1. 해시테이블

- **해시테이블(hash table):** 키-주소 매핑에 의해 구현된 사전 ADT
  - **예:** 컴파일러의 심볼 테이블, 환경변수들의 레지스트리
  - **구성:** 버켓 배열 + 해시함수
  - **성능:** 탐색, 삽입, 삭제 - 최악 O(n), 기대 O(1)

2. 버켓 배열

- 크기 M의 배열 A
  - 각 셀을 버켓(키-원소 쌍을 담는 그릇)으로 봄 - 슬롯(slot)이라고도 함
  - 정수 M은 배열의 용량
  - 키 k를 가진 원소 e는 버켓 A[k]에 삽입
  - 사전에 없는 키의 버켓은 NoSuchKey 객체를 가짐
  - 키가 유일한 정수이며 [0, M-1] 범위에 잘 분포: 탐색, 삽입, 삭제에 O(1) 최악 시간 소요
  - **결함:** O(n) 공간 사용 (M이 n에 비해 매우 크면 공간 낭비), 키가 [0, M-1] 범위의 유일한 정수여야 함(비현실적)

3. 해시함수

- 키를 [0, M-1] 범위의 정수로 매핑하는 함수
  - **예:** h(x) = x % M (x는 정수 키, M은 배열 크기)
  - h(x): 키 x의 해시값(hash value)
  - 해시테이블 구성요소: 해시함수 h, 크기 M의 배열(테이블)
  - 목표: 항목 (k, e)를 첨자 i = h(k)에 저장
- 보통 두 함수의 복합체
  - **해시코드맵(hash code map) h1:** keys → integers
  - **압축맵(compression map) h2:** integers → [0, M-1]
  - h(k) = h2(h1(k))
  - **좋은 해시함수 조건:** 키들을 무작위하게 분산, 계산이 빠르고 쉬움(상수시간)

4. 해시함수 예시 및 해시코드맵

- **학번 → 마지막 4자리 수 → 방번호[0, 2]**
  - **식별자 → 문자합 → 심볼 테이블 행번호[0, 27]**
  - **메모리 주소:** 키 객체의 메모리 주소를 정수로 재해석 (모든 Java 객체의 기본 해시코드), 수치 또는 문자열 키에는 적용 곤란
  - **정수 캐스트:** 키의 비트값을 정수로 재해석, 정수형에 할당된 비트 수를 초과하지 않는 길이의 키에 적당 (Java의 byte, short, int, float)
  - **요소합:** 키의 비트들을 고정길이 요소로 분할 후 합산 (overflow 무시), 정수형에 할당된 비트 수 이상의 고정길이 수치 키에 적당 (Java의 long, double), 문자열 키에는 부적당
  - **다항 누적:** 요소합과 마찬가지로 분할, 고정값 z를 사용하여 위치에 따른 계산 부과, 문자열에 적당 (예: z = 33, 50,000개 영단어에 6회 충돌)

5. 압축맵

- **나누기:** h2(k) = |k| % M (M은 소수로 선택)
  - **승합제(MAD):** h2(k) = |ak + b| % M (a, b는 음이 아닌 정수, a % M ≠ 0)

6. 충돌 해결

- **충돌(collision):** 두 개 이상의 원소들이 동일한 셀로 매핑되는 현상
- **충돌 발생 조건:** 상이한 키 k1과 k2에 대해 h(k1) = h(k2)
- **충돌 해결 전략:** 일관된 전략 필요

7. 분리연쇄법

- **개념:** 각 버켓 A[i]는 리스트 Li에 대한 참조를 저장. Li는 해시함수가 버켓 A[i]로 매핑한 모든 항목들을 저장하는 리스트
- **구현:** 무순서 리스트 또는 기록 파일 방식 사용
- **장점:** 단순하고 빠름
- **단점:** 테이블 외부에 추가적인 저장 공간 필요
- **예시:** h(k) = k % M, 키: 25, 13, 16, 15, 7, 28, 31, 20, 1 (삽입 순서)
- **리스트 삽입 위치:** 리스트의 테일 포인터를 별도로 유지하지 않는 경우, 리스트의 맨 앞에 삽입하는 것이 유리
- **알고리즘:**
  - **initBucketArray():** 버켓 배열 초기화 (각 버켓을 빈 리스트로 설정)
  - **findElement(k):** 키 k를 가진 원소 탐색
  - **insertItem(k, e):** 키 k, 값 e를 가진 원소 삽입
  - **removeElement(k):** 키 k를 가진 원소 삭제

8. 개방주소법

- **개념:** 충돌 항목을 테이블의 다른 셀에 저장
- **장점:** 분리연쇄법에 비해 공간 사용 절약
- **단점:** 삭제 어려움, 사전 항목들의 군집화(clustering)

9. 선형 조사법

- **개념:** 충돌 항목을 (원형으로) 바로 다음의 비어 있는 테이블 셀에 저장
- **조사 순서:** A[(h(k) + i) % M], i = 0, 1, 2, ...
- **문제점:** 1차 군집화(primary clustering) 발생
- **예시:** h(k) = k % M, 키: 25, 13, 16, 15, 7, 28, 31, 20, 1, 38 (삽입 순서)

10. 2차 조사법

- **개념:** 다음 순서에 의해 버켓을 조사: A[(h(k) + i") % M], i = 0, 1, 2, ...
- **문제점:** 2차 군집화(secondary clustering) 발생, M이 소수가 아니거나 버켓 배열이 반 이상 차면 비어 있는 버켓이 남아 있더라도 찾지 못할 수 있음

- **예시:** h(k) = k % M, f(i) = i"), 키: 25, 13, 16, 15, 7, 28, 31, 20, 1, 38 (삽입 순서)

11. 이중 해싱

- **개념:** 두 번째 해시 함수 h′를 사용하여 조사 순서 결정: A[(h(k) + i * h′(k)) % M], i = 0, 1, 2, ...
- **장점:** 군집화 최소화, h′(k)는 0이 되면 안 됨, h′(k)와 M은 서로소(relative prime)여야 최선의 결과
- **조사 횟수:** d1 * M = d2 * h′(k) 이면 d2개의 조사만 시도
- **h′(k) 선택:** q - (k % q) 또는 1 + (k % q) (q < M은 소수, M도 소수)
- **예시:** h(k) = k % M, h′(k) = 11 - (k % 11), 키: 25, 13, 16, 15, 7, 28, 31, 20, 1, 38 (삽입 순서)

12. 개방 주소법 알고리즘

- **insertItem(k, e):** 키 k, 값 e를 가진 원소 삽입
- **findElement(k):** 키 k를 가진 원소 탐색
- **getNextBucket(v, i):** 다음 버켓 계산 (선형 조사, 2차 조사, 이중 해싱에 따라 다름)
- **initBucketArray():** 버켓 배열 초기화
- **isEmpty(b):** 버켓 b가 비어있는지 확인

13. 개방 주소법에서의 갱신

- **비활성화 전략:** 삭제된 셀을 inactive로 표시하여 재사용
- **상태:** empty, active, inactive
- **insertItem(k, e):** 비어 있거나 비활성인 셀에 삽입 후 활성화
- **removeElement(k):** 활성 셀의 항목을 비활성화
- **findElement(k):** 활성 셀에서 탐색

14. 적재율

- **적재율(load factor), α = n/M:** n은 원소 개수, M은 버켓 개수
- **적재율 유지:** 낮게 유지 (1 아래로)
- **기대 실행 시간:** O(α) (좋은 해시 함수 사용 시)
- **분리 연쇄법:** α > 1 가능하지만 비효율적, α ≤ 1 (0.75 미만이면 더 좋음)일 때 O(1) 기대 실행 시간
- **개방 주소법:** α ≤ 1, α > 0.5이면 군집화 가능성 높음, α ≤ 0.5일 때 O(1) 기대 실행 시간

