자체를 복원할 수 있습니다.

장점: 모든 쌍 최단경로를 구하므로, 한 번의 실행으로 그래프의 거리 정보를 완전히 파악할 수 있습니다. **음수 가중치**도 처리 가능하며, 음수 사이클 존재 여부도 **대각선 원소**를 확인하여 알 수 있습니다 (dist[i][i] 값이 음수가 되면 음수 사이클 존재). 구현이 3중 중첩 루프로 간단하여 **코드 작성이 용이**합니다.
단점: 시간 복잡도가 높아(O(N^3)), N이 조금만 커져도 계산량이 급증합니다. 예를 들어 N=500일 때 1.25억 번 연산, N=1000이면 10억 번 연산이 필요합니다. 또한 인접 행렬에 기반하므로 **공간 복잡도**가 O(N^2)로, N이 큰 경우 메모리 사용량도 무시할 수 없습니다.

예시: 아래는 플로이드-워셜 알고리즘의 **Java**와 **Python** 구현 예시입니다. `dist[i][j]`를 그래프의 인접 행렬로 사용하며, 초기에는 간선이 있으면 그 가중치로, 없으면 큰 값(INF)으로 채워둡니다. 자기 자신에 대한 거리는 0으로 초기화합니다.

```
// Java: 플로이드-워셜 알고리즘 구현
int N = 4;
int INF = 1000000000;  // 충분히 큰 값으로 무한대 대체
int[][] dist = new int[N][N];
// 그래프 초기화 (인접 행렬): dist[i][j] = 가중치, 없으면 INF
for(int i=0;i<N;i++){
  for(int j=0;j<N;j++){
    if(i == j) dist[i][j] = 0;
    // else dist[i][j] = ...;
  }
}
// Floyd-Warshall
for(int k=0; k<N; k++){
  for(int i=0; i<N; i++){
    for(int j=0; j<N; j++){
```

```
        if(dist[i][j] > dist[i][k] + dist[k][j]){
          dist[i][j] = dist[i][k] + dist[k][j];
        }
      }
    }
  }
}
// 결과: dist 행렬에 모든 i->j 최단 거리
```

```
# Python: 플로이드-워셜 알고리즘 구현
N = 4
INF = 10**9  # 충분히 큰 값으로 초기화
# 그래프 인접 행렬 초기화
dist = [[INF]*N for _ in range(N)]
for i in range(N):
    dist[i][i] = 0
# 예시 간선들 (i,j 간선 가중치 설정)
# dist[i][j] = w (간선 존재 시 가중치로 설정)

for k in range(N):
    for i in range(N):
        for j in range(N):
            if dist[i][j] > dist[i][k] + dist[k][j]:
                dist[i][j] = dist[i][k] + dist[k][j]
# 결과: dist[i][j]는 i에서 j로의 최단 경로 거리
```

위 구현은 간단한 3중 반복문으로 이루어져 있으며, Python의 경우 N이 크면 매우 느려질 수 있으므로 주의가 필요합니다. 일반적으로 플로이드-워셜은 **N<=400~500** 정도까지의 문제에 사용되며, 그 이상에서는 다익스트라 알고리즘을 N번 수행하는 **존슨(Johnson) 알고리즘** 등의 대안이 사용됩니다.

그래프 조건: 플로이드-워셜은 **방향 그래프**와 **무방향 그래프** 모두에 적용 가능하며, 음수 간선도 처리할 수 있지만 **음수 사이클**은 없다고 가정합니다 [17]. 만약 음수 사이클이 존재한다면, 위 언급한 대로 `dist[i][i]` 값이 음수로 내려가게 되므로 이를 탐지할 수 있습니다. 이러한 경우 최단 거리 자체는 의미가 없어지지만, 문제 유형에 따라 음수 사이클을 발견하는 것이 목적이 될 수도 있습니다.

