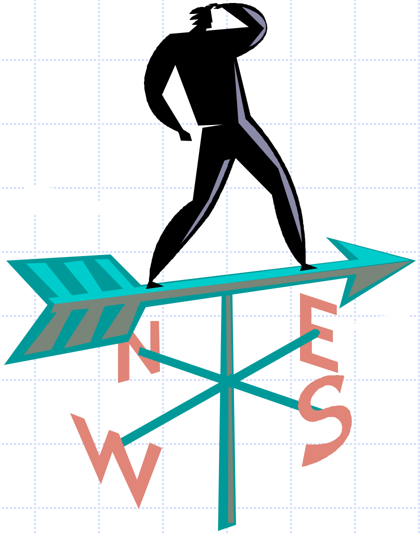
방향그래프

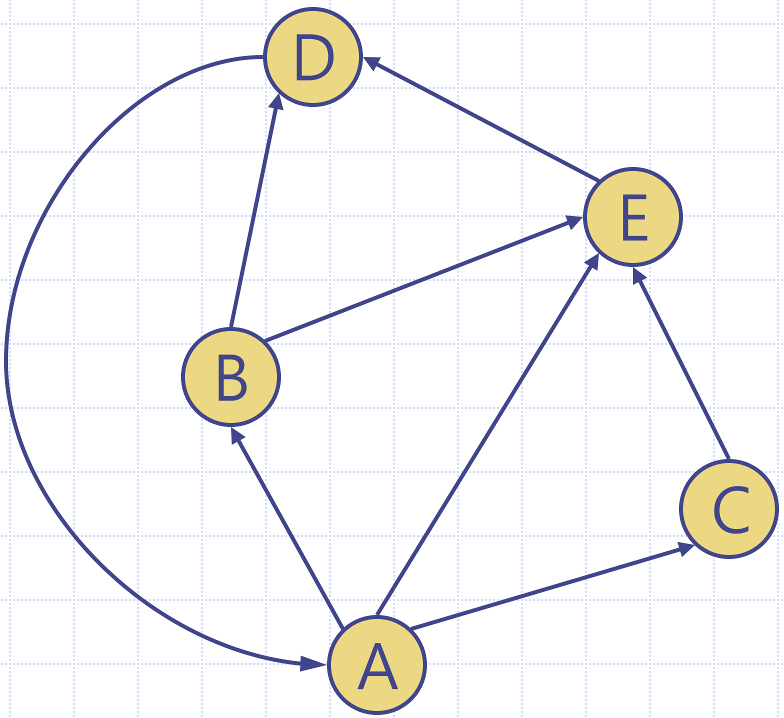


Outline

- 15.1 방향그래프
- 15.2 동적프로그래밍
- 15.3 방향 비싸이클 그래프
- 15.4 응용문제

방향그래프

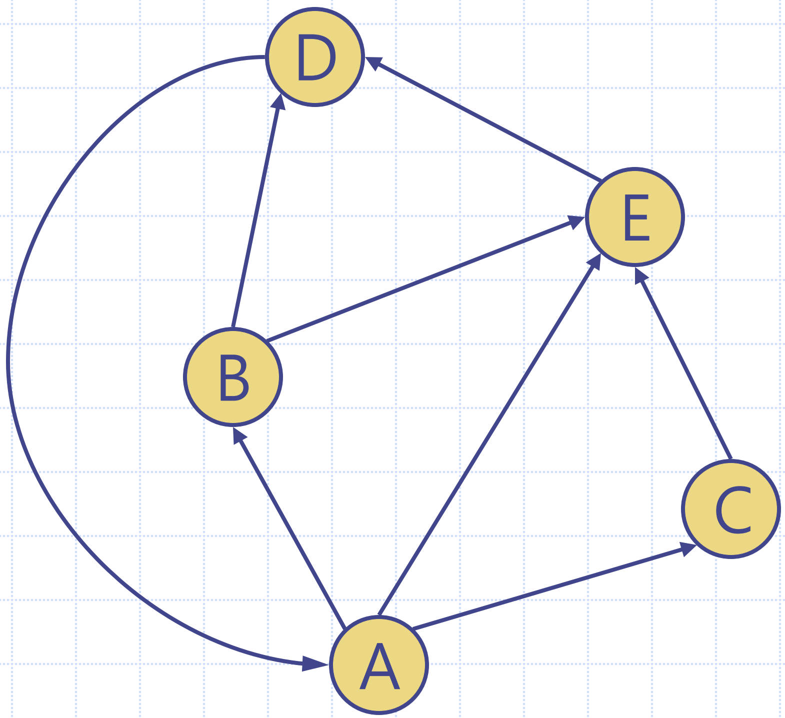
- **방향그래프**(digraph): 모든 간선이 **방향간선**인 그래프
  - "**directed graph**"의 준말
- **응용**
  - 일방통행 도로
  - 항공노선
  - 작업스케줄링



방향그래프 속성

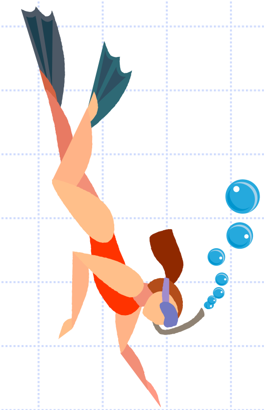
- 모든 간선이 한 방향으로 진행하는 그래프 $G = (V, E)$에서,
  - 간선 (a, b)는 a에서 b로 가지만 b에서 a로 가지는 않는다
- G가 단순하다면,

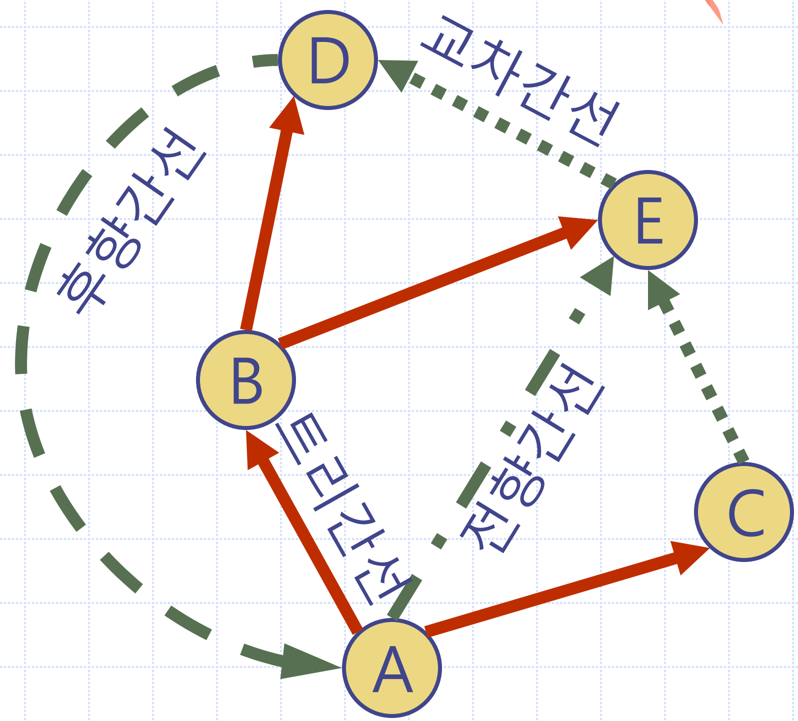
  $$m \leq n(n-1)$$
- **진입간선**들(in-edges)과 **진출간선**들(out-edges)을 각각 별도의 인접리스트로 보관한다면, 진입간선들의 집합과 진출간선들의 집합을 각각의 크기에 비례한 시간에 조사 가능



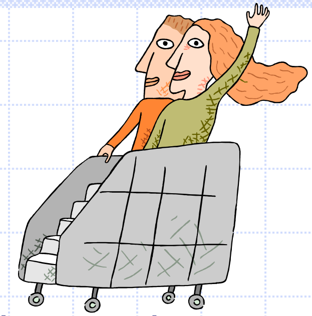
방향 DFS

- 간선들을 주어진 방향만을 따라 순회하도록 하면, DFS 및 BFS 순회 알고리즘들을 방향그래프에 특화 가능
- **방향** DFS 알고리즘에서, 네 종류의 간선이 발생
  - **트리간선**(tree edges)
  - **후향간선**(back edges)
  - **전향간선**(forward edges)
  - **교차간선**(cross edges)
- 정점 s에서 출발하는 방향 DFS는 s로부터 도달 가능한 정점들을 결정

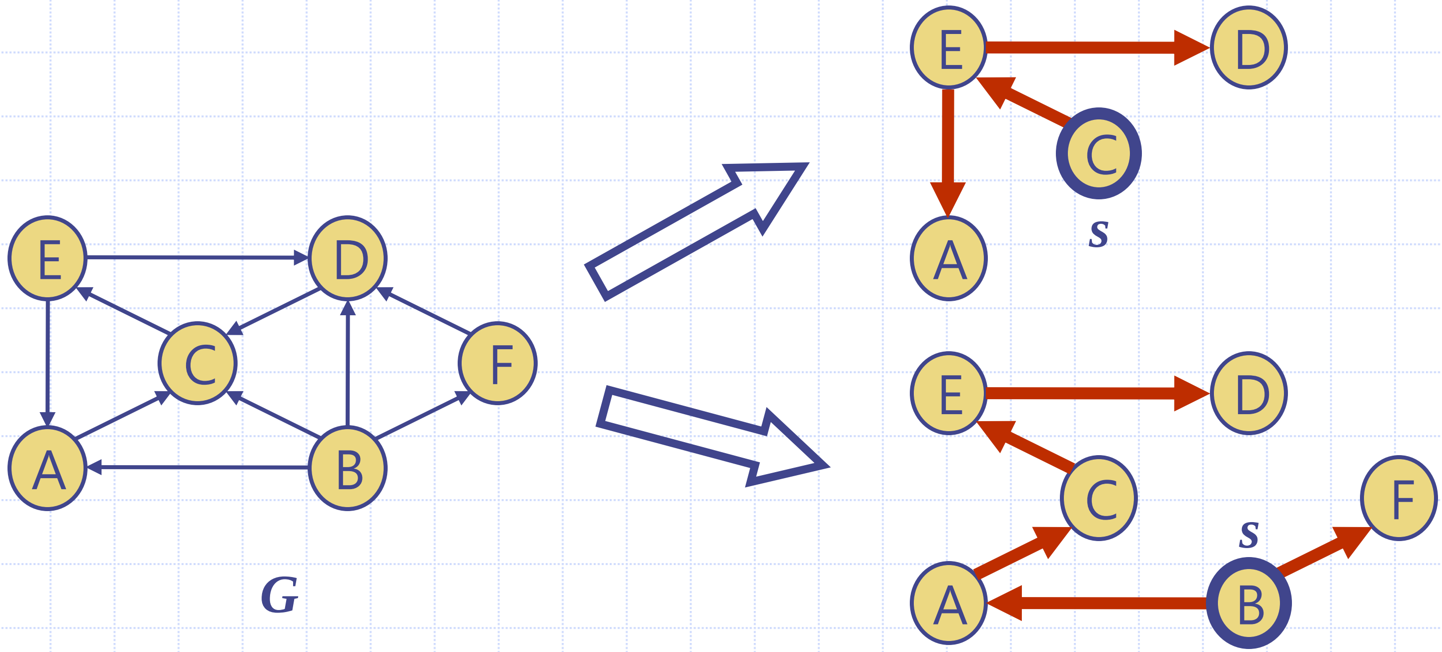




도달 가능성

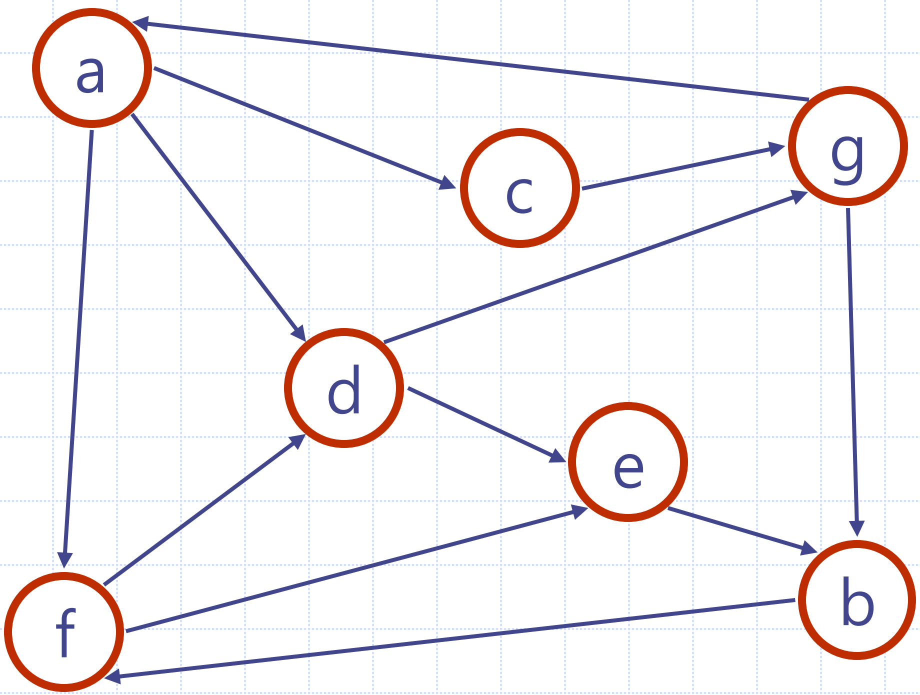


- 방향그래프 G의 두 정점 u와 v에 대해 만약 G에 u에서 v로의 방향경로가 존재한다면, "u는 v에 **도달한다**(u reaches v)", 또는 "v는 u로부터 **도달 가능**하다(v is reachable from u)"고 말한다
- s를 루트로 하는 **DFS 트리**: s로부터 방향경로를 통해 도달 가능한 정점들을 표시