15. 재해싱

- **목적:** 적재율을 상수 이하로 유지 (보통 0.75)
- **시기:** 적재율 최적치 초과, 삽입 실패, 많은 비활성 셀로 성능 저하 시
- **단계:** 버켓 배열 크기 증가 (두 배 정도, 소수로 설정), 압축 맵 수정, 기존 원소들을 새 테이블에 삽입

16. 해싱의 성능

- **최악의 경우:** O(n) (모든 키가 충돌)
- **성능 좌우 요소:** 적재율 (load factor), a = n/N
- **개방 주소법 삽입 기대 조사 횟수:** 1/(1 - a) (해시값들을 난수로 가정)
- **기대 실행 시간:** O(1) (적재율이 1에 가깝지 않다면)

17. 응용

- 소규모 데이터베이스
- 컴파일러
- 브라우저 캐시

18. 해시테이블

- **응용문제: 연결리스트 동일성**: 두 수들의 집합 S와 T가 단일연결리스트로 구현되어 있고, 헤드노드만 접근 가능할 때, S = T인지 결정하는 O(min(|S|, |T|)) 기대시간 알고리즘
  - 두 집합의 크기 비교: 크기가 다르면 동일하지 않음
  - 크기가 같다면, 해시테이블 이용하여 원소 비교: S의 원소들을 해시테이블에 삽입 후, T의 원소들을 탐색하며 존재 여부 확인
  - 시간복잡도: O(min(|S|, |T|)) 기대시간
- **비활성화 방식 삭제**: 개방주소법에서 비활성화 방식 삭제를 위한 알고리즘
  - `findElement(k)`: 키 k를 갖는 원소 탐색
  - `insertItem(k, e)`: 키 k, 값 e를 갖는 원소 삽입
  - `removeElement(k)`: 키 k를 갖는 원소 삭제 (비활성화)
  - `deactivate(b)`: 버켓 b 비활성화
  - `activate(b)`: 버켓 b 활성화
  - `inactive(b)`: 버켓 b가 비활성인지 확인
  - `active(b)`: 버켓 b가 활성인지 확인

19. 그래프

- 그래프 ADT
  - **그래프 정의**: 정점(vertex)들의 집합 V와 간선(edge)들의 집합 E로 구성
    - **간선의 유형**: 방향 간선(directed edge), 무방향 간선(undirected edge)
    - **그래프 유형**: 방향 그래프(directed graph), 무방향 그래프(undirected graph)
- 그래프 용어
  - **간선의 끝점**: 간선에 연결된 정점
    - **정점의 부착 간선**: 특정 정점에 연결된 간선
    - **인접 정점**: 간선으로 연결된 정점들
    - **정점의 차수**: 정점에 연결된 간선의 수
    - **병렬 간선**: 같은 두 정점을 연결하는 여러 개의 간선

- **루프**: 자기 자신을 연결하는 간선
- **경로**: 정점과 간선의 교대 열
- **단순 경로**: 모든 정점과 간선이 유일한 경로
- **싸이클**: 정점과 간선의 원형 열
- **단순 싸이클**: 모든 정점과 간선이 유일한 싸이클

- 그래프 속성
  - **속성 1**: $\sum_{v \in V} deg(v) = 2m$ (각 간선은 두 번 세어짐)
    - **속성 2**: 루프와 병렬 간선이 없는 무방향 그래프에서 $m \le \frac{n(n-1)}{2}$

- 그래프의 종류
  - **부그래프**: 정점과 간선의 부분집합으로 구성된 그래프
    - **신장 부그래프**: 모든 정점을 포함하는 부그래프
    - **연결 그래프**: 모든 정점쌍 사이에 경로가 존재하는 그래프
    - **연결 요소**: 최대 연결 부그래프
    - **희소 그래프**: 간선 수가 적은 그래프
    - **밀집 그래프**: 간선 수가 많은 그래프
    - **트리**: 연결되고 싸이클이 없는 무방향 그래프
    - **숲**: 싸이클이 없는 무방향 그래프 (연결 요소는 트리)

- 그래프 ADT 메소드
  - **공통 메소드**: `numVertices()`, `numEdges()`, `vertices()`, `edges()`
    - **접근 메소드**: `aVertex()`
    - **질의 메소드**: `isDirected(e)`
    - **반복 메소드**: `directedEdges()`, `unDirectedEdges()`
    - **갱신 메소드**: `insertVertex(o)`, `removeVertex(v)`, `removeEdge(e)`
    - **무방향 그래프 추가 메소드**: `deg(v)`, `opposite(v, e)`, `areAdjacent(v, w)`, `endVertices(e)`, `adjacentVertices(v)`, `incidentEdges(v)`, `insertEdge(v, w, o)`
    - **방향 그래프 추가 메소드**: `origin(e)`, `destination(e)`, `inDegree(v)`, `outDegree(v)`, `inIncidentEdges(v)`, `outIncidentEdges(v)`, `inAdjacentVertices(v)`, `outAdjacentVertices(v)`, `insertDirectedEdge(v, w, o)`, `makeUndirected(e)`, `reverseDirection(e)`

20. 그래프 구현

- **간선 리스트(edge list) 구조**: 정점과 간선에 대한 포인터 리스트
- **인접 리스트(adjacency list) 구조**: 간선 리스트 구조 + 각 정점에 대한 부착 리스트
- **인접 행렬(adjacency matrix) 구조**: 간선 리스트 구조 + 정점에 대한 정수 키(첨자) + n x n 배열(인접 정점 쌍에 대한 간선 노드 참조, 비인접 정점 쌍에 대한 null 정보)
- **간선 리스트 구조**: 정점 노드들에 대한 포인터 리스트와 간선 노드들에 대한 포인터 리스트로 구성. 정점 노드는 원소를 가지고, 간선 노드는 원소, 시점 노드, 종점 노드를 가짐
- **인접 리스트 구조**: 각 정점에 대한 부착 리스트를 추가하여 각 정점의 부착 간선들을 간선 노드에 대한 참조들의 리스트로 표시
- **인접 행렬 구조**: 정점 개체에 대한 확장으로 정점에 해당하는 정수 키(첨자)를 사용하고, n x n 배열로 인접 정점 쌍에 대응하는 간선 노드들에 대한 참조를 저장. 비인접 정점 쌍에는 null 정보를 저장. "구식 버전"은 간선의 존재 여부만 1(간선 존재)과 0(간선 부존재)으로 표시

- **연결 리스트를 이용한 상세 구현**
  - 인접 리스트: 정점 리스트, 간선 리스트를 연결 리스트로 구현
  - 인접 행렬: 2D 포인터 배열을 사용하여 인접 정보를 저장

- **배열을 이용한 상세 구현**
  - 인접 리스트: 정점, 간선 정보를 구조체 배열로 저장하고, 인접 정보는 첨자의 연결 리스트로 표현
  - 인접 행렬: 2D 첨자 배열을 사용하여 인접 정보를 저장

- **그래프 상세 구현 비교**
  - **인접 리스트**: 연결 리스트 사용, 동적 그래프에 유리하지만 다수의 포인터 사용으로 복잡
  - **인접 행렬**: 배열 사용, 다수의 포인터를 첨자로 대체하여 단순하지만 동적 그래프에 불리

- **접근 성능 비교**
  - 표를 이용하여 n개의 정점과 m개의 간선을 가진 그래프에서 간선 리스트, 인접 리스트, 인접 행렬의 공간 복잡도, incidentEdges(v), adjacentVertices(v), areAdjacent(v, w), insertVertex(o), insertEdge(v, w, o), removeVertex(v), removeEdge(e) 연산의 시간 복잡도를 비교

- **응용 문제: 그래프 구현 방식 선택**
  - 다음 각 경우에 인접 리스트 구조와 인접 행렬 구조 중 어느 것을 사용할지 선택하고 이유를 설명
    a. 10,000개의 정점과 20,000개의 간선을 가지며 최소한의 공간을 사용하는 것이 중요
    b. 10,000개의 정점과 20,000,000개의 간선을 가지며 최소한의 공간을 사용하는 것이 중요
    c. 공간 사용량에 관계없이 areAdjacent 질의에 가능한 빨리 답해야 함