활용 예시: 플로이드-워셜은 **그래프의 크기가 작고 모든 쌍 거리 정보가 필요한 경우** 쓰입니다. 예를 들어 **도시 지도**에서 모든 도시 간 최단 거리를 미리 계산해두는 경우, **APSP**(All-Pairs Shortest Path) 정보가 유용하며 플로이드-워셜로 구할 수 있습니다. 또한 **그래프 폐쇄성**(transitive closure)을 구하는 워셜 알고리즘, 혹은 **최대 가중치 경로**(widest path) 문제에도 응용될 수 있습니다 [19].

4. 크루스칼 알고리즘 (최소 신장 트리, DSU 활용)

개요: 크루스칼(Kruskal) 알고리즘은 **최소 신장 트리(MST)**를 찾는 대표적인 알고리즘입니다. 최소 신장 트리란 **가중치 그래프**에서 모든 정점을 연결하면서 **총 가중치 합이 최소**가 되는 트리 구조를 말합니다. 크루스칼 알고리즘은 **그리디 알고리즘**으로 분류되며, 간선을 하나씩 추가하면서 사이클을 없애는 방식으로 동작합니다. **간선의 가중치가 낮은 것부터 선택**해 나가며, 사이클이 생기는 간선은 건너뛰는 절차를 통해 MST를 완성합니다.

알고리즘 원리:
1. 그래프의 **모든 간선**을 가중치 기준 **오름차순 정렬**합니다.
2. 정렬된 간선 리스트를 순차적으로 확인하면서, 현재 간선이 **사이클을 형성하지 않는 경우** 해당 간선을 MST에 추가합

니다. 사이클을 형성하는 간선은 추가하지 않습니다.
3. 이때 사이클 여부를 빠르게 판단하기 위해 **DSU(Disjoint Set Union)** 자료구조를 활용합니다. 각 정점이 어느 연결 컴포넌트(집합)에 속해있는지 추적하여, 현재 간선의 두 끝점이 이미 같은 집합에 속하면 사이클이 생긴다고 보고 추가를 건너뜁니다.
4. 간선 선택을 계속하여 **(N-1)개의 간선**이 선택되면 알고리즘을 종료합니다 (N은 그래프의 정점 수, 연결 그래프 가정). MST가 완성되었으며, 선택된 간선들의 가중치 합이 최소가 됩니다.

시간 복잡도: 간선 정렬 단계가 **O(M log M)**이며, DSU 연산(Find/Union)이 거의 상수 시간에 일어나므로 전체 시간 복잡도는 **O(M log M)**으로 표현됩니다 [20] [21]. 간선 수 M에 비해 정점 수 N이 작을 경우, O(M log M)은 대략 O(M log N)과 비슷하게 간주하기도 합니다 [22]. 크루스칼 알고리즘은 **간선의 개수가 많은 그래프**에서도 정렬만 제대로 최적화된다면 충분히 빠르게 동작하며, 구현도 비교적 간단합니다.

장점: 알고리즘이 직관적이고 구현하기 쉽습니다. DSU 자료구조를 사용하면 사이클 판정을 효율적으로 할 수 있으며, 결과적으로 **MST를 구하는 속도가 빠르고** (거의 O(M log M)), 그래프가 여러 컴포넌트로 나뉘어 있어도 각 컴포넌트의 MST를 동시에 구할 수 있습니다.
단점: 간선 정렬을 해야 하므로, 간선 개수가 매우 많은 경우 정렬 비용이 부담될 수 있습니다. 또한 **동적 그래프**(간선 추가/삭제 등에 따라 MST를 유지해야 하는 경우)에서는 한 번 정렬로 끝나는 크루스칼 알고리즘을 직접 활용하기 어렵습니다.