강연결성

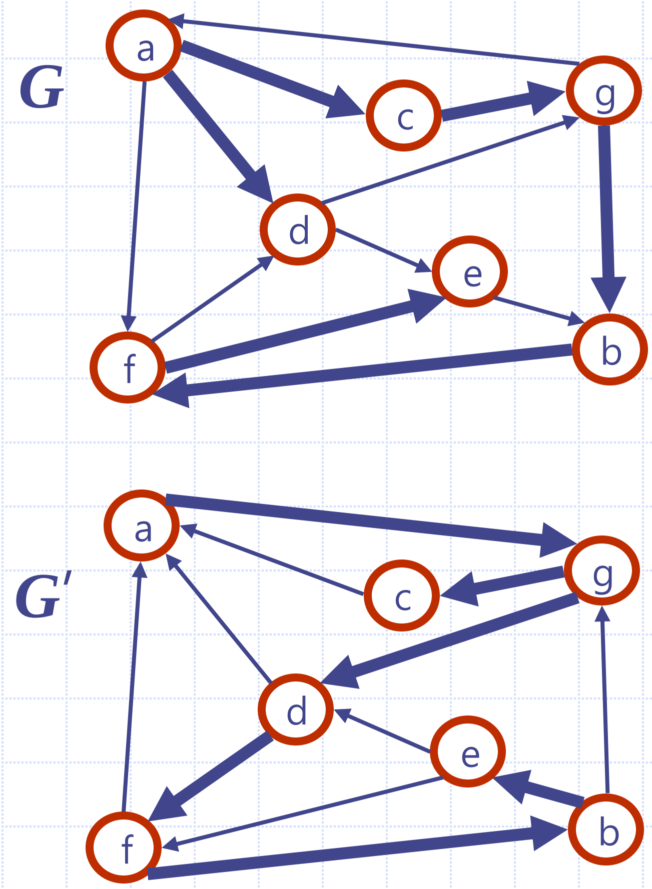
- 방향그래프 G의 어느 두 정점 u와 v에 대해서나 u는 v에 도달하며 v는 u에 도달하면, G를 **강연결**(strongly connected)이라고 말한다 – 즉, 어느 정점에서든지 다른 모든 정점에 도달 가능



강연결 검사 알고리즘

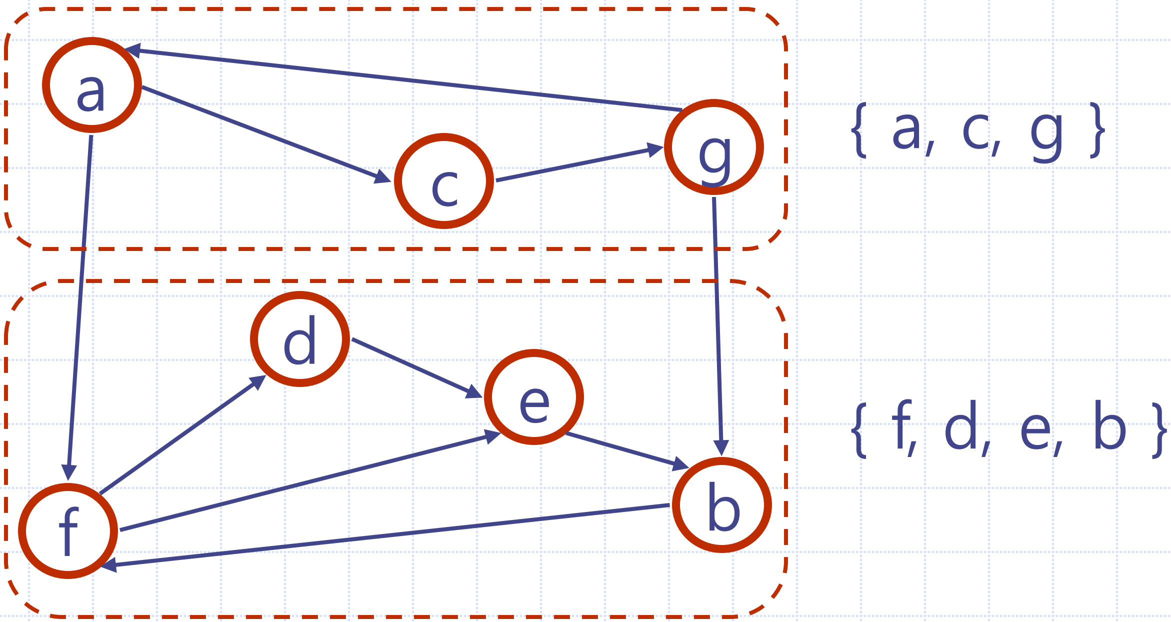
1. *G*의 임의의 정점 *v*를 선택
2. *G*의 *v*로부터 DFS를 수행
   1. 방문되지 않은 정점 *w*가 있다면, ***False***를 반환
3. *G*의 간선들을 모두 역행시킨 그래프 *G*′를 얻음
4. *G*′의 *v*로부터 DFS를 수행
   1. 방문되지 않은 정점 *w*가 있다면, ***False***를 반환
   2. 그렇지 않으면, ***True***를 반환

- **실행시간: O(*n* + *m*)**



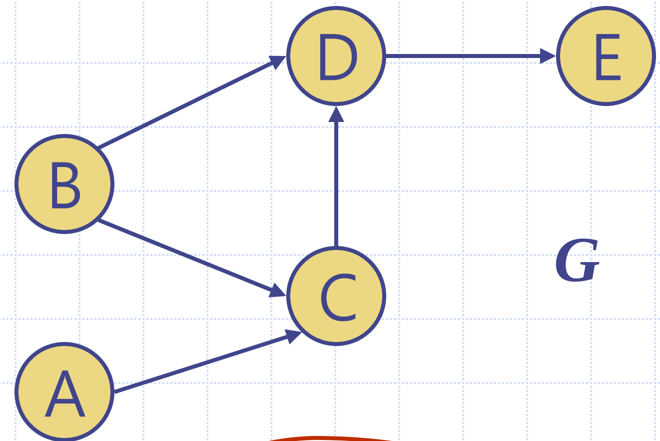
강연결요소

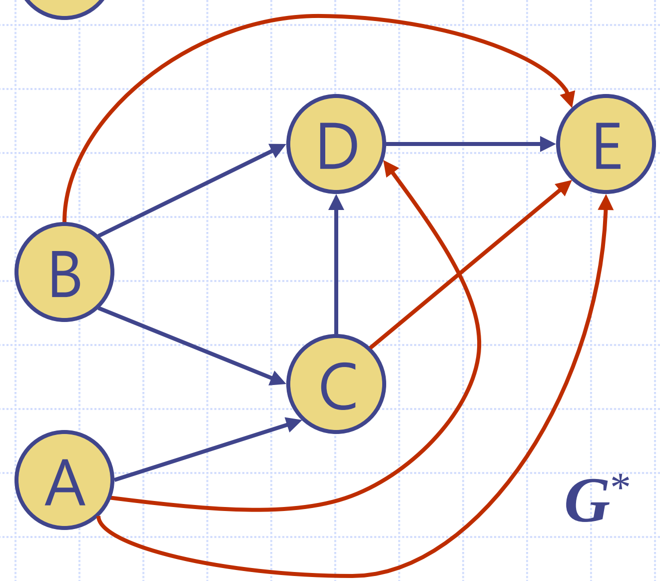
- 방향그래프에서 각 정점으로부터 다른 모든 정점으로 도달할 수 있는 최대의 부그래프
- DFS를 사용하여 **O(n + m)** 시간에 계산 가능(**biconnectivity**와 유사)



이행적폐쇄

- 주어진 방향그래프 ***G***에 대해, 그래프 ***G***의 **이행적폐쇄**(transitive closure): 다음을 만족하는 방향그래프 G^*
  - G^*는 ***G***와 동일한 정점들로 구성
  - ***G***에 ***u***로부터 $v \neq u$ 로의 방향경로가 존재한다면 G^*에 ***u***로부터 ***v***로의 방향간선이 존재
- **이행적폐쇄**는 방향그래프에 관한 도달 가능성 정보를 제공
- **예:** 컴퓨터 네트워크에서, "노드 ***a***에서 노드 ***b***로 메시지를 보낼 수 있을까?"



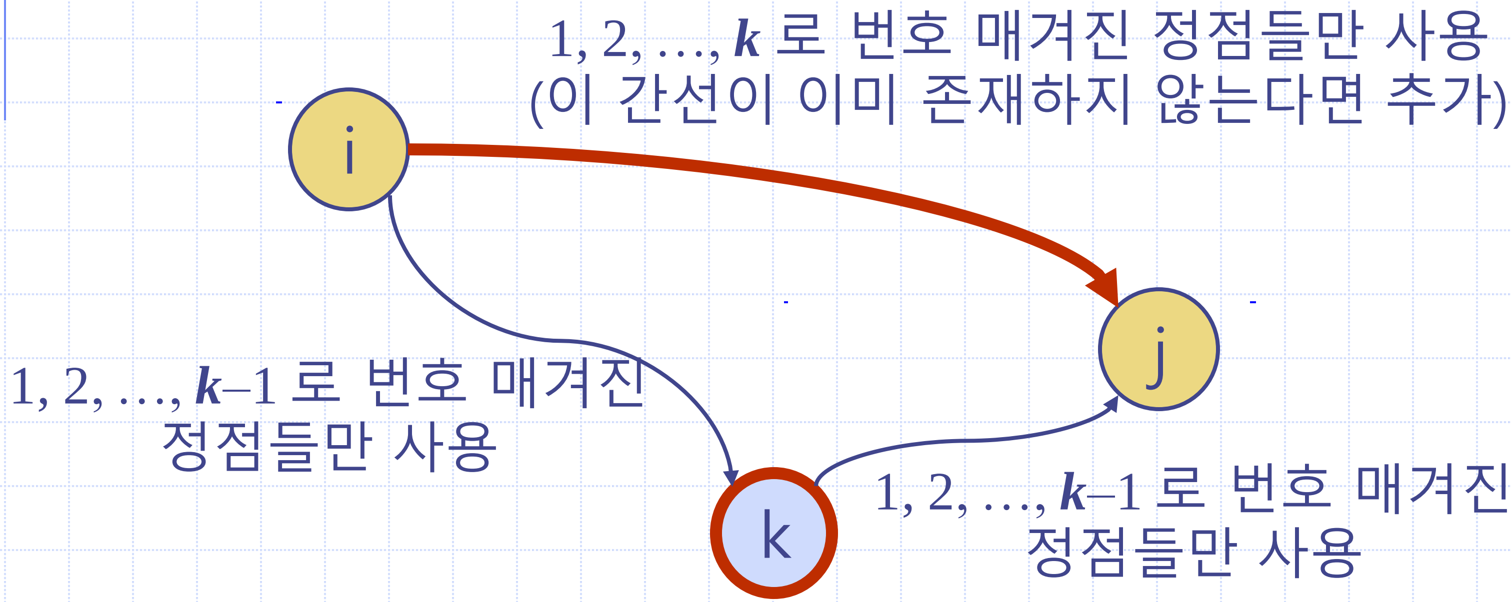


이행적폐쇄 계산

- 각 정점에서 출발하여 DFS를 수행할 수 있다
  - **실행시간:** $O(n(n+m))$
- 대안으로, **동적프로그래밍**(dynamic programming)을 사용할 수 있다: Floyd-Warshall의 알고리즘
  - **원리:** "A에서 B로 가는 길과 B에서 C로 가는 길이 있다면, A에서 C로 가는 길이 있다"