- **응용 문제: 배열을 이용한 그래프 데이터 구조**

- 그래프를 배열을 이용하여 구현하기 위한 데이터 구조를 설계. 인접 리스트와 인접 행렬 구조 모두에 공통적인 부분과 차별적인 부분을 명확히 제시. 방향 그래프를 구현하기 위한 설계 변경 사항 설명

- **응용 문제: 정점 또는 간선 삭제 작업의 성능**
  - removeVertex와 removeEdge의 성능을 구현할 수 있는 구체적인 방안 설명. "실제" 삭제와 "비활성화" 방식 삭제 비교

21. <u>그래프 순회</u>

- **순회(traversal):** 모든 정점과 간선을 검사하여 그래프를 탐험하는 체계적인 절차
- **순회 예시:**
  - 수도권 전철망 모든 역(정점) 위치 출력
  - 항공사 모든 항공편(간선) 노선 정보 수집
  - 웹 검색엔진: 웹 하이퍼텍스트 문서(정점)와 링크(간선) 검사
- **주요 전략:** 깊이우선탐색(DFS), 너비우선탐색(BFS)

22. <u>깊이우선탐색(DFS)</u>

- 그래프 순회를 위한 일반적 기법
- **DFS 순회 가능 작업:**
  - 모든 정점과 간선 방문
  - 연결 그래프 여부 결정
  - 연결 요소 계산
  - 신장 숲 계산
- **시간 복잡도:** O(n + m) (n: 정점 수, m: 간선 수)
- **DFS 확장 가능 문제:**
  - 두 정점 사이 경로 찾기
  - 그래프 내 사이클 찾기
- **이진 트리와의 유사성:** 이진 트리의 선위 순회와 유사
- **알고리즘 rDFS(G, v):**
  - 입력: 그래프 G, 시작 정점 v
  - 출력: v의 연결 요소 내 간선 라벨링 (트리 간선, 후향 간선)
  1. l(v) ← Visited
  2. 각 e ∈ G.incidentEdges(v)에 대해
     - if (l(e) = Fresh)
       - w ← G.opposite(v, e)
       - if (l(w) = Fresh)
         - l(e) ← Tree
         - rDFS(G, w)
       - else
         - l(e) ← Back
- **알고리즘 DFS(G):**
  - 입력: 그래프 G
  - 출력: 간선 라벨링 (트리 간선, 후향 간선)
  1. 각 u ∈ G.vertices()에 대해 l(u) ← Fresh
  2. 각 e ∈ G.edges()에 대해 l(e) ← Fresh
  3. 각 v ∈ G.vertices()에 대해
     - if (l(v) = Fresh)
       - rDFS(G, v)
- **DFS 수행 예시:** (그림 설명 생략 - 이미지 참조)
- **DFS와 미로 순회:** 미로 탐험 전략과 유사 (방문한 곳 표시, 경로 추적)
- **DFS 속성:**
  - rDFS(G, v)는 v의 연결 요소 내 모든 정점과 간선 방문
  - rDFS(G, v)에 의해 라벨된 트리 간선은 v의 연결 요소의 신장 트리(DFS 트리) 형성
- **DFS 분석:**
  - 정점/간선 라벨링: O(1)
  - 각 정점 두 번 라벨링 (Fresh, Visited)
  - 각 간선 두 번 라벨링 (Fresh, Tree 또는 Back)
  - incidentEdges 메소드 각 정점에 대해 한 번 호출
  - 인접 리스트 구조: O(n + m) 시간
- **DFS 템플릿 활용:** DFS 알고리즘 확장을 통해 연결성 검사, 경로 찾기, 사이클 찾기 등 수행

23. <u>너비우선탐색(BFS)</u>

- 그래프 순회를 위한 일반적 기법
- **BFS 순회 가능 작업:**
  - 모든 정점과 간선 방문
  - 연결 그래프 여부 결정
  - 연결 요소 계산
  - 신장 숲 계산
- **시간 복잡도:** O(n + m) (n: 정점 수, m: 간선 수)
- **BFS 확장 가능 문제:**
  - 두 정점 사이 최소 간선 경로 찾기
  - 그래프 내 단순 사이클 찾기
- **이진 트리와의 유사성:** 이진 트리의 레벨 순회와 유사
- **알고리즘 BFS1(G, v):**
  - 입력: 그래프 G, 시작 정점 v
  - 출력: v의 연결 요소 내 간선 라벨링 (트리 간선, 교차 간선)
  1. L0 ← 빈 리스트 (레벨 컨테이너)
  2. L0.addLast(v)
  3. l(v) ← Visited
  4. i ← 0

5. while (!Li.isEmpty())
    - Li+1 ← 빈 리스트
    - 각 v ∈ Li.elements()에 대해
        - 각 e ∈ G.incidentEdges(v)에 대해
            - if (l(e) = Fresh)
                - w ← G.opposite(v, e)
                - if (l(w) = Fresh)
                    - l(e) ← Tree
                    - l(w) ← Visited
                    - Li+1.addLast(w)
            - else
                - l(e) ← Cross
    - i ← i + 1

- **알고리즘 BFS(G):**
    - 입력: 그래프 G
    - 출력: 간선 라벨링 (트리 간선, 교차 간선)

1. 각 u ∈ G.vertices()에 대해 l(u) ← Fresh
2. 각 e ∈ G.edges()에 대해 l(e) ← Fresh
3. 각 v ∈ G.vertices()에 대해
    - if (l(v) = Fresh)
        - BFS1(G, v)

- **BFS 수행 예시:** (그림 설명 생략 - 이미지 참조)
- **BFS와 미로 순회:** 미로 탐험의 보수적인 전략과 유사 (레벨별 진행)
- **BFS 속성:**
    - BFS1(G, v)는 v의 연결 요소 내 모든 정점과 간선 방문
    - BFS1(G, v)에 의해 라벨된 트리 간선은 v의 연결 요소의 신장 트리(BFS 트리) 형성
    - Li 내 각 정점 w에 대해, v에서 w로 향하는 경로는 i개의 간선을 가지며, 모든 경로는 최소 i개의 간선을 가짐
- **BFS 분석:**
    - 정점/간선 라벨링: O(1)
    - 각 정점 두 번 라벨링 (Fresh, Visited)
    - 각 간선 두 번 라벨링 (Fresh, Tree 또는 Cross)
    - 각 정점 리스트 Li에 한 번 삽입
    - incidentEdges 메소드 각 정점에 대해 한 번 호출
    - 인접 리스트 구조: O(n + m) 시간
- **BFS 템플릿 활용:** 템플릿 메소드 패턴 사용, 연결 요소 계산, 신장 숲 계산, 단순 사이클 찾기, 최소 간선 경로 찾기 등 O(n + m) 시간에 해결

24. DFS와 BFS 비교

- **트리 간선, 후향 간선, 교차 간선:** (그림 설명 생략 - 이미지 참조)
- **DFS와 BFS 응용:**
    - 신장 숲, 연결 요소, 경로, 사이클, 최단 경로, 이중 연결 요소 계산 (표로 정리)

25. 경로 찾기 (Path Finding)

- **path(G, v, z):** G의 정점 v에서 z까지의 경로를 찾는 DFS 기반 알고리즘
    - 스택 S를 사용하여 v와 현재 정점 사이의 경로 추적
    - z를 만나면 S의 내용을 경로로 반환
    - **path(G, v, z) 알고리즘:**
        - 빈 스택 S 생성
        - pathDFS(G, v, z, S) 호출
        - S의 원소들을 반환
    - **pathDFS(G, v, z, S) 알고리즘:**
        - v를 방문 처리
        - v를 S에 push
        - v == z 이면 종료
        - v에 인접한 각 간선 e에 대해:
            - e가 방문되지 않았으면:
                - w = e의 반대쪽 정점
                - w가 방문되지 않았으면:
                    - e를 트리 간선으로 표시
                    - e를 S에 push
                    - pathDFS(G, w, z, S) 재귀 호출
                    - e를 S에서 pop
                - else: e를 후방 간선으로 표시
        - v를 S에서 pop