예시: 아래는 크루스칼 알고리즘의 구현을 **Java**와 **Python**으로 나타낸 것입니다. 간선을 객체 혹은 튜플로 표현하고, 가중치 기준으로 정렬한 뒤 DSU를 이용해 사이클을 관리합니다. DSU 구현은 하단의 **6. DSU 자료구조**에서 별도로 다루지만, 여기서도 간단히 포함합니다.

```
// Java: 크루스칼 MST 알고리즘 구현
class  Edge implements Comparable<Edge> {
  int u, v, w;
  Edge(int u, int v, int w){ this.u=u; this.v=v; this.w=w; }
  @Override
  public int compareTo(Edge other){
    return Integer.compare(this.w, other.w);
  }
}
int N = 6;
List<Edge> edges = new ArrayList<>();
// edges.add(new Edge(u, v, w)); 형태로 간선 추가
// 1. 간선 리스트 정렬
Collections.sort(edges);
// 2. DSU 초기화
int[] parent = new int[N];
int[] rank = new int[N];
for(int i=0;i<N;i++){ parent[i]=i; rank[i]=0; }
function find(int x){
  if(parent[x] != x) parent[x] = find(parent[x]);
  return parent[x];
}
function union(int a, int b){
  int rootA = find(a);
  int rootB = find(b);
  if(rootA == rootB) return false;
  if(rank[rootA] < rank[rootB]) parent[rootA] = rootB;
```

```
  else if(rank[rootA] > rank[rootB]) parent[rootB] = rootA;
  else { parent[rootB] = rootA; rank[rootA]++; }
  return true;
}
// 3. 간선 선택
int mstWeight = 0;
List<Edge> mstEdges = new ArrayList<>();
for(Edge e : edges){
  if(union(e.u, e.v)){  // 두 정점이 다른 집합일 경우만 선택
    mstWeight += e.w;
    mstEdges.add(e);
  }
}
// mstWeight 가 MST 전체 가중치, mstEdges 리스트가 MST의 간선 집합
```

```
# Python: 크루스칼 MST 알고리즘 구현
N = 6
edges = []  # (w, u, v) 형태로 간선 저장 (정렬 편의를 위해 w를 첫 요소)
# edges.append((w, u, v)) 형태로 간선 추가
# 1. 간선 리스트 정렬 (가중치 w 기준)
edges.sort(key=lambda x: x[0])
# 2. DSU 초기화
parent = list(range(N))
rank = [0]*N
def find(x):
    if parent[x] != x:
        parent[x] = find(parent[x])
    return parent[x]
def union(a, b):
    rootA = find(a)
    rootB = find(b)
    if rootA == rootB:
        return False
    if rank[rootA] < rank[rootB]:
        parent[rootA] = rootB
    elif rank[rootA] > rank[rootB]:
        parent[rootB] = rootA
    else:
        parent[rootB] = rootA
        rank[rootA] += 1
    return True

# 3. 간선 선택
mst_weight = 0
mst_edges = []
for w, u, v in edges:
    if union(u, v):
        mst_weight += w
        mst_edges.append((u, v, w))
```

```
# mst_weight: MST 총 가중치, mst_edges: MST를 구성하는 간선 목록
```

위 구현에서 `edges` 리스트는 (가중치, 정점1, 정점2)로 구성되어 정렬 및 처리를 합니다. 정렬 이후 가장 작은 가중치 간선부터 차례로 살펴보며, `union(u, v)`를 통해 서로 다른 컴포넌트를 연결할 때만 간선을 채택합니다. DSU를 활용해 사이클 여부를 상수시간에 가깝게 판별하므로, 전체 효율이 높아집니다. **결과적으로** `mst_edges`**에 선택된 간선들이 최소 신장 트리를 구성**하며, `mst_weight`는 그 합계 가중치입니다.

그래프 조건: MST 알고리즘은 **무향 연결 그래프**에 주로 적용됩니다. 방향 그래프의 경우에도 (사실상 방향을 무시하고) MST 개념을 적용할 수 있지만 일반적이진 않습니다. 또한 간선 가중치에 음수가 포함되어도 최소 신장 트리를 찾는 데 문제는 없지만, 음수 간선이 있다면 **모든 음수 간선을 우선 선택**하게 되므로 결과에 큰 영향을 미칩니다.