Floyd-Warshall 이행적폐쇄

1. 정점들을 1, 2, …, n 으로 번호를 매긴다
2. 1, 2, …, k 로 번호 매겨진 정점들만 경유 정점으로 사용하는 경로들을 고려

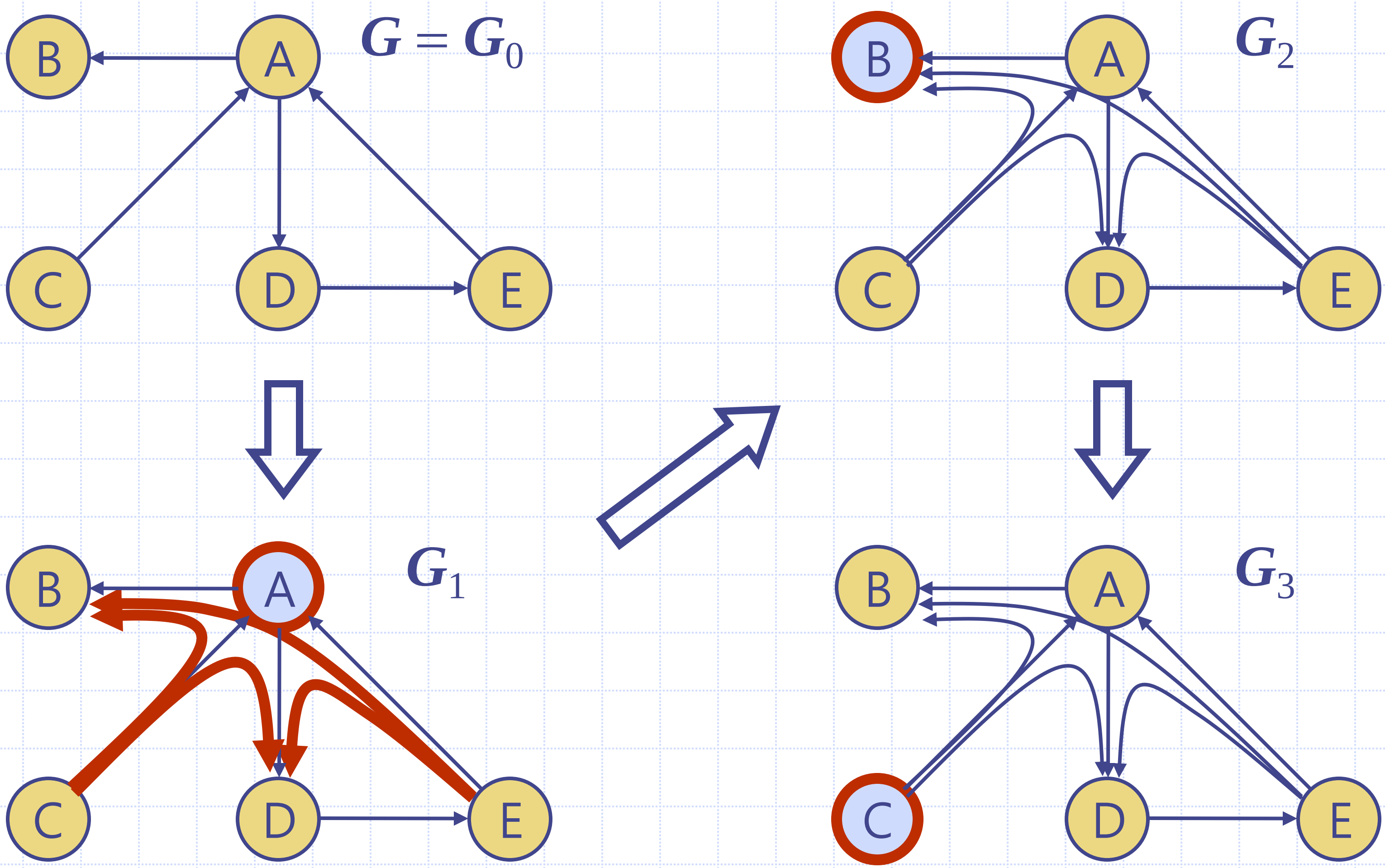


Floyd-Warshall 알고리즘

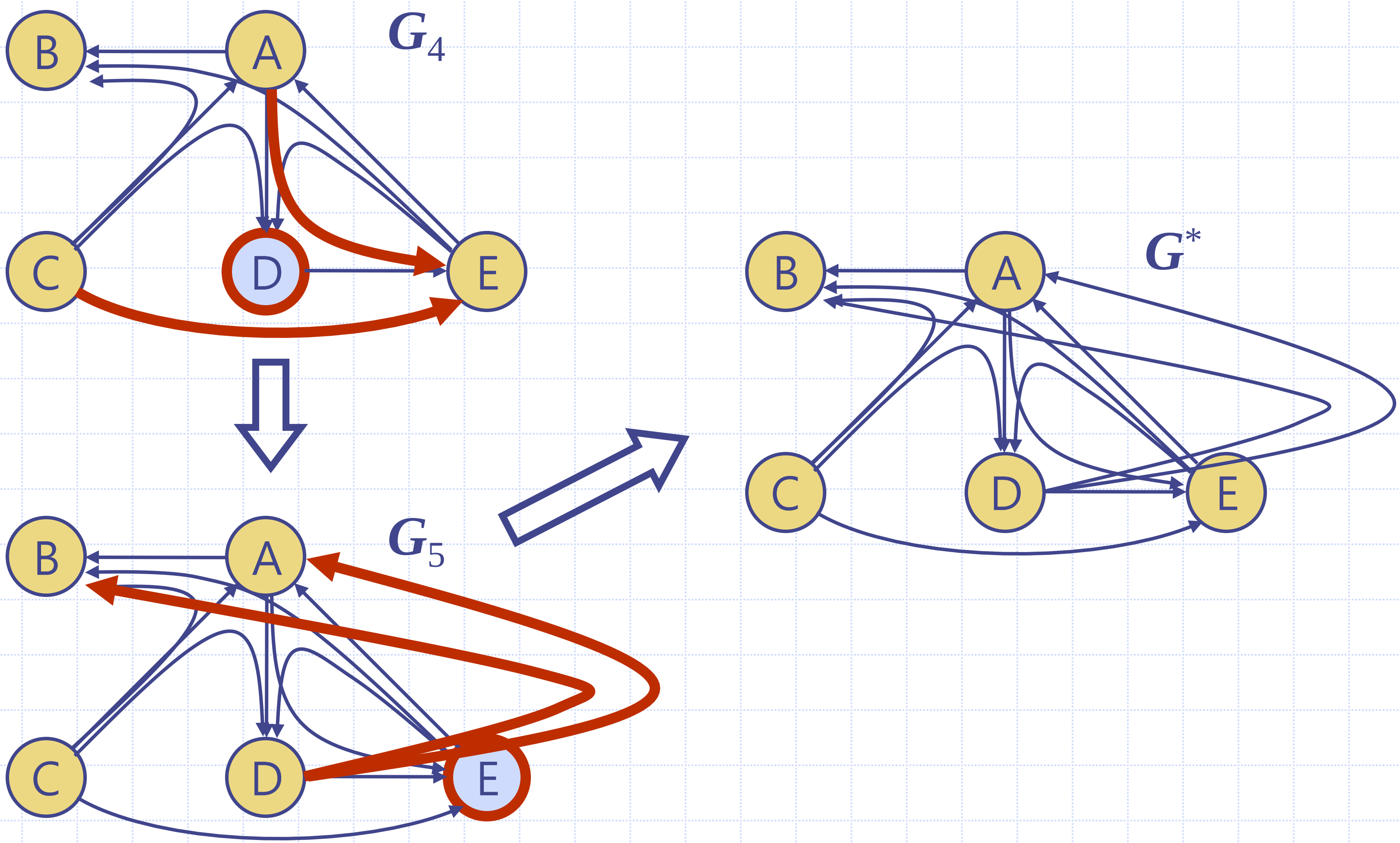
- Floyd-Warshall 알고리즘은 G의 정점들을 $v_1, \dots, v_n$로 번호 매긴 후, 방향그래프 $G_0, \dots, G_n$을 잇달아 계산
  - $G_0 = G$
  - G에 $\{v_1, \dots, v_k\}$ 집합 내의 경유정점을 사용하는, v_i에서 v_j로의 방향경로가 존재하면 G_k에 방향간선 (v_i, v_j)를 삽입
- k 단계에서, 방향그래프 G_{k-1}로부터 G_k를 계산
- 마지막에 $G_n = G^*$를 얻음
- **실행시간:** $O(n^3)$
  - **전제:** areAdjacent가 $O(1)$ 시간에 수행(즉, 인접행렬)

```
Alg Floyd-Warshall(G)
    input a digraph G with n vertices
    output transitive closure G* of G

1. Let v1, v2, …, vn be an arbitrary numbering of
    the vertices of G
2. G0 ← G
3. for k ← 1 to n                        {stopover vertex}
       Gk ← Gk−1
       for i ← 1 to n, i ≠ k             {start vertex}
           for j ← 1 to n, j ≠ i, k      {end vertex}
               if (Gk−1.areAdjacent(vi, vk) &
                       Gk−1.areAdjacent(vk, vj))
                   if (!Gk.areAdjacent(vi, vj))
                       Gk.insertDirectedEdge(vi, vj, k)
4. return Gn
```

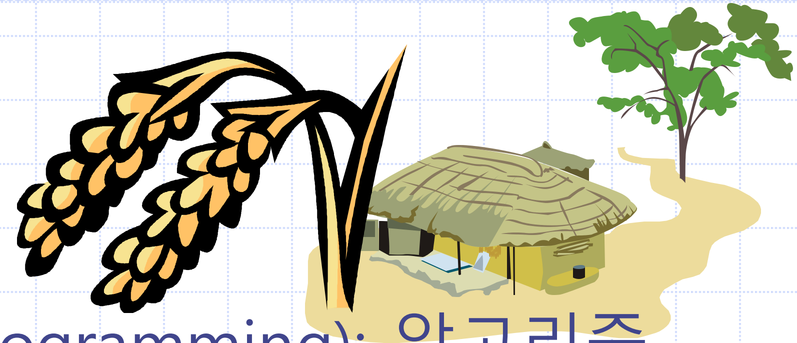
Floyd-Warshall 알고리즘 수행 예



Floyd-Warshall 알고리즘 수행 예 (conti.)



동적프로그래밍



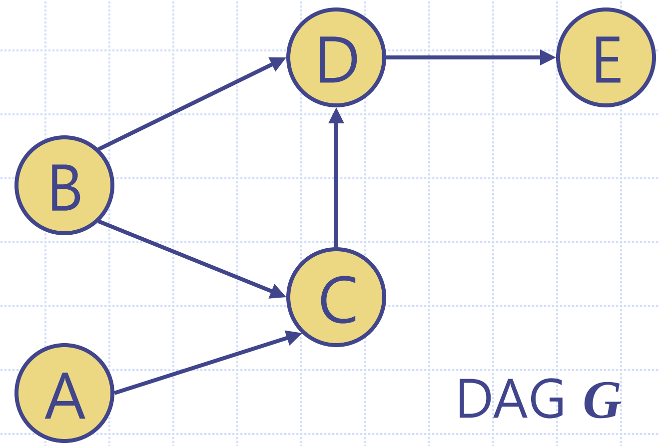
- **동적프로그래밍**(dynamic programming): 알고리즘 설계의 일반적 기법 가운데 하나
- 언뜻 보기에 많은 시간(심지어 지수 시간)이 소요될 것 같은 문제에 주로 적용 – 적용의 조건은:
  - **부문제 단순성**(simple subproblems): 부문제들이 j, k, l, m, 등과 같은 몇 개의 변수로 정의될 수 있는 경우
  - **부문제 최적성**(subproblem optimality): 전체 최적치가 최적의 부문제들에 의해 정의될 수 있는 경우
  - **부문제 중복성**(subproblem overlap): 부문제들이 독립적이 아니라 상호 겹쳐질 경우 – 따라서, 해가 "상향식"으로 구축되어야 함
- **예**
  - 피보나치 수열(Fibonacci progression)에서 n-번째 수 찾기
  - 그래프의 이행적폐쇄 계산하기

동적프로그래밍 vs. 분할통치법

- 공통점
  - 알고리즘 설계기법의 일종
  - 문제공간: **원점-목표점** 구조
    - **원점:** 문제의 초기 또는 기초 지점(복수 개수 가능)
    - **목표점:** 최종해가 요구되는 지점 (보통 1개)
    - 추상적 개념 상의 두 지점
- 차이점
  - 문제해결 진행 방향
    - **동적프로그래밍**(단방향): 원점 ⇒ 목표점
    - **분할통치**(양방향): 목표점 ⇒ 원점 ⇒ 목표점 (단, **해**를 구하기 위한 연산 진행 방향은 원점 ⇒ 목표점)
- 성능
  - **동적프로그래밍**
    - 단방향 특성때문에 종종 효율적
  - **분할통치**
    - 분할 회수
    - 중복연산 수행 회수

방향 비싸이클 그래프

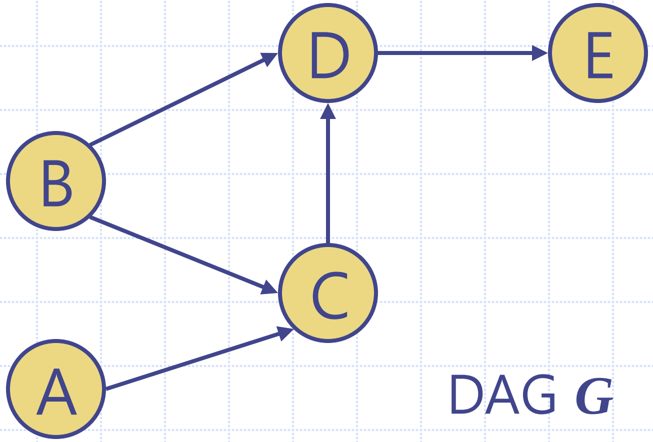
- **방향 비싸이클 그래프**(directed acyclic graph, **DAG**): 방향싸이클이 존재하지 않는 방향그래프
- **예**
  - C++ 클래스 간의 상속 또는 Java 인터페이스
  - 교과목 간의 선수 관계
  - 프로젝트의 부분 작업들 간의 스케줄링 제약
  - 사전의 용어 간의 상호의존성
  - 엑셀과 같은 스프레드시트에서 수식 간의 상호의존성