26. 자유 트리의 중심 (Center of a Free Tree)

- **문제:** 자유 트리에서 모든 노드까지의 최대 거리가 최소인 노드(중심) 찾기
    - **이심율(eccentricity):** 특정 노드에서 다른 모든 노드까지의 최장 경로 길이
    - **중심:** 이심율이 최소인 노드
    - **알고리즘:**
        - 트리의 복사본 G' 생성
        - G'의 노드 개수가 2보다 클 때까지 잎 노드들을 제거
        - 남은 노드(들)을 중심으로 반환
    - **removeLeaves(G, v, p) 알고리즘:**
        - v의 자식 노드 개수 c 계산
        - 각 자식 노드 w에 대해 removeLeaves(G, w, v) 재귀 호출
        - c == 1 이면 v를 G에서 제거
    - **중심의 개수:** 최대 2개

27. 방향 그래프 (Directed Graph)

- **방향 그래프:** 모든 간선이 방향을 가지는 그래프

- **응용:** 일방통행 도로, 항공 노선, 작업 스케줄링 등
- **방향 DFS:** 방향을 따라 간선을 순회하는 DFS
  - 트리 간선, 후방 간선, 전방 간선, 교차 간선 등의 간선 유형 발생
- **도달 가능성:** 한 정점에서 다른 정점으로의 경로 존재 여부
- **강연결성:** 모든 정점 쌍이 서로 도달 가능한 경우
- **강연결 요소:** 서로 강연결된 최대 부그래프
- **이행적 폐쇄:** u에서 v로의 경로가 존재하면 u에서 v로의 간선을 추가한 그래프
  - **계산 방법:** 각 정점에서 DFS 수행 또는 Floyd-Warshall 알고리즘 사용
- **Floyd-Warshall 알고리즘:** 동적 프로그래밍 기법을 이용한 이행적 폐쇄 계산 알고리즘. 시간복잡도 O(n^3)

28. 방향 비순환 그래프 (DAG)

- **정의**: 방향 사이클이 존재하지 않는 방향 그래프
- **예시**: C++ 클래스 상속 관계, Java 인터페이스, 교과목 선수 관계, 프로젝트 작업 스케줄링, 사전 용어 상호 의존성, 스프레드시트 수식 상호 의존성

29. 위상 정렬

- **정의**: DAG의 모든 정점을 선행 관계를 만족하도록 순서대로 나열하는 것
- **목적**: 작업 스케줄링 등에서 작업 순서 제약을 만족하는 순서를 찾는 데 사용
- **정리**: 방향 그래프가 DAG일 필요충분조건은 위상 순서를 가진다는 것
- **알고리즘 (진입 차수 이용)**:

1. 진입 차수가 0인 정점들을 큐에 넣는다.
2. 큐에서 정점을 하나씩 꺼내 위상 순서에 추가하고, 해당 정점에서 나가는 간선들을 제거한다. (연결된 정점들의 진입 차수 감소)
3. 진입 차수가 0이 된 정점들을 큐에 넣는다.
4. 큐가 빌 때까지 2-3 과정을 반복한다.
5. 모든 정점이 위상 순서에 추가되지 않았다면 사이클이 존재한다.

- **알고리즘 (DFS 이용)**:

1. 각 정점의 방문 상태를 '방문 전', '방문 중', '방문 후'로 관리한다.
2. DFS를 수행하면서, '방문 후' 상태가 된 정점을 위상 순서의 앞쪽에 추가한다.
3. 사이클이 존재하면 DFS 중에 '방문 중' 상태의 정점을 다시 만나게 된다.

- **시간 복잡도**: O(V+E)
- **공간 복잡도**: O(V)

30. 응용문제: 그래프 키우기

- 동적으로 커가는 방향 그래프 G = (V, E) 지원 데이터 구조 설계
- 초기 V = {1, 2, ..., n}, E = ø
- 사용자 작업: insertDirectedEdge(u, v), reachable(u, v)
- 그래프 완전히 연결될 때까지 확장
- insertDirectedEdge 작업 총 수: n(n-1)
- reachable 작업 p회 수행
- 효율적인 데이터 구조 설계

31. 해결: 문제 해결 개요

- n x n 크기의 이행적 폐쇄 행렬 T 유지
- reachable 작업 O(1) 실행 시간
- insertDirectedEdge 작업 총 O(n^3) 최악 실행 시간
- 총 실행 시간: O(n^3 + p)
- p가 작은 경우 O(min(n^3 + p, n^2p))로 개선된 데이터 구조 제시

32. 해결: 이행적 폐쇄 행렬

- 이행적 폐쇄 행렬 T 유지: G의 u에서 v로 방향 경로 존재 시 T[u, v] = 1, 아니면 T[u, v] = 0
- 인접 행렬과 차이점: 경로 존재 여부 추적
- u-번째 행의 1: u가 도달할 수 있는 정점
- u-번째 열의 1: u에 도달할 수 있는 정점
- T[u, u] = 1 (초기화)

33. 해결: reachable, insertDirectedEdge 설계

- reachable(u, v): T[u, v] 조회 (상수 시간)
- insertDirectedEdge(u, v):
  - 간선 (u, v) 추가 시 모든 정점 x 검사
  - T[x, u] & !T[x, v] 이면, v가 도달하는 모든 정점에 x도 도달 가능하도록 행렬 갱신

34. Alg reachable(u, v), Alg insertDirectedEdge(u, v)

- Alg reachable(u, v)
  input transitive closure T, vertex u, v
  output boolean

1. return T[u, v]

   Alg insertDirectedEdge(u, v)
   input transitive closure T, vertex u, v
   output none

1. for x ← 1 to n
   if (T[x, u] & !T[x, v])
   for y ← 1 to n
   T[x, y] ← T[x, y] || T[v, y]

35. 해결: insertDirectedEdge

- 간선 (u, v) 삽입 후 방향 경로 갱신 과정 그림

36. 해결: 알고리즘 성능

- reachable(u, v): O(1) 시간
- insertDirectedEdge: 중첩 반복문, O(n^2) 최악 실행 시간, 총 O(n^3) 시간
- 총 실행 시간: O(n^3 + p)

37. 해결: 행렬을 인접 리스트로 대체하여 성능 개선 시도

- 인접 리스트 사용:
  - insertDirectedEdge: O(1) 시간
  - reachable: O(n^2) 시간 (DFS 또는 BFS 사용)
- 크기 n의 배열 A[0..n-1] 유지, 각 원소는 진출 간선 연결 리스트
- 총 수행 시간: O(n^2 + n^2p)

38. 첫 번째 vs. 두 번째 데이터 구조 – p의 크기에 따라 유불리

- p >> n 또는 p >> n^2 가정 시 두 데이터 구조 비교
- p를 미리 알면 유리한 데이터 구조 선택

39. 해결: 두 데이터 구조를 혼용

- n회 질의까지 인접 리스트 사용, n번째 질의 시 이행적 폐쇄 행렬 구축 후 사용
- 이행적 폐쇄 행렬 구축: O(n^3) 시간
- p ≤ n 이면 O(n^2p), p ≥ n 이면 O(n^3 + p)
- 최악 실행 시간: O(min(n^3 + p, n^2p))

40. 응용문제: 에어텔

- n개 도시, 일직선 상에 위치 (0부터 n-1까지 번호)
- 도시 0에서 n-1로 이동 (오른쪽, 항공편, 하루 한 개)
- 항공편 도착 도시에서 1박
- 항공 요금: A[i], 숙박 요금: H[i]
- 여행 최소 비용 알고리즘 작성

41. 해결: 개요

- 분할 정복 vs. 동적 프로그래밍
- 정방향/역방향 해결 가능
- 정방향: 출발 도시 0 고정, 도착 도시 1부터 n-1까지 변경
- 역방향: 도착 도시 n-1 고정, 출발 도시 n-2부터 0까지 변경
- H[0]과 H[n-1]에 0 저장

42. 해결: 분할 통치법 (정방향)