비교 - Kruskal vs Prim: 크루스칼 알고리즘은 **간선 중심** 접근 방식인 반면, 프림 알고리즘은 **정점 중심** 접근 방식입니다. **밀집 그래프**의 경우 프림 알고리즘이 이론적으로 유리할 수 있고 (인접 행렬 사용 시 O(N^2)), **희소 그래프**의 경우 크루스칼이나 우선순위 큐 기반 프림(O(M log N))이 모두 효율적입니다 [23] [24]. 구현 난이도는 두 방식 모두 쉬운 편이며, DSU 구현에 익숙하다면 크루스칼이, 우선순위 큐 사용에 익숙하다면 프림이 편할 수 있습니다.

5. 프림 알고리즘 (최소 신장 트리)

개요: 프림(Prim) 알고리즘 역시 최소 신장 트리를 찾는 알고리즘으로, 크루스칼과 함께 널리 사용됩니다. **하나의 정점에서 시작**하여 신장 트리를 단계적으로 확장해 나가는 방식이 특징입니다. 초기에는 한 정점만 포함된 트리에서 출발하여, **항상 현재 형성된 트리와 인접한 간선 중 가장 가중치가 작은 간선을 선택**하여 새로운 정점을 추가하는 과정을 N-1번 반복합니다.

알고리즘 원리:
1. 임의의 시작 정점을 선택하여 MST 집합에 추가합니다.
2. 현재 MST에 속한 정점들과 **인접한 모든 간선** 중에서 **가중치가 가장 작은 간선**을 선택합니다. 이 간선이 연결하는 **MST 밖의 정점**을 새로운 정점으로 추가합니다.
3. 다시 이 신규 정점을 MST에 포함시키고, 다음 간선을 선택합니다. 이때 항상 "현재 MST 내부 정점 vs 외부 정점"을 연결하는 최소 가중치 간선을 선택해야 합니다.
4. 이 과정을 **N-1번** (총 정점 수 N이면 MST는 N-1개의 간선으로 구성) 반복하여 MST를 완성합니다.

프림 알고리즘은 다익스트라와 유사하게 **우선순위 큐**를 사용하여 효율을 높일 수 있습니다. 현재 트리와 연결된 간선들을 우선순위 큐에 넣고 최소 값을 뽑아오는 방식으로 구현하면, 크루스칼과 같은 O(M log N) 복잡도를 얻을 수 있습니다 [3]. 한편, 간선이 많은 경우 인접 행렬을 사용한 구현으로 O(N^2)에 동작시킬 수도 있습니다 [23].

시간 복잡도: - **인접 행렬 + 단순 선택**으로 구현하면 O(N^2)입니다. 각 단계마다 N개의 정점을 검사하여 최소 간선을 찾고 N-1번 반복하므로 N^2. 이 방식은 **밀집 그래프(N에 비해 M이 큰 경우)**에서 효율적이며, 우선순위 큐 overhead를 줄일 수 있습니다. - **인접 리스트 + 우선순위 큐**로 구현하면 O(M log N)으로, 일반적인 희소 그래프에서 효율적입니다 [3]. BFS와 유사하지만 우선순위 큐를 사용한다는 점에서 다익스트라와 매우 유사한 구조를 가집니다. 실제로 가중치가 없는 그래프에서 프림 알고리즘은 BFS와 동일한 순서로 노드를 추가하며, 가중치가 있는 경우 다익스트라와 유사하게 가장 가까운 정점을 추가합니다.

장점: 구현이 비교적 쉽고, **임의의 시작점에서 시작할 수 있어** 부분 MST를 구하다가 확장하는 형태로 생각하기 좋습니다. 다익스트라를 구현해봤다면 프림도 거의 비슷한 구조이므로 이해하기 쉽습니다. 또한 한 번에 하나의 정점씩 추가하므로, **중간 과정에서도 현재까지의 최소 연결 그래프 상태**를 항상 유지합니다.
단점: 크루스칼과 마찬가지로 기본적인 형태는 정적 그래프에 대해서만 고려합니다. 우선순위 큐 사용 시, 크루스칼에 비해 메모리 사용이 조금 늘 수 있습니다 (간선 리스트 외에 PQ에 간선을 저장해야 함).