DAG *G*

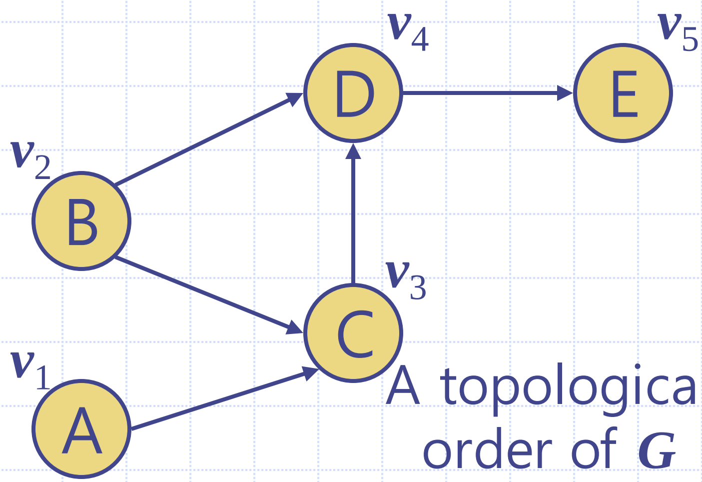
DAG와 위상 정렬

- 방향그래프의 **위상순서**(topological order)
  - 모든 $i < j$인 간선 (v_i, v_j)에 대해 정점들을 번호로 나열한 것
  - **예:** 작업스케줄링 방향그래프에서 위상순서는 작업들의 우선 순서 제약을 만족하는 작업 순서



DAG G

정리: 방향그래프가 DAG면, 위상순서를 가지며, 그 역도 참이다



A topological order of G

위상 정렬

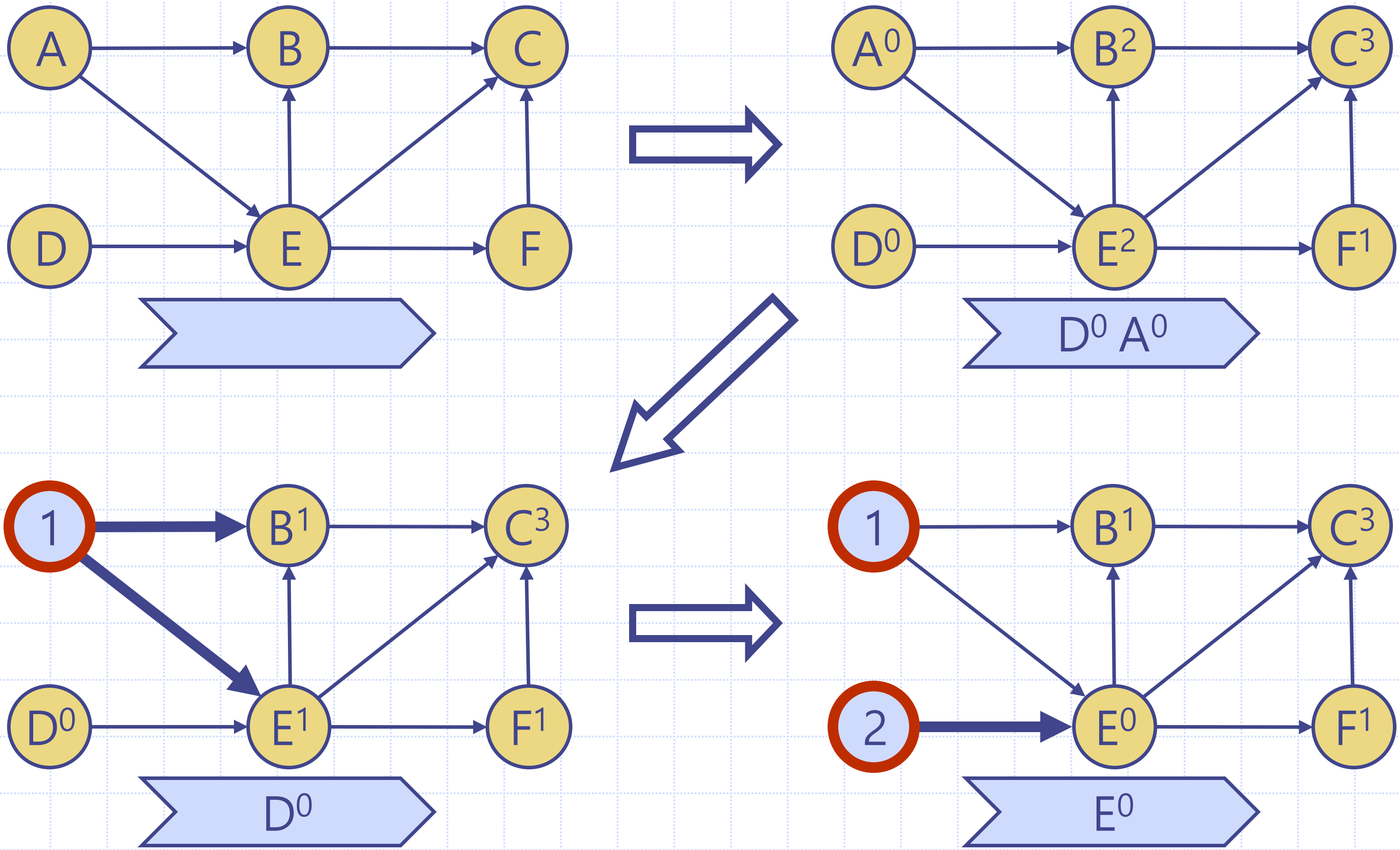
- **위상 정렬**(topological sort): DAG로부터 위상순서를 얻는 절차
- 알고리즘
  - topologicalSort: 정점의 진입차수(in-degree)를 이용
  - topologicalSortDFS: DFS의 특화

정점의 진입차수를 이용하는 위상 정렬

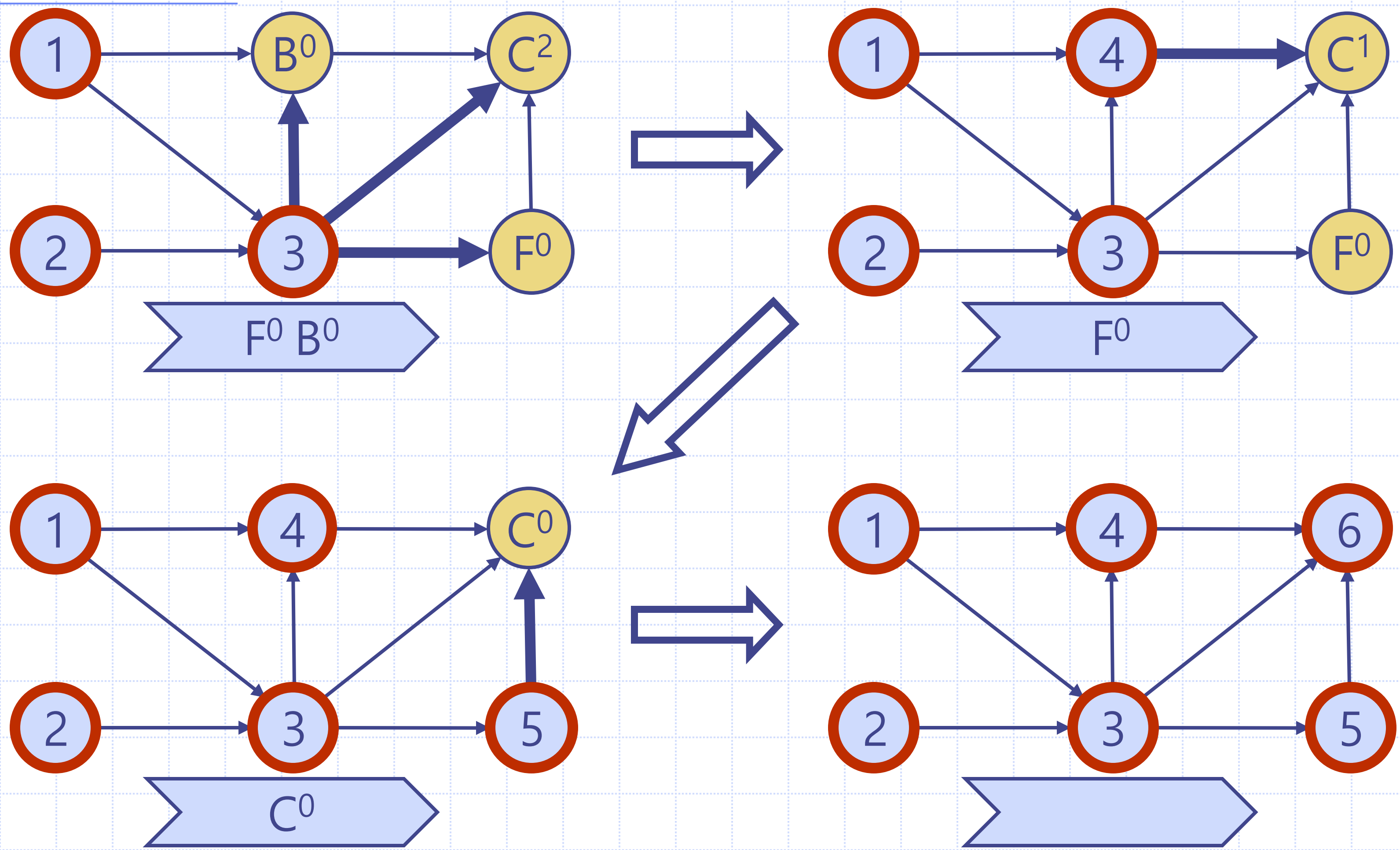
```
Alg topologicalSort(G)
    input a digraph G with n
        vertices
    output a topological ordering
        v_1 , …, v_n of G, or an
        indication that G has a
        directed cycle

1. Q ← empty queue
2. for each u ∈ G.vertices()
        in(u) ← inDegree(u)
        if (in(u) = 0)
            Q.enqueue(u)
3. i ← 1        {topological number}
4. while (!Q.isEmpty())
        u ← Q.dequeue()
        Label u with topological number i
        i ← i + 1
        for each e ∈ G.outIncidentEdges(u)
            w ← G.opposite(u, e)
            in(w) ← in(w) – 1
            if (in(w) = 0)
                Q.enqueue(w)
5. if (i ≤ n)   {i = n + 1, for DAG}
        write("G has a directed cycle")
6. return
```

- 각 정점에 새로운 라벨을 정의
  - **현재 진입차수**(inCount): $in(v)$, 정점 v의 현재의 진입차수

위상 정렬 수행 예



위상 정렬 수행 예 (conti.)



DFS를 특화한 위상 정렬

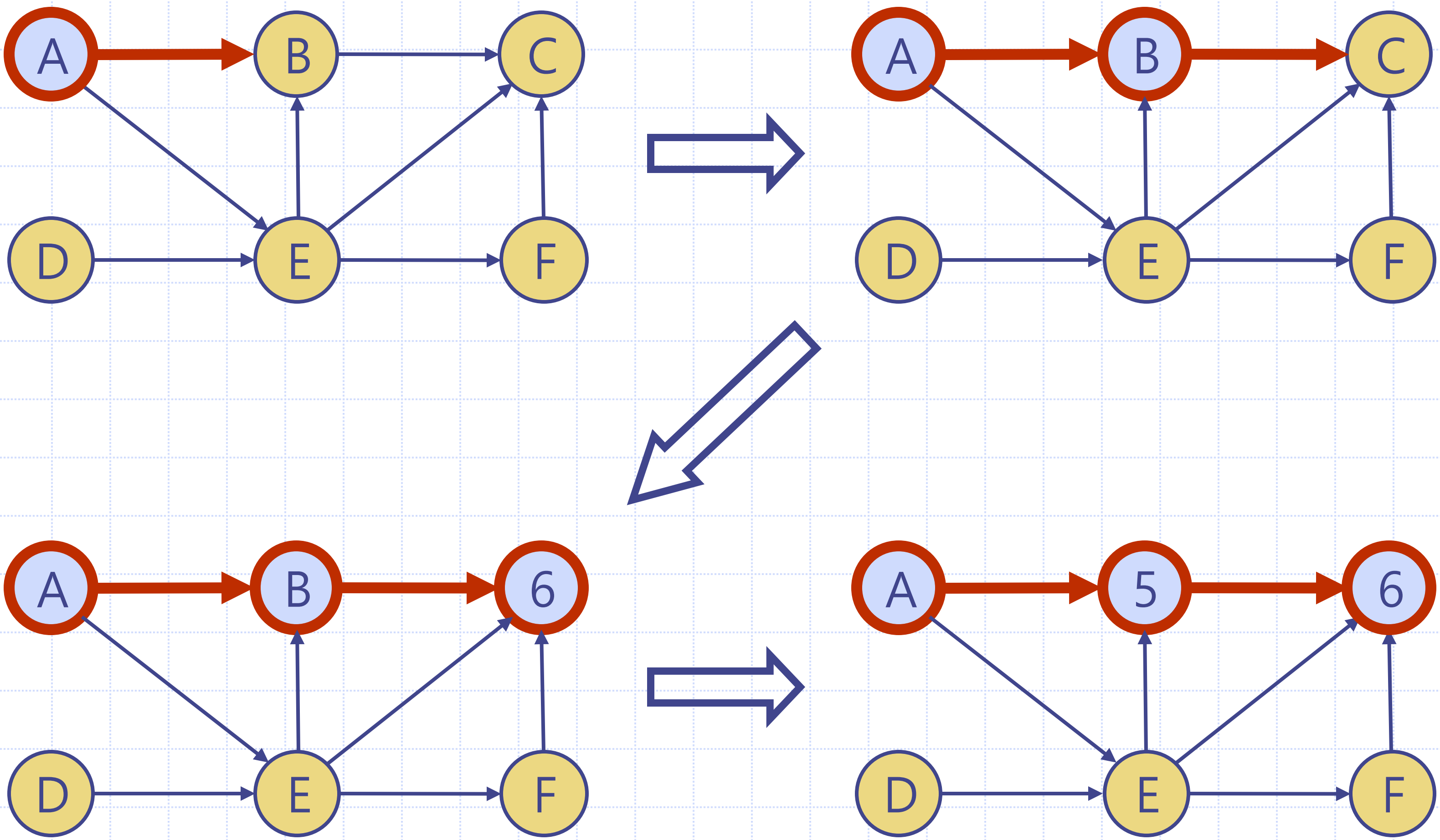
Alg *topologicalSortDFS(G)*
input dag G
output topological ordering of G