- 도착 도시 d에 대해, 도시 k (0 ≤ k ≤ d-1) 경유 시 총 비용 계산, 최소값 찾기
- O(2^n) 시간 소요

43. Alg airtel(n), Alg rAirtel(d)

- Alg airtel(n)
  {
  divde and conquer, forward ver.
  }
  input integer n
  output minimum cost of travel from city 0 to n-1

  1. return rAirTel(n-1)

  Alg rAirtel(d)
  input destination city d
  output minimum cost of travel from city 0 to d

  1. if (d = 0)
     return 0
  2. mincost ← ∞
  3. for k ← 0 to d-1
     {
     stopover
     }
     cost ← rAirtel(k) + H[k] + A[d-k]
     mincost ← min(mincost, cost)
  4. return mincost
     {
     Total O(2^n)
     }

44. 해결: 분할 통치법 (역방향)

- 출발 도시 s에 대해, 도시 k (s+1 ≤ k ≤ n-1) 경유 시 총 비용 계산, 최소값 찾기
- O(2^n) 시간 소요

45. Alg airtel(n), Alg rAirtel(s)

- Alg airtel(n)
  {
  divde and conquer, backward ver.
  }
  input integer n
  output minimum cost of travel from city 0 to n-1

1. return rAirtel(0)

   Alg rAirtel(s)
   input start city s
   output minimum cost of travel from city s to n-1

1. if (s = n-1)
   return 0
2. mincost ← ∞
3. for k ← s+1 to n-1
   {
   stopover
   }
   cost ← A[k-s] + H[k] + rAirtel(k)
   mincost ← min(mincost, cost)
4. return mincost
   {
   Total O(2^n)
   }

46. 해결: 분할 통치법의 성능

- 과도한 중복 호출로 효율 저하
- 동적 프로그래밍 방식으로 중복 계산 방지

47. 해결: 동적 프로그래밍 (정방향)

- m[0] = 0 초기화
- m[d]: 도시 0에서 d로 가는 최소 비용
- O(n) 공간, O(n^2) 시간 소요

48. Alg airtel(n)

- Alg airtel(n)
  {
  dynamic programming, forward ver.
  }
  input integer n
  output minimum cost of travel from city 0 to n-1

1. m[0] ← 0
2. for d ← 1 to n-1
   {
   compute m[d]
   }
   m[d] ← ∞
   for k ← 0 to d-1
   {
   stopover
   }
   cost ← m[k] + H[k] + A[d-k]
   m[d] ← min(m[d], cost)
3. return m[n-1]
   {
   Total O(n^2)
   }

49. 해결: 동적 프로그래밍 (역방향)

- m[n-1] = 0 초기화
- m[s]: 도시 s에서 n-1로 가는 최소 비용
- O(n) 공간, O(n^2) 시간 소요

50. Alg airtel(n)

- Alg airtel(n)
  {
  dynamic programming, backward ver.
  }
  input integer n
  output minimum cost of travel from city 0 to n-1

1. m[n-1] ← 0
2. for s ← n-2 downto 0
   {
   compute m[s]
   }
   m[s] ← ∞
   for k ← s+1 to n-1
   {
   stopover
   }
   cost ← A[k-s] + H[k] + m[k]
   m[s] ← min(m[s], cost)
3. return m[0]
   {
   Total O(n^2)
   }

51. 응용문제: 금화 강도

- n x n 셀의 정방형 격자 A
- 각 셀 [i, j]: A[i, j] 금화
- [0, 0]에서 [n-1, n-1]로 이동 (직진, 여러 셀 이동)
- 이동 중 셀 금화 뺏김
- 최적 경로에서 뺏기는 금화 최소량 찾는 알고리즘 (분할 정복, 동적 프로그래밍)

52. 해결: 개요

- 분할 정복 vs. 동적 프로그래밍
- 정방향/역방향 해결 가능
- 정방향: 출발 셀 [0, 0], 도착 셀 [n-1, n-1]까지 변경

- 역방향: 도착 셀 [n-1, n-1], 출발 셀 [0, 0]까지 변경

53. 해결: 분할 통치법 (정방향)

- m(i, j): [0, 0]에서 [i, j]까지 뺏기는 최소 금화량
- m(i, j) = min(minright, mindown)
- minright: k (j-1 ≥ k ≥ 0)에 대해 최소 m(i, k) + A[i, j]
- mindown: k (i-1 ≥ k ≥ 0)에 대해 최소 m(k, j) + A[i, j]
- 베이스 케이스: m(0, 0) = A[0, 0]
- O(2^n) 시간 소요

54. 해결: 분할 통치법 (정방향)

- Alg minGold(A, n), Alg m(i, j)

55. 해결: 분할 통치법 (역방향)

- m(i, j): [i, j]에서 [n-1, n-1]까지 뺏기는 최소 금화량
- m(i, j) = min(minright, mindown)
- minright: k (j+1 ≤ k ≤ n-1)에 대해 최소 A[i, j] + m(i, k)
- mindown: k (i+1 ≤ k ≤ n-1)에 대해 최소 A[i, j] + m(k, j)
- 베이스 케이스: m(n-1, n-1) = A[n-1, n-1]
- O(2^n) 시간 소요

56. 해결: 분할 통치법 (역방향)

- Alg minGold(A, n), Alg m(i, j)

57. 해결: 동적 프로그래밍 (정방향)

- m[i, j]: [0, 0]에서 [i, j]까지 뺏기는 최소 금화량
- m[0, 0] = A[0, 0] 초기화
- O(n^2) 공간, O(n^3) 시간 소요

58. 해결: 동적 프로그래밍 (정방향)

- Alg minGold(A, n)

59. 해결: 동적 프로그래밍 (역방향)

- m[i, j]: [i, j]에서 [n-1, n-1]까지 뺏기는 최소 금화량
- m[n-1, n-1] = A[n-1, n-1] 초기화
- O(n^2) 공간, O(n^3) 시간 소요

60. 해결: 동적 프로그래밍 (역방향)

- Alg minGold(A, n)

61. 응용문제: 부배열의 최대 구간합

- 크기 n의 실수 배열 A
- 부배열 구간합 ∑A[i:j]가 최대가 되는 구간 i:j (i ≤ j)와 구간합 찾는 알고리즘

62. 해결: 단순 직선적

- 모든 가능한 i:j 구간 검사
- O(n^3) 시간, O(1) 공간

63. Alg maxSubarray(A, n) {v.1}

- Alg maxSubarray(A, n) {v.1}
  input array A of n real numbers
  output maximum subarray A[i:j], index i, j

1. maxSum ← -∞
2. for i ← 0 to n-1
   {
   O(n)
   }
   for j ← i to n-1
   {
   O(n^2)
   }
   sum ← 0
   for k ← i to j
   {
   O(n^3)
   }
   sum ← sum + A[k]
   if (maxSum < sum)
   maxSum, maxi, maxj ← sum, i, j
3. return maxSum, i, j
   {
   Total O(n^3)
   }

64. 해결: 누적합을 사용

- ∑A[i:j] = ∑A[i:j-1] + A[j] 이용
- 누적합으로 구간합 계산
- O(n^2) 시간, O(1) 공간

65. Alg maxSubarray(A, n) {v.2}

- Alg maxSubarray(A, n) {v.2}
  input array A of n real numbers
  output maximum subarray A[i:j], index i, j

1. maxSum ← -∞
2. for i ← 0 to n-1
   {
   O(n)
   }
   sum ← 0
   for j ← i to n-1
   {
   O(n^2)
   }
   sum ← sum + A[j]
   if (maxSum < sum)
   maxSum, maxi, maxj ← sum, i, j
3. return maxSum, i, j
   {
   Total O(n^2)
   }

66. 해결: 초기 구간합을 사용

- 초기 구간합 s[i] = ∑A[0:i] 사용
- ∑A[i:j] = s[j] - s[i-1]
- O(n^2) 시간, O(n) 공간

67. Alg maxSubarray(A, n) {v.3}

- Alg maxSubarray(A, n) {v.3}
  input array A of n real numbers
  output maximum subarray A[i:j], index i, j