예시: 아래는 프림 알고리즘의 **Java**와 **Python** 구현 예시입니다. 우선순위 큐를 사용한 방식으로, 임의의 정점 0부터 시작해서 MST를 만들어갑니다.

```java
// Java: 프림 MST 알고리즘 구현 (우선순위 큐 사용)
int N = 6;
List<int[]>[] graph = new ArrayList[N];
for(int i=0;i<N;i++){ graph[i] = new ArrayList<>(); }
// 그래프 초기화: graph[u].add(new int[]{v, w});
boolean[] visited = new boolean[N];
PriorityQueue<int[]> pq = new PriorityQueue<>(Comparator.comparingInt(a -> a[0]));
// {간선 가중치, 도착 정점}으로 저장
int start = 0;
visited[start] = true;
for(int[] edge : graph[start]){
  pq.offer(new int[]{ edge[1], edge[0] });  // {w, v}
}
int mstWeight = 0;
List<int[]> mstEdges = new ArrayList<>();
int edgesUsed = 0;
while(!pq.isEmpty() && edgesUsed < N-1){
  int[] cur = pq.poll();
  int w = cur[0], v = cur[1];
  if(visited[v]) continue;
  // 간선 (현재 MST 연결 -> v) 선택
  visited[v] = true;
  mstWeight += w;
  edgesUsed++;
  mstEdges.add(new int[]{ v, w });
  // 새로 추가된 정점 v에 연결된 간선들을 PQ에 추가
  for(int[] edge : graph[v]){
    if(!visited[edge[0]]){
      pq.offer(new int[]{ edge[1], edge[0] });
    }
  }
}
// mstWeight: MST 가중치 합, mstEdges: MST에 선택된 간선 정보
```

```python
# Python: 프림 MST 알고리즘 구현 (우선순위 큐 사용)
import heapq
N = 6
graph = [[] for _ in range(N)]
# 그래프 초기화: graph[u].append((v, w))
visited = [False]*N
start = 0
visited[start] = True
pq = []
for v, w in graph[start]:
    heapq.heappush(pq, (w, start, v))  # (가중치, 시작정점, 도착정점)
mst_weight = 0
```

```
mst_edges = []
edges_used = 0
while pq and edges_used < N-1:
    w, u, v = heapq.heappop(pq)
    if visited[v]:
        continue
    # 간선 (u->v, 가중치 w) 선택
    visited[v] = True
    mst_weight += w
    mst_edges.append((u, v, w))
    edges_used += 1
    # 새로 추가된 정점 v로부터 이어지는 간선들을 큐에 추가
    for nv, nw in graph[v]:
        if not visited[nv]:
            heapq.heappush(pq, (nw, v, nv))
# mst_weight: MST 가중치 합, mst_edges: MST 간선 목록
```

프림 알고리즘 코드에서, `visited` 배열로 MST에 포함된 정점을 관리하고, 우선순위 큐 `pq`에는 현재 MST와 인접한 간선들이 들어갑니다. 가장 작은 가중치 간선을 꺼내서 연결되지 않은 정점을 합류시키는 과정이 반복됩니다. **프림 알고리즘은 연결 그래프가 아닌 경우**에도 동작은 하지만, 모든 정점을 연결할 수 없으므로 MST를 찾을 수 없습니다. 일반적으로 MST 문제에서는 그래프가 **하나의 연결 컴포넌트**라고 가정합니다 (아니라면 각 컴포넌트별로 MST를 구할 수도 있습니다).

요약: 크루스칼과 프림 알고리즘 모두 MST를 찾는 데 유용하며, **시간 복잡도는 대동소이**합니다 [24]. 구현의 난이도나 익숙함에 따라 편한 방법을 선택하면 되며, 두 방법 모두 알아두면 좋습니다. 코딩 테스트에서는 간선 수가 적당히 많을 경우 크루스칼, 매우 많을 경우 프림(인접 행렬 O(N^2) 또는 PQ O(M log N))을 권장하는 식으로 문제 조건에 따라 선택할 수 있습니다.