1. $n \leftarrow G.numVertices()$
2. **for each** $u \in G.vertices()$
   $l(u) \leftarrow Fresh$
3. **for each** $v \in G.vertices()$
   if $(l(v) = Fresh)$
   $rTopologicalSortDFS(G, v)$

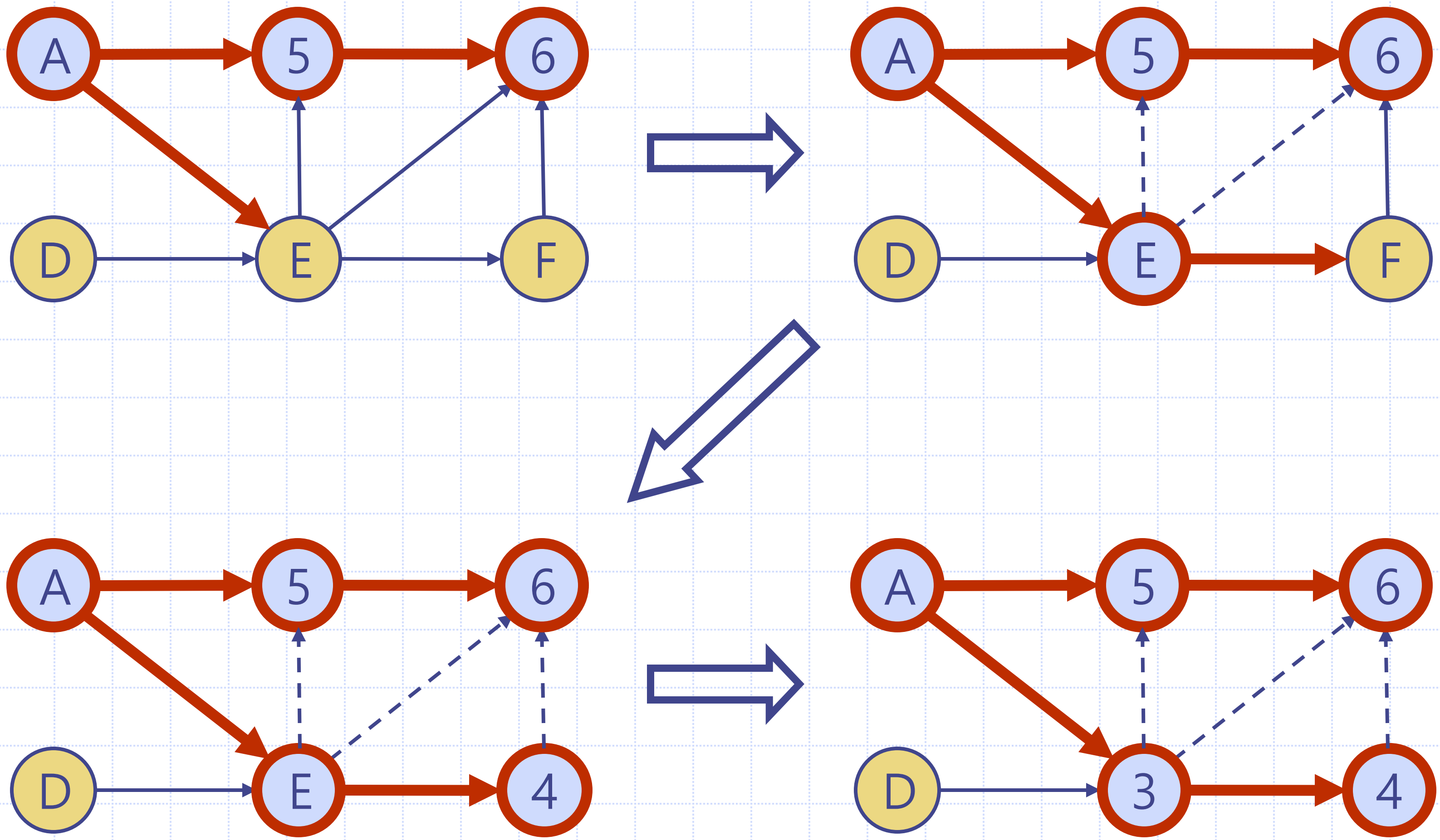
Alg *rTopologicalSortDFS(G, v)*
input graph G, and a start vertex v of G
output labeling of the vertices of G in the connected component of v

1. $l(v) \leftarrow Visited$
2. **for each** $e \in G.outIncidentEdges(v)$
   $w \leftarrow opposite(v, e)$
   if $(l(w) = Fresh)$ {e is a tree edge}
   $rTopologicalSortDFS(G, w)$
   elseif w *is not labelled with a topological number*
   write(*"G has a directed cycle"*)
   {**else**
   e is a nontree edge}
3. *Label* v *with topological number* n
4. $n \leftarrow n - 1$

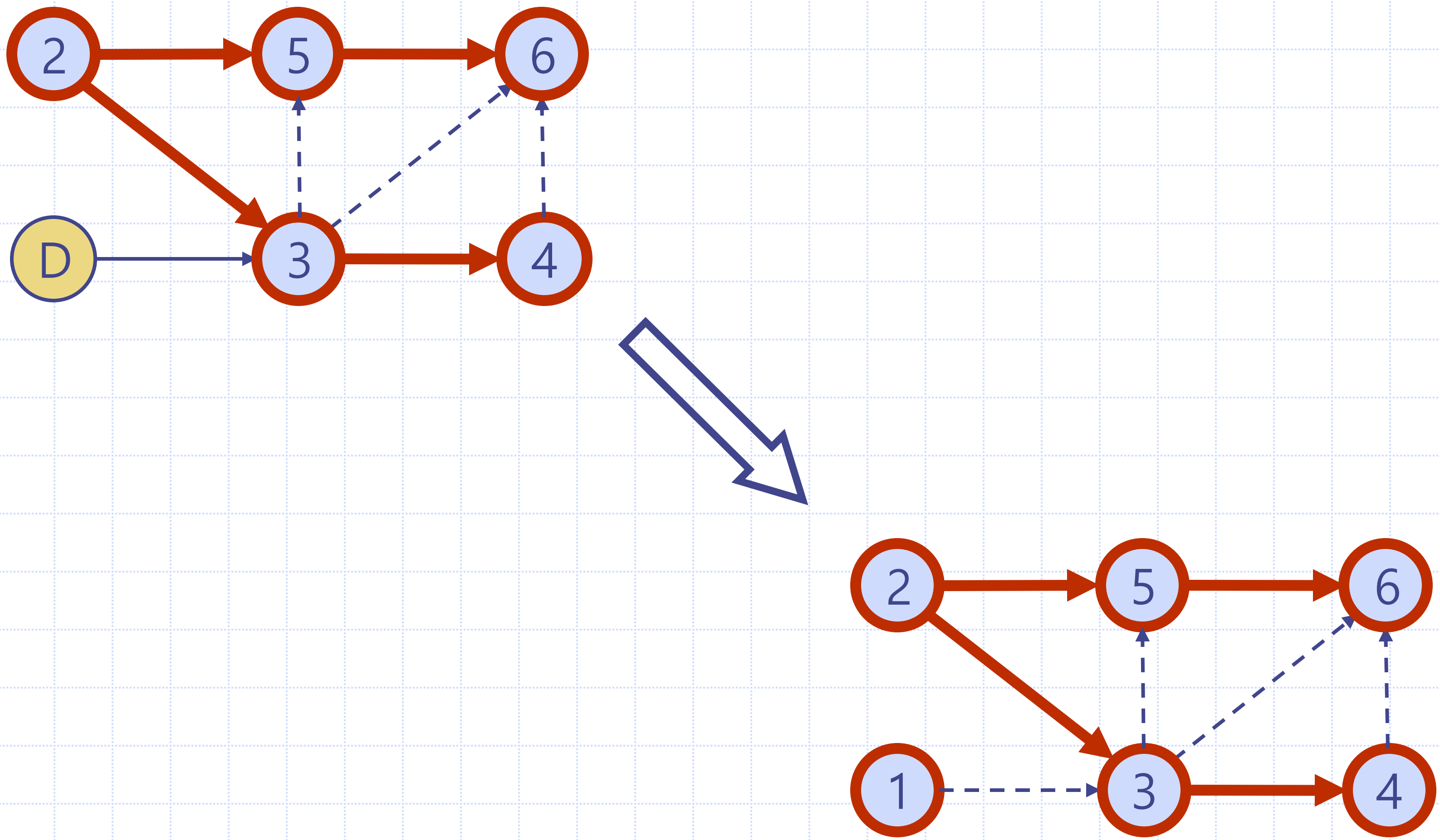
위상 정렬 수행 예



위상 정렬 수행 예 (conti.)



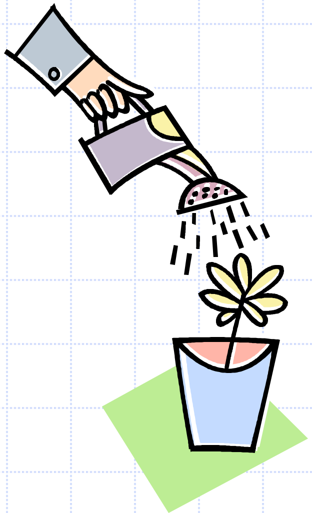
위상 정렬 수행 예 (conti.)



위상 정렬 알고리즘 분석

- 두 개의 버전 모두
  - $\mathbf{O}(\boldsymbol{n} + \boldsymbol{m})$ 시간과 $\mathbf{O}(\boldsymbol{n})$ 공간 소요
  - $\boldsymbol{G}$가 DAG인 경우, $\boldsymbol{G}$의 위상순서를 계산
  - $\boldsymbol{G}$에 방향싸이클이 존재할 경우, 일부 정점의 순위를 매기지 않은 채로 정지

응용문제: 그래프 키우기



- 동적으로 커가는 방향그래프 $G = (V, E)$를 지원할 데이터구조를 설계하고자 한다
- 초기에는 $V = \{1, 2, \ldots, n\}$이며 $E = \varnothing$이다
- 사용자는 다음 작업을 통해 그래프를 확장한다
  - insertDirectedEdge(u, v): G에 정점 u에서 v로 향하는 방향간선을 삽입, 즉 $E \leftarrow E \cup \{(u, v)\}$
- 여기에다, 사용자는 아무 때나 그래프의 두 정점이 연결되었는지 질의할 수 있다
  - reachable(u, v): 정점 u에서 v로 도달 가능한지, 즉 방향경로가 존재하는지 여부를 반환

응용문제: 그래프 키우기 (conti.)