1. s[-1] ← 0
2. for i ← 0 to n-1
   {
   O(n)
   }
   s[i] ← s[i-1] + A[i]
3. maxSum ← -∞
4. for i ← 0 to n-1
   {
   O(n)
   }
   for j ← i to n-1
   {
   O(n^2)
   }
   sum ← s[j] - s[i-1]
   if (maxSum < sum)
   maxSum, maxi, maxj ← sum, i, j
5. return maxSum, maxi, maxj
   {
   Total O(n^2)
   }

68. 해결: 동적 프로그래밍을 사용

- s[i] = max(s[i-1] + A[i], A[i])
- s[i-1] < 0 이면 k = i
- O(n) 시간, O(n) 공간 (O(1)로 개선 가능)

69. Alg maxSubarray(A, n) {v.4}

- Alg maxSubarray(A, n) {v.4}
  input array A of n real numbers
  output maximum subarray A[i:j], index i, j

1. s[-1] ← 0
2. maxSum, maxi, k ← -∞, 0, 0
3. i ← 0
4. while (i < n)
   {
   O(n)
   }
   s[i] ← max(s[i-1] + A[i], A[i])
   if (s[i-1] < 0)
   k ← i
   if (maxSum < s[i])
   maxSum ← s[i]
   maxi, maxj ← k, i
   i ← i + 1
5. return maxSum, maxi, maxj
   {
   Total O(n)
   }

70. 최소 신장 트리

- **가중 그래프**: 각 간선이 무게(weight)라는 수치값을 가지는 그래프
  - 무게: 거리, 비용, 시간 등
- **신장 부그래프**: 그래프 G의 모든 정점을 포함하는 부그래프
- **신장 트리**: (자유) 트리인 신장 부그래프
- **최소 신장 트리(MST)**: 가중 그래프의 총 간선 무게가 최소인 신장 트리
  - 응용: 통신망, 교통망

- **최소 신장 트리 속성**

- **싸이클 속성**: T를 가중 그래프 G의 최소 신장 트리라 하자. e를 T에 존재하지 않는 G의 간선으로, C를 e를 T에 추가하여 형성된 싸이클이라 가정. 그러면 C의 모든 간선 f에 대해, weight(f) ≤ weight(e)
    - 증명: 모순법. 만약 weight(f) > weight(e)라면, f를 e로 대체함으로써 무게가 더 작은 신장 트리를 얻을 수 있기 때문
- **분할 속성**: G의 정점들을 두 개의 부분집합 U와 V로 분할한다고 하자. e를 분할을 가로지르는 최소 무게의 간선이라고 하자. 간선 e를 포함하는 G의 최소 신장 트리가 반드시 존재
    - 증명: T를 G의 MST라 하자. 만약 T가 e를 포함하지 않는다면, e를 T에 추가하여 형성된 싸이클 C를 구성하는 간선들 가운데 분할을 가로지르는 간선 f가 존재. 싸이클 속성에 의해, weight(f) ≤ weight(e). 그러므로, weight(f) = weight(e). f를 e로 대체하면 또 하나의 MST를 얻을 수 있다

- **탐욕법**
  - 탐욕법(greedy method): 일반적인 알고리즘 설계 기법 중 하나. 다음 요소에 기초하여 설계
    - 구성(configuration): 다양한 선택, 모음, 또는 찾아야 할 값들
    - 목표(objective): 구성에 할당된 점수가 존재하며, 이를 최대화 또는 최소화해야 하는 상황
    - 탐욕적 선택 속성(greedy-choice property)을 가진 문제에 적용할 경우 가장 잘 맞는다. 출발 구성으로부터 시작하여 지속적인 지역적 향상을 통해 전체 최적해를 항상 찾을 수 있다
  - 예: 잔돈 거스르기, 부분적 배낭 문제, 최소 신장 트리 문제

- **알고리즘 예시**
  - **잔돈 거스르기 문제**: 동전 종류에 따라 탐욕적 선택 속성 유무가 달라짐 (예: 32원, 8원, 1원 vs 30원, 20원, 5원, 1원)
  - **부분적 배낭 문제**: 각 항목의 일부만을 취할 수 있는 문제. $\sum_{i \in S} b_i x_i$ 를 최대화, $\sum_{i \in S} v_i x_i \le V$
  - **0-1 배낭 문제**: 각 항목의 일부만을 취할 수 없는 문제. 탐욕적 선택 속성을 만족하지 않음

71. 최소 신장 트리 알고리즘

- **Prim-Jarnik 알고리즘**: 탐욕 알고리즘, 단순 연결 무방향 가중 그래프에 적용
    - 임의의 정점 s에서 시작, MST T를 키워나감
    - 각 정점 v에 라벨 d(v) 정의 (배낭 안 정점과 밖의 정점 연결 간선 무게)
    - 반복: 배낭 밖 정점 중 최소 d(z) 라벨 가진 정점 z를 배낭에 넣고, z에 인접한 정점 라벨 갱신
    - 우선순위 큐 사용 (키: 거리, 원소: 정점)
    - 보조 메소드 Q.replaceKey(e, k): 원소 e의 키를 k로 변경하고 우선순위 큐 내 위치 갱신
    - 각 정점 v에 거리, 위치자, 부모 라벨 저장
    - 탐욕법의 일반 공식과 일치 (구성: 다양한 수치 항목, 목표: 총수치 최소화, 해결: 최소 수치 항목부터 포함)
    - 탐욕적 선택 속성 만족 (탐욕적 문제 구성과 목표 설정 가능하고, 탐욕적 해결로 목표 달성 가능)
    - 정확성: 각 회전에서 최소 무게 간선 선택, MST에 타당한 간선 추가, 분할 속성 만족
    - 분석: O((n+m)logn) 시간, 인접 리스트 구조면 O(m log n)
- **Kruskal 알고리즘**: 탐욕 알고리즘
    - 초기 작업: 모든 정점을 각각의 배낭에 넣고, 배낭 밖 간선을 우선순위 큐에 저장 (키: 무게, 원소: 간선), 비어있는 MST T 초기화
    - 반복: 두 개의 다른 배낭에 양끝점을 가진 최소 무게 간선을 MST T에 포함, 두 배낭 합침
    - 반복 완료: MST T를 포함하는 한 개의 배낭만 남음
    - 정확성: 분할 속성으로 유도, 각 회전마다 타당한 MST 간선 추가
    - 데이터 구조: 인접 정보 사용 X, 간선 리스트 구조, 트리들의 숲을 분리 집합으로 저장 (find: O(1), union: O(min(nu, nv)))
    - 분석: O((n+m)logn) 시간, 단순 연결 그래프면 O(m log n)
- **Baruvka 알고리즘**: 탐욕 알고리즘, 우선순위 큐 사용 X
    - 초기 작업: 모든 정점을 각각의 배낭에 넣음
    - 반복: 각 연결 요소 Ci에서 다른 요소로 가는 최소 무게 간선 선택, T에 추가 (이미 T에 있으면 제외)
    - 정확성: 각 단계에서 MST에 반드시 포함되어야 하는 간선 선택, 분할 속성 만족
    - 구현: 인접 리스트 사용, 연결 요소 찾기 위해 DFS 사용, 각 정점에 라벨 정의
    - 분석: O(m log n) 시간

72. MST 알고리즘 비교

- 알고리즘 | 주요 전략 | 수행 시간 | 외부 데이터 구조
    - --- | --- | --- | ---
    - Prim-Jarnik | 탐욕 | O(m log n) | 정점 저장 위한 우선순위 큐
    - Kruskal | 탐욕 | O(m log n) | 간선 저장 위한 우선순위 큐, 배낭 구현 위한 분리 집합 (리스트로 구현 가능)
    - Baruvka | 탐욕 | O(m log n) | 연결 요소 표현 위한 데이터 구조 필요

73. 보석 전시회 경비 배치 문제

- **문제:** 긴 복도(1차원 축 L)를 따라 놓인 보석들의 위치 X = {x0, x1, ..., xn-1}가 주어짐.
- **조건:** 한 경비는 자신의 위치에서 좌우 최대 k 거리까지 커버 가능 (좌우 k/2).