6. DSU (Disjoint Set Union, 서로소 집합) 자료구조

개요: DSU(Disjoint Set Union) 자료구조, 흔히 **Union-Find**로 불리는 구조는 **상호 배타적인 여러 집합들을 효율적으로 관리**하기 위한 자료구조입니다. 주요 연산은 **합치기(Union)**와 **찾기(Find)** 두 가지이며, 이를 통해 원소들이 어떤 그룹에 속하는지, 두 원소가 같은 그룹인지 등을 빠르게 판단할 수 있습니다. 그래프 알고리즘에서는 주로 **연결 여부**를 체크하거나 **사이클**을 검사하는 용도로 쓰이는데, 앞서 소개한 크루스칼 알고리즘에서 **사이클 판정**을 위해 사용되었습니다.

동작 원리: DSU는 초기에는 모든 원소가 자기 자신만을 원소로 갖는 **단일 원소 집합**으로 시작합니다. 구조적으로 각 집합을 **트리 형태**로 표현하며, 각 원소는 **부모 포인터**를 가집니다. 트리의 **루트(root)** 노드가 그 집합의 대표자(leader) 역할을 합니다. 기본 연산은 다음과 같습니다: - **Find(x):** 원소 x가 속한 집합의 대표자를 찾는 연산입니다. 구현은 x에서 시작하여 부모 포인터를 따라가 최종 루트 노드를 찾으면 됩니다. 경로상의 노드들을 직접 루트에 연결시키는 **경로 압축(path compression)** 기법을 적용하여 이후 Find 연산을 더욱 빠르게 합니다. - **Union(a, b):** 원소 a가 속한 집합과 원소 b가 속한 집합을 합치는 연산입니다. 먼저 각 원소의 대표자(Find 결과)를 찾은 뒤, 서로 다르면 한 쪽을 다른 쪽에 붙여 하나의 집합으로 만듭니다. 이때 트리 높이를 최소화하기 위해 **랭크(rank) 또는 사이즈** 개념을 사용하여 항상 높이가 낮은 트리를 높은 트리에 붙이는 식으로 합칩니다 (union by rank/size).

시간 복잡도: 경로 압축과 랭크 최적화를 적용한 DSU의 Find와 Union 연산은 **아주 작은 상수 시간에 가까운** 성능을 보입니다. 엄밀히는 **α(N)** (알파 함수, 역 아커만 함수) 정도의 복잡도를 가지는데, α(N)은 N이 현실적인 수십억 이상이어도 5 이하의 값을 가지는 매우 천천히 증가하는 함수입니다. 따라서 DSU의 **평균 연산 시간은 거의 O(1)**으로 간주해도 무방합니다 [25]. 이러한 뛰어난 성능 덕분에 DSU는 그래프의 연결성 판정이나 군집화 문제 등에 광범위하게 쓰입니다.

장점: 구현이 쉽고(배열로 부모 관리), 연산 속도가 거의 상수 시간이어서 **대규모 입력**에도 효율적으로 동작합니다. 자료구조가 차지하는 메모리도 O(N)으로 비교적 적습니다.
단점: 트리 형태로 집합을 표현하므로, 집합 내 원소들 간 직접적인 관계 정보를 제공하지는 않습니다. 오직 대표자 비교를 통한 같은 집합 여부 확인만 가능하며, 계층적인 구조나 정렬이 필요한 경우에는 활용할 수 없습니다. 또한 동적인 원소 추가/삭제의 경우 기본 DSU 구조에서는 삭제 연산이 지원되지 않습니다.