- 사용자는 그래프가 **완전히 연결**될 때까지 그래프를 키워 나간다
- 그래프 내 간선의 수는 단순 증가하며, 사용자는 동일한 간선을 두 번 이상 추가하지 않으므로, insertDirectedEdge 작업의 총 수는 정확히 $n(n-1)$
- 그래프를 키우는 동안, 사용자는 $n(n-1)$회의 insertDirectedEdge 작업에 p회의 reachable 작업을 **섞어** 수행
- 위의 작업 과정 전체를 효율적으로 지원하는 **데이터구조**를 설계하라

해결: 문제해결 개요

- 이 문제를 해결하기 위해서, 각 정점쌍 간에 방향경로가 있는지 추적하는 $n \times n$ 크기의 **이행적폐쇄** 행렬 T를 유지
- 1회의 reachable 작업을 $\mathbf{O}(1)$ 실행시간에, 그리고 $n(n-1)$회의 insertDirectedEdge 작업을 총 $\mathbf{O}(n^3)$ 최악실행시간에 수행하는 알고리즘을 작성할 것이다
- 이 두 실행시간을 합치면, $n(n-1)$회의 insertDirectedEdge 작업과 m회의 reachable 작업을 어떤 순서로 하더라도 $\mathbf{O}(n^3 + p)$ 시간에 수행
- 이후, p가 작은 경우에 대하여, 실행시간이 $\mathbf{O}(min(n^3 + p, n^2p))$ 으로 개선된 데이터구조를 제시한다

해결: 이행적폐쇄 행렬

- 우선, 데이터구조에 다음과 같은 **이행적폐쇄** 행렬 T를 유지
  - G의 u에서 v로 방향경로가 존재하면, $T[u, v] = 1$
  - 그렇지 않으면, $T[u, v] = 0$
- **인접행렬**과 차이점: 행렬 T는 u에서 v로 향하는 **간선**의 존재를 추적하는 대신, u에서 v로 향하는 **경로**의 존재를 추적
- 참고로, 행렬의 u-번째 **행**에 있는 1들은 u가 도달할 수 있는 정점들을 나타내며, 행렬의 u-번째 **열**에 있는 1들은 u에 도달할 수 있는 정점들을 나타낸다
- 모든 정점 u에서 스스로에게 (간선 없는) 방향경로가 존재하므로, $T[u, u]$ 는 1로 초기화

해결: reachable, insertDirectedEdge 설계

- T가 주어지면 reachable(u, v) 구현은 $T[u, v]$에 대한 조회로 충분
  - 이 질의는 상수시간에 수행되므로 reachable은 상수시간에 수행
- insertDirectedEdge(u, v)의 핵심
  - 간선 (u, v)가 추가될 때, 모든 정점 x를 검사
  - 만약 x가 u에 도달하지만 v에는 도달하지 못하면, (방금 추가된 간선에 의해) v가 도달하는 모든 정점에 x도 도달할 수 있다는 것을 나타내도록 행렬을 갱신

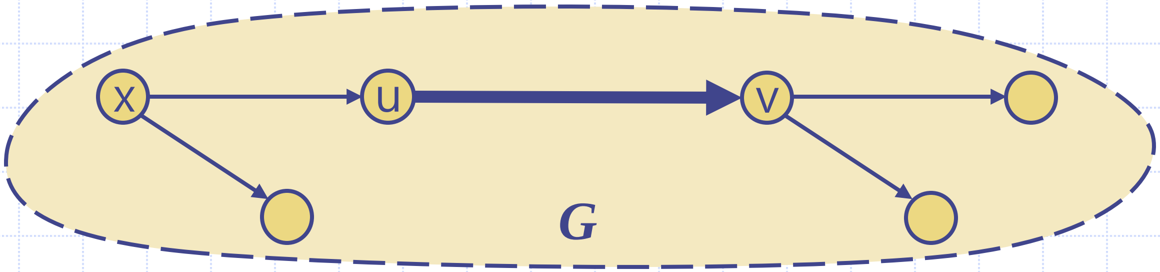
Alg ***reachable*****(*u*, *v*)**
    input transitive closure ***T***, vertex ***u*, *v***
    output boolean

1. **return** $T[u, v]$

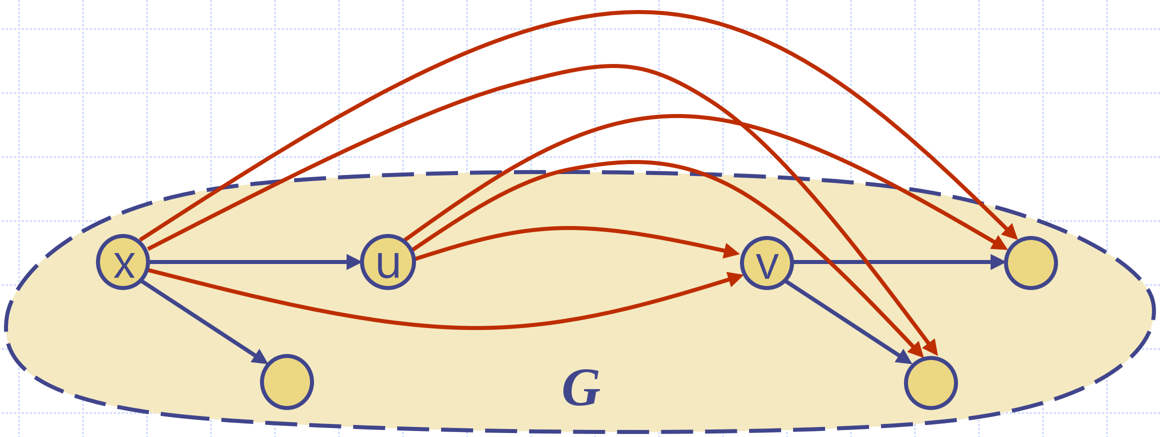
Alg ***insertDirectedEdge*****(*u*, *v*)**
    input transitive closure ***T***, vertex ***u*, *v***
    output none

```
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]
```

해결: insertDirectedEdge



(a) 방향간선 (u, v) 삽입



(b) 삽입에 따른 방향경로 갱신

해결: 알고리즘 성능

- 각 reachable(u, v) 작업은 배열 데이터 조회에 불과하므로 $\mathbf{O}(1)$ 시간에 수행
- insertDirectedEdge은 **중첩반복문**이므로 단순히 $\mathbf{O}(n^2)$ 최악실행시간으로 분석할 수도 있지만, 총 $n(n-1)$회의 insertDirectedEdge 작업에 대한 **종합분석**이 타당
  - 1회의 insertDirectedEdge 작업이 수행될 때마다 바깥 루프가 n개의 정점에 대해 수행되므로, 종합적으로는 $\mathbf{O}(n^3)$ 시간에 수행
  - 내부 루프는 $T[x, v] = 0$일 때만 수행되며 수행이 끝나면 $T[x, v] = 1$이 된다
  - 따라서 특정 정점 x에 대해 내부 루프는 최대 n회 수행 – 실제로는 초기값 $T[x, x] = 1$이므로 $n - 1$회 수행
  - 정점이 n개 있으므로, 내부 루프는 종합적으로 최대 n^2회 수행
  - 따라서 전체적으로 최악의 경우 $\mathbf{O}(n^3)$ 시간
- 그러므로 총 $n(n-1)$회의 insertDirectedEdge 작업과 p회의 reachable 작업은 $\mathbf{O}(n^3 + p)$ 시간에 수행

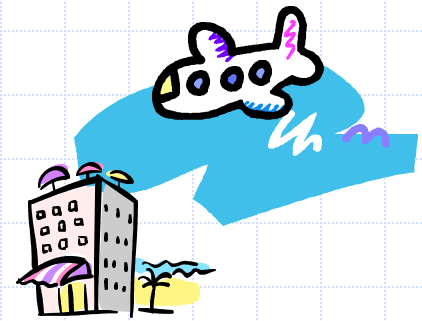
해결: 행렬을 인접리스트로 대체하여 성능 개선 시도

- 두 번째 가능한 데이터구조로써 **인접리스트**를 사용하면,
  - insertDirectedEdge는 $\mathbf{O}(1)$ 시간에,
  - reachable은 $\mathbf{O}(n^2)$ 시간에 수행 가능
- 여기서는 크기 n의 배열 $A[0..n-1]$을 유지
  - A의 각 원소는 각 정점의 진출간선들에 대한 연결리스트를 유지
- insertDirectedEdge(u, v): $\mathbf{O}(1)$
  - $A[u]$의 맨 앞 또는 맨 뒤에 간선 (u, v)를 삽입
- reachable(u, v): $\mathbf{O}(n^2)$
  - 정점 u로부터 시작하는 (DFS 또는 BFS와 같은) 탐색을 통해 수행
  - 탐색 과정에서 v를 만나면 ***True***를, 그렇지 않으면 ***False***를 반환
  - 완전 연결그래프에서 DFS 또는 BFS는 $\mathbf{O}(n + m) = \mathbf{O}(n^2)$ 시간에 수행
- 따라서 **종합 수행시간:** $\mathbf{O}(n^2 + n^2p)$
- 첫 번째 vs. 두 번째 데이터구조 – p의 크기에 따라 유불리
  - $p \gg n$: 가능한 가정, 즉 각 정점에 대해 최소 한 번씩 질의한다는 가정
  - $p \gg n^2$: 또 다른 가정
  - p를 미리 안다면, 두 가지 데이터구조 중에 유리한 것을 선택

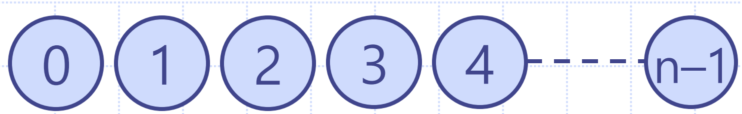
해결: 두 데이터구조를 혼용

- p를 미리 모른다고 해도, 두 가지 데이터구조 각각의 장점을 살리기 위해 둘을 **혼합** 사용 가능
- **혼합 전략:** n회의 질의(reachable)가 행해질 때까지는 **인접리스트** 구조를 사용하다가, n-번째 질의가 행해질 때 **이행적폐쇄 행렬**을 구축하고 이후 작업부터는 이를 사용
  - 행렬의 구축은, 각 정점 u로부터 DFS 또는 BFS를 수행하여 도달 가능한 모든 정점 v를 $T[u, v] \leftarrow 1$로 표시하면 $O(n^3)$ 시간에 수행 가능
- 따라서, $p \le n$이면,
  - **인접리스트**만 사용함으로써, 총 $O(n^2p)$ 실행시간을 얻는다
- 반대로, $p \ge n$이면,
  - 우선 **인접리스트**를 사용함으로써, 총 $O(n^3)$의 작업을 완수한 후,
  - $O(n^3)$ 시간을 사용하여 **이행적폐쇄 행렬**로 전환하고,
  - 이후의 모든 작업에는 **행렬**을 사용
  - 그리하면 총 $O(n^3 + p)$ 실행시간을 얻는다
- 그러므로, 이 데이터구조를 통해 $O(min(n^3 + p, n^2p))$ 최악실행시간을 얻을 수 있다

응용문제: 에어텔

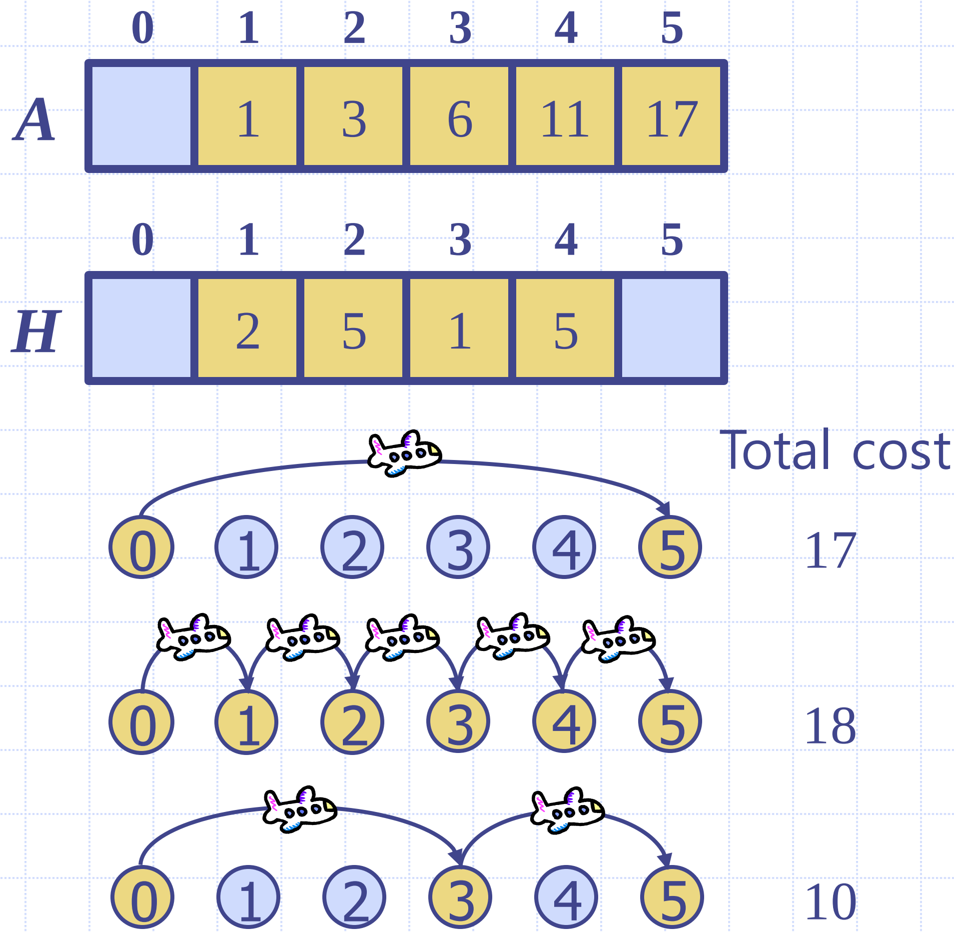


- **배경**
  - 당신은 n개의 도시가 있는 나라에 살고 있다
  - 도시들은 일직선상에 위치하며 0부터 $n-1$까지 번호가 매겨져 있다
- **제약**
  - 도시 0에서 출발하여 도시 $n-1$로 가고자 한다
  - 도시와 도시 사이는 오른쪽으로만, 그리고 항공편으로만 이동해야 하며 하루에 한 개의 항공편만 탈 수 있다
  - 항공편이 도착한 도시에서는 반드시 그 도시의 호텔에서 1박해야 하며 다음날 아침 새로운 항공편으로 여행을 계속한다
- **전제**
  - **항공요금**은 도시 구간이 멀수록 비싸며 i $(1 \le i \le n-1)$ 구간에 대한 항공요금은 배열 A의 $A[i]$ 원소값으로 주어진다
  - 각 도시의 호텔 **숙박요금**은 배열 $H[1:n-2]$에 주어진다(도시 0과 $n-1$의 숙박비는 계산에 불포함)



응용문제: 에어텔 (conti.)