- **목표:** 최소 인원의 경비로 모든 보석을 지키는 경비 배치 계산.
- **알고리즘:** gemGuard(X, k)
- **알고리즘 gemGuard(X, k):**

1. X의 원소들을 오름차순으로 정렬
2. G (경비 위치 리스트)를 빈 리스트로 초기화
3. x = X에서 첫 번째 원소 제거
4. g = x + k/2 (첫 번째 경비 위치)
5. G에 g 추가
6. X가 빌 때까지 반복:

- **알고리즘 gemGuard(X, k):**

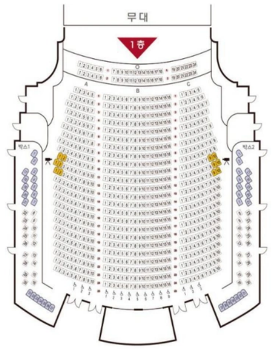
1. X의 원소들을 오름차순으로 정렬
2. G (경비 위치 리스트)를 빈 리스트로 초기화
3. x = X에서 첫 번째 원소 제거
4. g = x + k/2 (첫 번째 경비 위치)
5. G에 g 추가
6. X가 빌 때까지 반복:

- x = X에서 첫 번째 원소 제거
- 만약 (x - g > k/2):
- g = x + k/2 (다음 경비 위치)
- G에 g 추가

7. G 반환

74. 공연홀 좌석 배정 문제

- **문제:** 티켓 판매고를 극대화하기 위한 좌석 배정 전략.
- **조건:** 단체 관람객은 모든 멤버가 좌석을 배정받아야 함.
- **기존 전략:** 선착순 배정
- **새로운 전략:** 큰 단체 우선 배정 후 작은 단체 순으로 배정, 마지막으로 개인 배정.
- **알고리즘:** seat(G, n)



- **알고리즘 seat(G, n):**

1. admitted = 0
2. G를 크기 순으로 내림차순 정렬
3. remaining = n (남은 좌석 수)
4. i = 1부터 m까지 반복:

- **알고리즘 seat(G, n):**

1. admitted = 0
2. G를 크기 순으로 내림차순 정렬
3. remaining = n (남은 좌석 수)
4. i = 1부터 m까지 반복:

- 만약 (G[i] ≤ remaining):
- admit(i) (단체 i 입장)
- remaining = remaining - G[i]
- admitted = admitted + G[i]
- 그렇지 않으면:
- reject(i) (단체 i 거절)

5. admitted 반환

- **탐욕 알고리즘 seat의 비최적성 예시:** G = ((n+2)/2, n/2, n/2)일 때, seat 알고리즘은 (n+2)/2 크기의 단체만 입장시키지만, 최적 알고리즘은 n/2 크기의 두 단체를 입장시켜 n개의 좌석을 모두 채움.
- **seat 알고리즘의 하한 보장:** 만약 k명을 입장시키는 것이 최적해라면, seat 알고리즘은 적어도 k/2명을 입장시킴. 증명은 여러 경우를 나누어 고려하여 seat 알고리즘이 모든 단체를 입장시키거나, 입장시키지 못하는 경우 모두 k/2명 이상을 입장시킨다는 것을 보임.

75. 8자 모양 그래프의 최소 신장 트리

- **문제:** 두 개의 사이클이 한 정점에서 만나는 8자 모양 그래프 G의 최소 신장 트리(MST)를 구하는 알고리즘.
- **알고리즘:** eight-ShapedMST(G)
- **방법:** 각 사이클에서 최대 가중치 간선을 제거하여 MST 생성.
- **알고리즘 eight-ShapedMST(G):**

1. e1 = 왼쪽 사이클의 최대 가중치 간선 (O(n))
2. e2 = 오른쪽 사이클의 최대 가중치 간선 (O(n))
3. T = E - ({e1} ∪ {e2})
4. T 반환 (총 O(n))

76. 최단 경로 문제

- **문제:** 가중 그래프와 두 정점 u, v가 주어졌을 때, u와 v 사이의 최소 가중치 경로를 구하는 문제.
- **최단 경로 길이:** 간선 가중치의 합.
- **응용:** 인터넷 패킷 라우팅, 항공편 예약, 내비게이션 등.
- **최단 경로 속성:**

1. 최단 경로의 부분 경로도 최단 경로이다.
2. 출발 정점으로부터 다른 모든 정점까지의 최단 경로 트리가 존재한다 (단일점 최단 경로).

- **최소 신장 트리와의 비교:** 최단 경로는 방향 그래프에서도 정의되며, 음의 가중치 사이클이 존재하면 최단 경로가 존재하지 않을 수 있다.
- **음의 가중치 간선과 사이클:** 가중 방향 그래프에 음의 가중치 사이클이 있거나, 가중 무방향 그래프에 음의 가중치 간선이 있으면 최단 경로가 존재하지 않을 수 있다.

77. 최단경로 알고리즘

- **다익스트라(Dijkstra) 알고리즘**: 음의 무게 간선이 없는 그래프에서 사용, 시간복잡도 O(m log n) or O(n^2)
- **벨만-포드(Bellman-Ford) 알고리즘**: 음의 무게 간선이 있는 방향 그래프에서 사용, 시간복잡도 O(nm)

- **BFS(Breadth-First Search)**: 비가중 그래프에서 사용, 시간복잡도 O(n+m)
- **위상 정렬(Topological Ordering)**: DAG(Directed Acyclic Graph)에서 사용, 시간복잡도 O(n+m)
- **다익스트라 알고리즘 전제조건**: 그래프 연결, 무방향 간선, 음수 아닌 간선 무게
- **다익스트라 알고리즘 기본 구성**: 배낭(우선순위 큐 사용), 각 정점의 거리 라벨(d(v)), 위치자 라벨
  - **우선순위 큐**: 키는 거리, 원소는 정점
  - **보조 메소드 Q.replaceKey(e, k)**: 원소 e의 키를 k로 변경하고 우선순위 큐에서 위치 갱신
- **다익스트라 알고리즘 단계**:

1. 각 정점 v에 대해 d(v) = ∞, d(s) = 0
2. 우선순위 큐 Q에 모든 정점을 d 라벨을 키로 하여 삽입
3. while(!Q.isEmpty()){
   u = Q.removeMin()
   u에 인접한 각 정점 z에 대해
   if(z ∈ Q.elements() && d(u) + w(u, z) < d(z)){
   d(z) = d(u) + w(u, z)
   Q.replaceKey(z, d(z))
   }
   }

- **간선 완화(relaxation)**: d(z) = min(d(z), d(u) + w(u, z))
- **다익스트라 알고리즘 정확성**: 탐욕 알고리즘 기반, 거리가 늘어나는 순서로 정점을 배낭에 삽입. 모순법을 통해 정확성 증명 가능
- **다익스트라 알고리즘 음의 무게 간선 문제**: 음의 무게 간선이 있으면 이미 배낭에 있는 정점의 거리를 혼란시킴
- **음의 무게 간선 해결 시도(잘못된 방법)**: 모든 간선 무게에 상수 k를 더하고, 최단 경로를 구한 후, 결과를 보정하는 방법은 잘못됨
- **다익스트라 알고리즘 분석**:
  - 힙 기반 우선순위 큐 사용 시: O((n+m)log n) → O(m log n) (연결 그래프)
  - 무순서 리스트 기반 우선순위 큐 사용 시: O(n^2 + m) → O(n^2) (단순 그래프)
- **힙 vs 무순서 리스트**: 희소 그래프(m < n^2/log n)에서는 힙, 밀집 그래프(m > n^2/log n)에서는 리스트가 유리
- **벨만-포드 알고리즘**: 음의 무게 간선 존재 가능, 방향 그래프 전제, 시간복잡도 O(nm), 음의 무게 사이클 검출 가능
- **DAG에서의 최단경로 알고리즘**: 위상 정렬 이용, 음의 무게 간선 존재 가능, 별도 데이터 구조 필요 없음, 시간복잡도 O(n+m)