예시: 아래는 DSU 자료구조의 **Java**와 **Python** 구현 예시입니다. 경로 압축과 랭크 최적화를 적용한 표준 구현을 보여줍니다.

```
// Java: DSU (Union-Find) 구현
class  UnionFind {
  int[] parent;
  int[] rank;
  UnionFind(int n){
    parent = new int[n];
    rank = new int[n];
    for(int i=0;i<n;i++){
      parent[i] = i;
      rank[i] = 0;
    }
  }
  int find(int x){
    if(parent[x] != x){
      parent[x] = find(parent[x]);  // 경로 압축
    }
    return parent[x];
  }
  boolean union(int a, int b){
    int rootA = find(a);
    int rootB = find(b);
    if(rootA == rootB) return false;
    if(rank[rootA] < rank[rootB]){
      parent[rootA] = rootB;
    } else if(rank[rootA] > rank[rootB]){
      parent[rootB] = rootA;
    } else {
      parent[rootB] = rootA;
      rank[rootA]++;
    }
    return true;
  }
}
// 사용 예시:
UnionFind dsu = new UnionFind(10);
dsu.union(1, 2);
dsu.union(2, 3);
System.out.println(dsu.find(3) == dsu.find(1));  // true, 1과 3은 같은 집합
```

```python
# Python: DSU (Union-Find) 구현
class UnionFind:
    def __init__(self, n):
        self.parent = list(range(n))
        self.rank = [0]*n
    def find(self, x):
        if self.parent[x] != x:
            self.parent[x] = self.find(self.parent[x])  # 경로 압축
        return self.parent[x]
    def union(self, a, b):
        rootA = self.find(a)
        rootB = self.find(b)
        if rootA == rootB:
            return False
        if self.rank[rootA] < self.rank[rootB]:
            self.parent[rootA] = rootB
        elif self.rank[rootA] > self.rank[rootB]:
            self.parent[rootB] = rootA
        else:
            self.parent[rootB] = rootA
            self.rank[rootA] += 1
        return True

# 사용 예시:
uf = UnionFind(10)
uf.union(1, 2)
uf.union(2, 3)
print(uf.find(3) == uf.find(1))  # True, 1과 3이 같은 집합에 속함
```

위 구현에서 `parent` 배열은 각 원소의 부모를 가리키며, `rank` 배열은 각 집합 트리의 대략적인 높이를 나타내어 합칠 때 사용됩니다. `find(x)`는 재귀적으로 부모를 따라가 최종 루트를 찾으며, 돌아오는 길에 부모 포인터를 루트로 직접 갱신하여 경로를 압축합니다. `union(a, b)`는 각각 a와 b의 루트를 찾은 뒤, rank 값을 비교하여 높이가 더 낮은 트리를 높은 트리에 붙입니다. 만약 rank가 같다면 한쪽을 루트로 하고 rank를 1 증가시킵니다.

활용 예시: DSU는 **그래프의 연결성 문제**에서 자주 활용됩니다. 예를 들어 **네트워크 연결 문제**(여러 컴퓨터가 주어지고 연결 여부를 묻는), **도시 분할 계획**(도로를 없애 특정 마을들을 분리), **사이클 판정**, **섬의 개수** 등의 문제에서 쓰일 수 있습니다. 또한 **이미지 처리**에서 픽셀 군집을 구분하거나, **게임**에서 서로 관계 맺기(길드 등)를 구현할 때도 DSU를 사용할 수 있습니다. 코딩 테스트에서는 주로 **MST 구현**(크루스칼)이나 **무방향 그래프 사이클 검사**, **서로소 그룹 쿼리 처리** 등에 DSU 활용 문제가 출제되는 편입니다.

DSU를 잘 활용하면 **그래프 탐색으로 해결할 문제를 더 간단히 해결**하는 경우도 있습니다. 예를 들어 매 번 연결성을 확인하기 위해 DFS/BFS를 하면 O(N+M) 시간이 들지만, DSU로 미리 군집화해두면 Find 연산으로 거의 O(1)에 답을 얻을 수 있는 식입니다. 이렇듯 상황에 맞게 DSU를 사용하면 효율적인 해결이 가능하므로, 그래프 심화 단계에서 반드시 알아두어야 할 도구입니다.