- **문제:** 여행의 **최소비용**을 구하는 알고리즘을 작성하라
- **예:** $n = 6$
  - 최소 비용 = 10

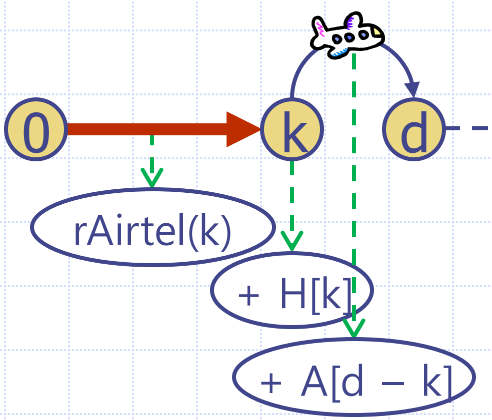


해결: 개요

- **분할통치법 vs. 동적프로그래밍**에 의한다
- 문제공간: 1D 공간 상의 n개 도시
- 두 전략 각각 **정방향** 또는 **역방향** 해결 가능
  - **정방향:** 출발도시를 0(**원점**)으로 고정하고, 도착도시를 출발도시에서 가장 가까운 도시 1부터 가장 먼 도시 $n-1$(**목표점**) 쪽으로 변경하면서 해를 구한다
  - **역방향:** 도착도시를 $n-1$(**원점**)로 고정하고, 출발도시를 도착도시에서 가장 가까운 도시 $n-2$부터 가장 먼 도시 0(**목표점**) 쪽으로 변경하면서 해를 구한다
- 계산의 방향만 반대일 뿐, 정방향과 역방향 모두 나름의 **원점**에서 **목표점** 방향으로 해를 구하는 연산을 수행하여 동일한 최종해를 구한다
- **분할통치**는 **재귀** 방식의 알고리즘이므로, 해를 구하는 순서는 재귀호출로부터의 **반환** 순서와 **일치**하며, **재귀호출**의 순서와는 **반대**
- **동적프로그래밍**은 **비재귀** 방식의 알고리즘이므로, 해를 구하는 순서는 위에 말한 계산 순서와 그대로 **일치**
- 계산 편의 상, $H[0]$과 $H[n-1]$에 0을 저장

해결: 분할통치법 (정방향)

- 도착도시 d에 대해, 도시 $k(0 \le k \le d-1)$를 경유할 경우의 총비용 $cost$를 계산하여 그 가운데 최소값을 찾는다
- 총 $\mathbf{O}(2^n)$ 시간 소요



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

1. **return** $rAirTel(n-1)$

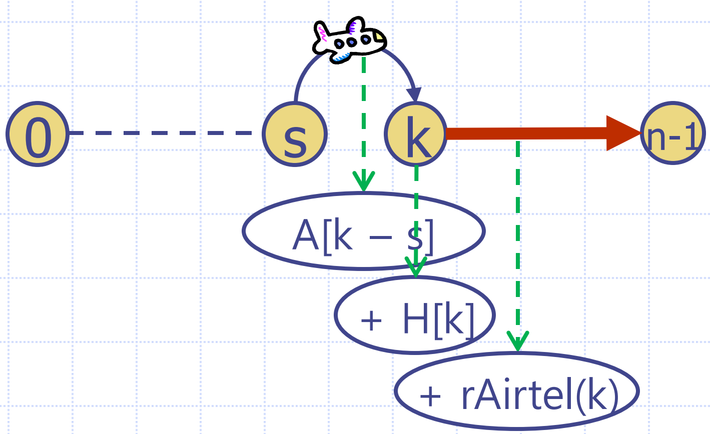
Alg *rAirtel(d)*
  input destination city d
  output minimum cost of travel from city 0 to d

1. **if** $(d = 0)$
   return 0
2. $mincost \leftarrow \infty$
3. **for** $k \leftarrow 0$ **to** $d-1$ {stopover}
   $cost \leftarrow rAirtel(k) + H[k] + A[d-k]$
   $mincost \leftarrow min(mincost, cost)$
4. **return** $mincost$

{Total $\mathbf{O}(2^n)$}

해결: 분할통치법 (역방향)

- 출발도시 s에 대해, 도시 k($s + 1 \le k \le n - 1$)를 경유할 경우의 총비용 $cost$를 계산하여 그 가운데 최소값을 찾는다
- 총 $\mathbf{O}(2^n)$ 시간 소요



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

1. **return** ***rAirtel***(0)

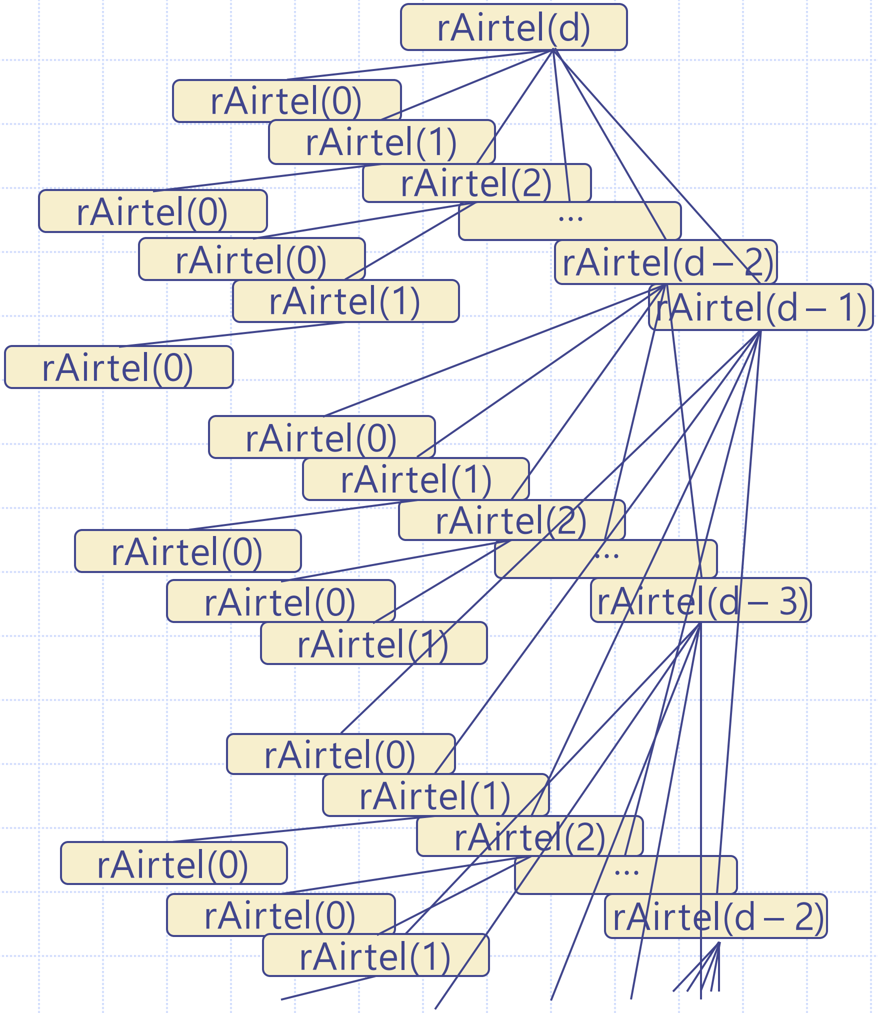
Alg ***rAirtel***(s)
    input start city s
    output minimum cost of travel from city s to $n - 1$

1. **if** ($s = n - 1$)
        return 0
2. $mincost \leftarrow \infty$
3. **for** $k \leftarrow s + 1$ **to** $n - 1$ {stopover}
        $cost \leftarrow A[k - s] + H[k] + rAirtel(k)$
        $mincost \leftarrow min(mincost, cost)$
4. **return** $mincost$

{Total $\mathbf{O}(2^n)$}

해결: 분할통치법의 성능

- **문제점**: 에어텔 문제에 대한 **분할통치** 방식 해결은 과도한 **중복 호출**(즉, 동일한 매개변수를 가진 호출이 셀 수 없이 중복됨)로 인해 **효율**이 크게 저하(오른 편의 정방향 해결 버전 호출절차 그림 참고)
- 총 **O**(2^n) 시간 소요
- **동적프로그래밍** 방식에서는, 크기 n의 배열 m을 사용하여 중간 계산값들을 저장하여 사용함으로써 한 번 계산한 배열원소에 대한 중복계산을 방지



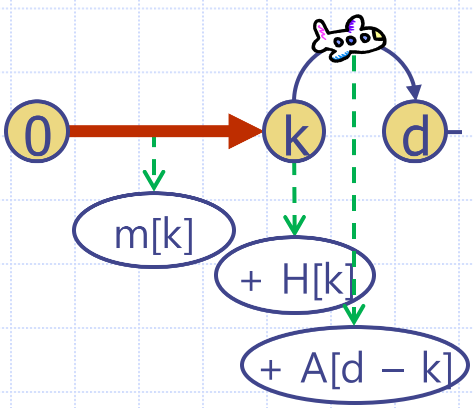
해결: 동적프로그래밍 (정방향)

- 도착도시 0에 대한 최소비용 $m[0]$을 0으로 초기화
- 도착도시 $d(1 \le d \le n-1)$에 대해, 도시 $k(0 \le k \le d-1)$를 경유할 경우의 총비용 $cost$를 계산하여 그 가운데 최소값을 찾아 $m[d]$에 저장
- 총 $O(n)$ 공간, $O(n^2)$ 시간 소요

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

1. $m[0] \leftarrow 0$
2. **for** $d \leftarrow 1$ **to** $n-1$ {compute $m[d]$}
   $m[d] \leftarrow \infty$
   for $k \leftarrow 0$ **to** $d-1$ {stopover}
     $cost \leftarrow m[k] + H[k] + A[d-k]$
     $m[d] \leftarrow min(m[d], cost)$
3. **return** $m[n-1]$

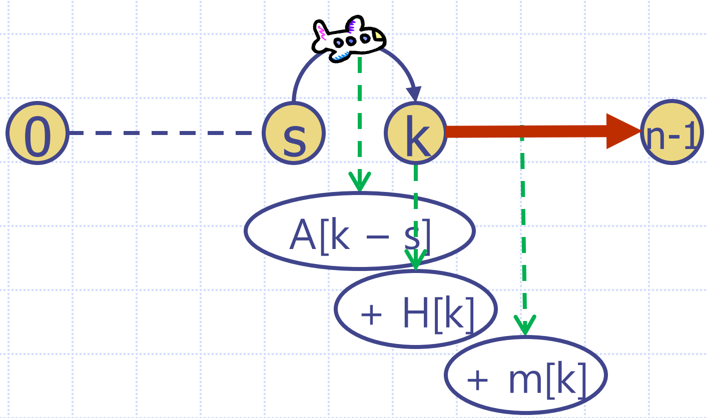
{Total $O(n^2)$}



- $m[d]$ = 도시 0에서 d로 가는 최소비용
- $\therefore m[n-1]$ = 도시 0에서 $n-1$로 가는 최소비용

해결: 동적프로그래밍 (역방향)

- 출발도시 $n-1$에 대한 최소비용 $m[n-1]$을 0으로 초기화
- 출발도시 $s(n-2 \geq s \geq 0)$에 대해, 도시 $k(s+1 \leq k \leq n-1)$를 경유할 경우의 총비용 *cost*를 계산하여 그 가운데 최소값을 찾아 $m[s]$에 저장
- 총 $\mathbf{O}(n)$ 공간, $\mathbf{O}(n^2)$ 시간 소요



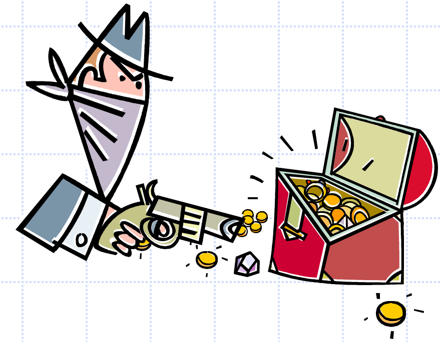
Alg *airtel*(n) {dynamic programming, backward ver.}
  input integer n
  output minimum cost of travel from city 0 to $n-1$

1. $m[n-1] \leftarrow 0$
2. **for** $s \leftarrow n-2$ **downto** 0 {compute $m[s]$}
   $m[s] \leftarrow \infty$
   for $k \leftarrow s+1$ **to** $n-1$ {stopover}
     $cost \leftarrow A[k-s] + H[k] + m[k]$
     $m[s] \leftarrow min(m[s],\ cost)$
3. **return** $m[0]$

{Total $\mathbf{O}(n^2)$}

- $m[s]$ = 도시 s에서 $n-1$로 가는 최소비용
- $\therefore m[0]$ = 도시 0에서 $n-1$로 가는 최소비용

응용문제: 금화 강도



- 에어텔 문제의 **2D** 버전
- 당신이 살고 있는 **Sin City**의 도로 지도가 $n \times n$개의 셀로 이루어진 정방형 격자 A로 주어져 있다
- 각 셀 $[i, j]$마다 $A[i, j] \geq 0$의 **금화**를 뺏어가는 **강도**들이 있다
- 좌상 셀 $[0, 0]$에서 출발하여 우하 셀 $[n-1, n-1]$로 택시로 여행한다
- 택시는 직진운행만 가능하며, 한 번에 여러 셀씩($1 \leq i \leq n-1$) **오른쪽** 또는 **아래** 방향으로만 이동할 수 있다
- 택시 승차 중에는 괜찮지만 그렇지 않은 곳에서는 그 셀에 있는 숫자 만큼의 금화를 강도에게 뺏긴다

- **예:** 아래 8×8 격자 A에서 뺏길 수 있는 금화의 최소량은 20이다

	0	1	2	3	4	5	6	7
0	1	3	7	2	11	17	16	25
1	6	2	3	4	7	2	12	15
2	11	4	6	8	8	1	9	14
3	20	8	8	11	6	3	3	9
4	0	10	9	8	7	15	17	22
5	17	12	7	10	3	1	8	13
6	19	25	10	15	14	11	3	3
7	21	18	16	20	15	13	19	0

A

응용문제: 금화 강도 (conti.)