78. <u>모든 쌍 최단 경로</u>

- **문제**: 가중 방향 그래프 G의 모든 정점쌍 간의 거리를 찾는 문제
- **음의 무게 간선 없을 때**: Dijkstra 알고리즘 n번 호출, O(nm log n) 시간 복잡도
- **음의 무게 간선 있을 때**: Bellman-Ford 알고리즘 n번 호출, O(n^2m) 시간 복잡도
- **대안**: 동적 프로그래밍 사용, O(n^3) 시간 복잡도 (Floyd-Warshall 알고리즘 유사)
- **알고리즘 단계**:

1. 정점 번호 매기기: v1, v2, ..., vn
2. 초기화:

- **알고리즘 단계**:

1. 정점 번호 매기기: v1, v2, ..., vn
2. 초기화:
   - D[i, j] = 0 (i = j)
   - D[i, j] = w(vi, vj) ((vi, vj) ∈ G.edges())
   - D[i, j] = ∞ (그 외)
3. 반복 계산:
   - for k = 1 to n
   - for i = 1 to n
   - for j = 1 to n
   - D[i, j] = min(D[i, j], D[i, k] + D[k, j])

79. <u>Floyd-Warshall 알고리즘</u>

- **입력**: 음의 가중치 cycle이 없는 단순 가중 방향 그래프 G
- **출력**: 정점 번호 v1, v2, ..., vn과 거리 행렬 D (D[i, j]: vi에서 vj까지의 거리)
- **설명**: k번째 반복에서, 정점 1부터 k까지를 중간 정점으로 사용하여 vi에서 vj까지의 최단 경로를 계산. k가 증가함에 따라 더 짧은 경로를 찾을 수 있음.

80. <u>두 지국 사이의 최대 대역폭</u>

- **정점**: 지국
- **간선**: 지국 간의 통신선로 (각 간선은 대역폭과 함께 표시됨)
- **경로의 대역폭**: 경로 내 최소의 대역폭을 가지는 간선(즉, 병목)의 대역폭
- **문제**: 다이어그램과 두 개의 지국 x, y가 주어졌을 때, x, y 사이의 경로 가운데 최대 대역폭을 구하는 알고리즘을 작성하라
- **힌트**: Dijkstra 알고리즘의 확장

81. <u>Dijkstra 알고리즘 확장</u>

- Dijkstra 알고리즘에서와 동일한 아이디어 사용
- 각 정점 v에 새로운 라벨 b를 유지: bandwidth: b(v), x로부터 v까지 경로의 대역폭

- 모든 정점에 대해 b 라벨 값을 0으로 초기화 (단, 출발점 x의 b 값은 무한대로 초기화)
- 작업 removeMin은 우선순위 큐로부터 최대 b 값을 가지는 노드를 삭제
- 실행시간: (힙에 기초한 우선순위 큐를 사용할 경우) O((n + m)log n)
- Alg DijkstraMaxBandwidth(G, x, y)

1. for each v ∈ G.vertices()
   b(v) ← 0
2. b(x) ← ∞
3. Q ← a priority queue containing all the vertices of G using the b labels as keys
4. while (!Q.isEmpty())
   u ← Q.removeMin()
   for each e ∈ G.incidentEdges(u)
   z ← G.opposite(u, e)
   if (z ∈ Q.elements())
   if (min(b(u), w(u, z)) > b(z))
   b(z) ← min(b(u), w(u, z))
   Q.replaceKey(z, b(z))
5. return b(y)

82. <u>간선 완화</u>

- 간선 완화 단계는 Dijkstra에서와 매우 유사
- 간선 e = (u, z)을 고려: u는 배낭에 최근에 추가된 정점, z는 배낭에 존재하지 않는다
- 간선 e의 완화는 b(z)를 다음과 같이 갱신: b(z) ← max(b(z), min(b(u), weight(e)))

83. <u>항공편 스케줄링</u>

- **문제**: 주어진 두 개의 공항 a, b와 시각 t에 대해 a에서 시각 t 정시 혹은 이후에 출발할 경우 가장 이른 시각에 b에 도착할 수 있도록 하는 연결 항공편을 계산하고 알고리즘의 실행시간을 n과 m의 함수로 구하라 (환승공항에서의 최소 연결시간 준수)
- **해결**: 최단 경로 문제로 전환
  - 가중 방향 그래프 G 구축: 각 공항 a ∈ A의 24시간을 표현하는 circle을 그림, 각 항공편의 출발/도착 공항, 시각 정보를 이용하여 정점과 간선 생성 (자정을 지나가는 항공편도 고려)
  - 최단 경로 문제 해결: Dijkstra 알고리즘 확장 적용 (방향 그래프, 최단 경로 회수)
- **알고리즘 성능**: 그래프 구축: O(n + m), 최단 경로 찾기: O((n + m)log n) (힙 사용), 일반적으로 n = O(m)이므로 O(m log n)
- Alg flightScheduling(G, a, b, t)

1. v ← first vertex on circlea representing time t or after t
2. for each u ∈ G.vertices()
   d(u) ← ∞
   p(v) ← ∅
3. d(v) ← 0
4. Q ← a priority queue containing all the vertices of G using d labels as keys
5. while (!Q.isEmpty())
   u ← Q.removeMin()
   for each e ∈ G.outIncidentEdges(u)
   z ← G.opposite(u, e)
   if (z ∈ Q.elements())
   if (d(u) + w(u, z) < d(z))
   d(z) ← d(u) + w(u, z)
   p(z) ← e
   Q.replaceKey(z, d(z))
6. w ← vertex on circleb with minimum d label
7. return reversed path from w to v

84. <u>좌회전을 못하는 차</u>

- **문제**: 출발지 s에서 목적지 t까지 우회전을 최소화한 경로 찾기 (우회전 수가 같다면 총 주행거리가 짧은 경로)
- **해결**: 최단 경로 문제로 재구성
  - 가중 방향 그래프로 전환: 각 셀에 E, W, S, N 정점 생성, 직진(무게 1), 우회전(무게 4mn + 1), 좌회전, U턴 없음
  - Dijkstra 알고리즘 적용
  - 성능: 그래프 구축 O(mn), 최단 경로 찾기 O(mn log mn)

85. <u>괴물성에 갇힌 낙랑</u>

- **문제**: 초기 에너지 L을 가지고 미로의 입구 s에서 출발하여 낙랑이 있는 방 t를 찾기 (에너지 레벨 L을 0 이상으로 유지)
- **해결 (개요)**: 그래프에 대한 통찰력과 Bellman-Ford 최단 경로 알고리즘을 이용
- **해결 A (간선에 무게가 실린 그래프로 변환)**: 정점의 무게를 간선으로 이동 (정점 v가 무게 f(v)를 가진다면, 모든 간선 (u, v) ∈ E에 대해 w(u, v) = f(v) 저장)
- **해결 B (에너지 증가 싸이클이 없는 경우)**: Bellman-Ford 알고리즘 수정 버전 사용, 각 정점 u에 대해 최대 에너지 e(u) 계산, e(u)가 양수이면 레벨-r
  - 간선 완화 수정: if ((d(v) > d(u) + w(u, v)) & (d(u) + w(u, v) > 0)) d(v) ← d(u) + w(u, v)
- **해결 C (에너지 증가 싸이클이 있는 경우)**: 모든 간선을 n - 1번 완화 후에도 d(t)가 양수가 아니면 한 번 더 완화, d(u)가 변화하면 양의 무게를 가진 싸이클 발견 (레벨-r), 도달 가능한 양의 무게를 가진 싸이클을 찾지 못하고 d(t) = -∞이면 레벨-r 아님
- **해결 D (최소 r 찾기)**: 그래프가 레벨-1인지 검사, 아니면 레벨-2, 레벨-4... 검사, 이진 탐색으로 최소 r 찾기, 실행시간 O(mn log r)