7. 마무리 및 다음 주 예고

이번 주 9주차에서는 **그래프 알고리즘 심화 주제**들을 다루며, 최단 경로 알고리즘 (다익스트라, 벨만-포드, 플로이드-워셜)과 최소 신장 트리 알고리즘 (크루스칼, 프림), 그리고 중요한 자료구조인 DSU까지 폭넓게 학습했습니다. 다루는 내

용이 많고 복잡했지만, 각각의 알고리즘들은 **그래프 문제 해결의 핵심 기법**들로서 competitive programming 뿐 아니라 실제 응용 환경에서도 활용도가 높습니다. 이번에 공부한 알고리즘들을 정리하면 다음과 같습니다:

- **다익스트라 알고리즘:** 양의 가중치 그래프의 단일 출발 최단 경로 알고리즘으로, 우선순위 큐를 사용한 구현이 효율적입니다. 최근 연구를 통해 이론적 개선도 있었지만, 일반적으로 O(M log N) 알고리즘으로 널리 사용됩니다.
- **벨만-포드 알고리즘:** 음수 가중치 처리와 음수 사이클 검출이 가능한 최단 경로 알고리즘입니다. 다익스트라에 비해 느리지만, 적용 범위가 더 넓습니다.
- **플로이드-워셜 알고리즘:** 동적 계획법으로 모든 쌍 최단 경로를 구하는 알고리즘입니다. O(N^3)으로 느리지만 구현이 쉬워, 작은 그래프나 밀집 그래프에 사용됩니다.
- **크루스칼 & 프림 (MST):** 그래프를 최소 비용으로 연결하는 간선 집합을 찾는 알고리즘들입니다. 둘 다 그리디 알고리즘으로서, 서로 다른 접근 방식이지만 결과를 동일하게 도출합니다. 크루스칼은 간선 정렬 + 사이클 판정(DSU), 프림은 정점 확장 + 우선순위 큐가 핵심입니다.
- **DSU (Union-Find):** 그래프의 연결성 및 군집을 효율적으로 관리하는 자료구조로, MST 뿐 아니라 다양한 그래프 문제에서 활용됩니다. Find/Union 연산이 거의 상수시간으로 매우 효율적입니다.

각 알고리즘의 **원리**와 **제약 조건**(예: 음수 간선 여부), **시간 복잡도와 특징**, **예제 코드** 등을 살펴보았으니, 꼭 한 번씩 직접 구현해 보고 작은 그래프 예시로 출력 결과를 확인해보길 권장합니다. 그래프 심화 알고리즘들은 문제에 바로 적용하기 전에 개념을 확실히 이해해야 실수를 줄일 수 있으므로, 오늘 학습한 내용을 토대로 충분히 복습하시기 바랍니다.

다음 주는 **그리디 알고리즘**을 학습하며 탐욕적 선택 전략과 다양한 유형의 문제를 풀어봅니다. 수고하셨습니다! [6] [1] [13] [17] [24] [25]

---

[1] [2] Dijkstra's algorithm - Wikipedia
https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm

[3] [23] Difference between Prim's and Kruskal's algorithm for MST
https://www.geeksforgeeks.org/dsa/difference-between-prims-and-kruskals-algorithm-for-mst/

[4] [5] [6] [7] [8] [9] [10] [11] [12] Tsinghua University Breaks a 65-Year Limit: A Faster Alternative to Dijkstra's Algorithm | by Vaibhav Verma | Aug, 2025 | Medium
https://medium.com/@vverma4313/tsinghua-university-breaks-a-65-year-limit-a-faster-alternative-to-dijkstras-algorithm-e2f42a608369

[13] [14] [15] [16] Bellman–Ford algorithm - Wikipedia
https://en.wikipedia.org/wiki/Bellman%E2%80%93Ford_algorithm

[17] [18] [19] Floyd–Warshall algorithm - Wikipedia
https://en.wikipedia.org/wiki/Floyd%E2%80%93Warshall_algorithm

[20] Kruskal's algorithm - Wikipedia
https://en.wikipedia.org/wiki/Kruskal%27s_algorithm

[21] [24] When should I use Kruskal as opposed to Prim (and vice versa)?
https://stackoverflow.com/questions/1195872/when-should-i-use-kruskal-as-opposed-to-prim-and-vice-versa

[22] About Kruskal's algorithm's time complexity - Reddit
https://www.reddit.com/r/algorithms/comments/pf63mm/about_kruskals_algorithms_time_complexity/

[25] Disjoint Set Union - Algorithms for Competitive Programming
https://cp-algorithms.com/data_structures/disjoint_set_union.html