- **문제:** 최적의 경로를 따라 뺏기는 금화의 **최소량**을 찾는 알고리즘의 **분할통치법** 버전과 **동적프로그래밍** 버전을 각각 작성하라
- **힌트: 에어텔** 문제와 마찬가지로, **정방향** 또는 **역방향**으로 진행하면서 해결할 수 있다
- **참고:** 각 셀에서 뺏기는 금화를 각 교차로에서 택시를 환승하는데 소요되는 시간으로 본다면 이 문제는 총 환승시간이 가장 짧은 길을 찾는 문제가 된다

해결: 개요

- **분할통치법 vs. 동적프로그래밍**에 의한다
- 문제공간: 2D 공간($n \times n$)
- 각각 **정방향** 또는 **역방향** 해결 가능
  - **정방향:** 출발셀을 셀 $[0, 0]$(**원점**)으로 고정하고, 도착셀을 출발셀에서 가장 가까운 셀부터 가장 먼 셀 $[n-1, n-1]$(**목표점**) 쪽으로 변경하면서 해를 구한다
  - **역방향:** 도착셀을 셀 $[n-1, n-1]$(**원점**)로 고정하고, 출발셀을 도착셀에서 가장 가까운 셀부터 가장 먼 셀 $[0, 0]$(**목표점**) 쪽으로 변경하면서 해를 구한다

해결: 분할통치법 (정방향)

- **부문제 정의:** $m(i, j)$를 셀 [0, 0]에서 출발하여 셀 [i, j]에 도달할 때까지 뺏길 수 있는 최소 금화량이라 하면 다음이 성립

  $k(j-1 \geq k \geq 0)$에 대해, 최소의
  $m(i, k) + A[i, j]$가 $minright$이고,

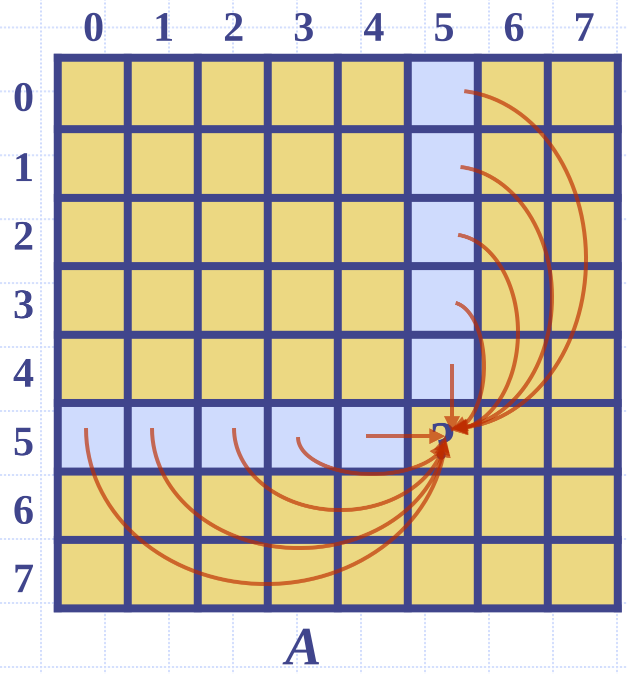
  $k(i-1 \geq k \geq 0)$에 대해, 최소의
  $m(k, j) + A[i, j]$가 $mindown$이면,

  $m(i, j)$는 $minright$와 $mindown$ 중 최소값

- **베이스 케이스**

  $m(0, 0) = A[0, 0]$

- 2^n개의 **재귀호출**이 일어나므로, 전체적으로 $\mathbf{O}(2^n)$ 시간 소요!



해결: 분할통치법 (정방향)

Alg *minGold*(A, n)
{divide and conquer, forward ver.}
input array A of $n \times n$ gold coins
output minimum possible gold coins moving from [0, 0] to [$n - 1$, $n - 1$]

1. **return** $m(n - 1, n - 1)$

Alg *m*(i, j)
input index i, j
output minimum possible gold coins moving from [0, 0] to [i, j]

1. **if** $((i = 0)$ & $(j = 0))$
   return $A[0, 0]$
2. $minright \leftarrow \infty$
3. **for** $k \leftarrow j - 1$ **downto** 0 {move right}
   $cost \leftarrow m(i, k) + A[i, j]$
   $minright \leftarrow min(minright, cost)$
4. $mindown \leftarrow \infty$
5. **for** $k \leftarrow i - 1$ **downto** 0 {move down}
   $cost \leftarrow m(k, j) + A[i, j]$
   $mindown \leftarrow min(mindown, cost)$
6. **return** $min(minright, mindown)$
   {Total $\mathbf{O}(2^n)$}

해결: 분할통치법 (역방향)

- **부문제 정의:** $m(i,j)$를 셀 $[i,j]$에서 출발하여 셀 $[n-1, n-1]$에 도달할 때까지 뺏길 수 있는 최소 금화량이라 하면 다음이 성립

  $k(j+1 \le k \le n-1)$에 대해, 최소의
  $A[i,j] + m(i,k)$가 *minright*이고,

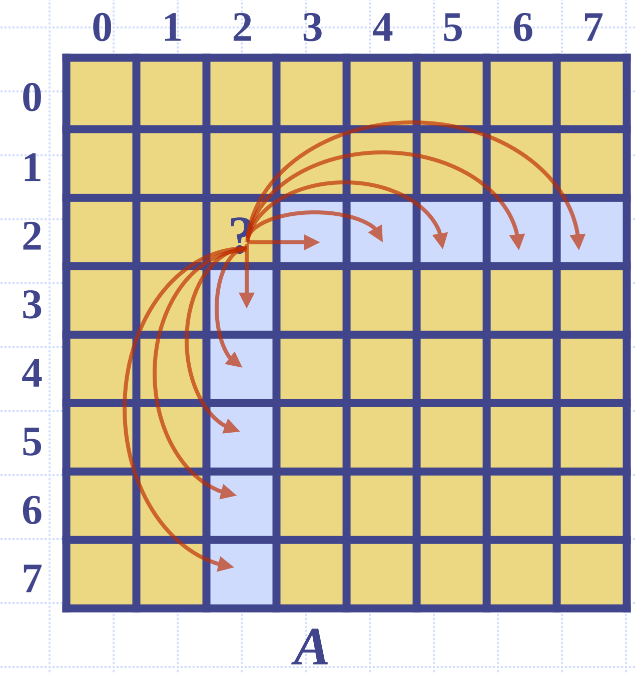
  $k(i+1 \le k \le n-1)$에 대해, 최소의
  $A[i,j] + m(k,j)$가 *mindown*이면,

  $m(i,j)$는 *minright*와 *mindown* 중 최소값

- **베이스 케이스**

  $m(n-1, n-1) = A[n-1, n-1]$

- 2^n개의 **재귀호출**이 일어나므로, 전체적으로 $\mathbf{O}(2^n)$ 시간 소요!



해결: 분할통치법 (역방향)

Alg *minGold*(A, n)
{divide and conquer, backward ver.}
input array A of $n \times n$ gold coins
output minimum possible gold coins moving from [0, 0] to [n – 1, n – 1]

1. **return** $m(0, 0)$

Alg $m(i, j)$
input index i, j
output minimum possible gold coins moving from [i, j] to [n – 1, n – 1]

1. **if** $((i = n - 1)$ & $(j = n - 1))$
   return $A[n - 1, n - 1]$
2. $minright \leftarrow \infty$
3. **for** $k \leftarrow j + 1$ **to** $n - 1$ {move right}
   $cost \leftarrow A[i, j] + m(i, k)$
   $minright \leftarrow min(minright, cost)$
4. $mindown \leftarrow \infty$
5. **for** $k \leftarrow i + 1$ **to** $n - 1$ {move down}
   $cost \leftarrow A[i, j] + m(k, j)$
   $mindown \leftarrow min(mindown, cost)$
6. **return** $min(minright, mindown)$
   {Total $\mathbf{O}(2^n)$}

해결: 동적프로그래밍 (정방향)

- **부문제 정의:** $m[i,j]$를 셀 [0, 0]에서 출발하여 셀 $[i,j]$에 도달할 때까지 뺏길 수 있는 최소 금화량이라 하면 다음이 성립

  $k(j-1 \geq k \geq 0)$에 대해, 최소의
  $m[i,k] + A[i,j]$가 $minright$이고,

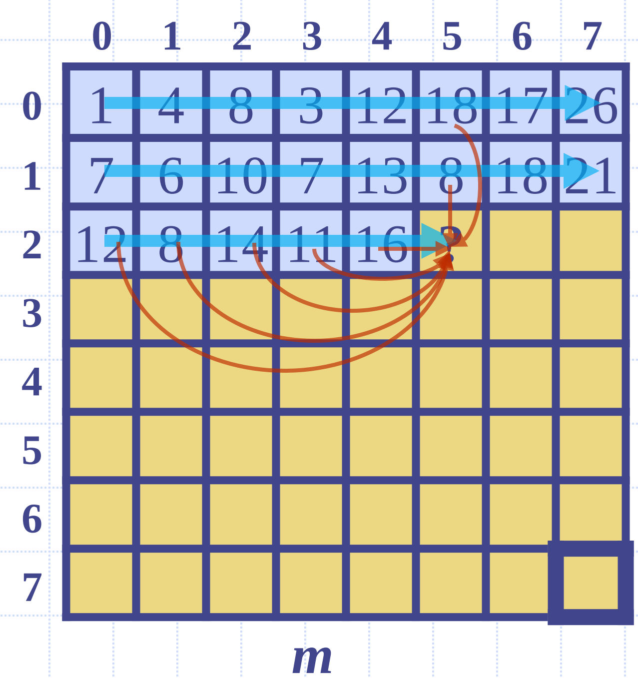
  $k(i-1 \geq k \geq 0)$에 대해, 최소의
  $m[k,j] + A[i,j]$가 $mindown$이면,

  $m[i,j]$는 $minright$와 $mindown$ 중 최소값

- **초기화**

  $m[0, 0] = A[0, 0]$

- n^2개의 부문제가 존재하고 각각의 해결에 $\mathbf{O}(n)$ 시간을 소요하므로, 전체적으로 $\mathbf{O}(n^2)$ 공간, $\mathbf{O}(n^3)$ 시간 소요



해결: 동적프로그래밍 (정방향)

Alg *minGold*(A, n)
{dynamic programming, forward ver.}
input array A of $n \times n$ gold coins
output minimum possible gold coins moving from [0, 0] to [n – 1, n – 1]

1. $m[0, 0] \leftarrow A[0, 0]$

2. **for** $i \leftarrow 0$ **to** $n - 1$
    for $j \leftarrow 0$ **to** $n - 1$
        if ($i = j = 0$)
            continue
        $minright \leftarrow \infty$
        for $k \leftarrow j - 1$ **downto** 0   {move right}
            $cost \leftarrow m[i, k] + A[i, j]$
            $minright \leftarrow min(minright, cost)$
        $mindown \leftarrow \infty$
        for $k \leftarrow i - 1$ **downto** 0   {move down}
            $cost \leftarrow m[k, j] + A[i, j]$
            $mindown \leftarrow min(mindown, cost)$
        $m[i, j] \leftarrow min(minright, mindown)$
3. **return** $m[n - 1, n - 1]$

{Total $O(n^3)$}

- $m[i, j]$ = 셀 [0, 0]에서 출발하여 셀 [i, j]에 도달할 때까지 뺏길 수 있는 최소 금화량
- $\therefore$ $m[n - 1, n - 1]$ = 셀 [0, 0]에서 출발하여 셀 [n – 1, n – 1]에 도달할 때까지 뺏길 수 있는 최소 금화량

해결: 동적프로그래밍 (역방향)

- **부문제 정의:** $m[i,j]$를 셀 $[i,j]$에서 출발하여 셀 $[n-1, n-1]$에 도달할 때까지 뺏길 수 있는 최소 금화량이라 하면 다음이 성립

  $k(j+1 \le k \le n-1)$에 대해, 최소의
  $A[i,j] + m[i,k]$가 $minright$이고,

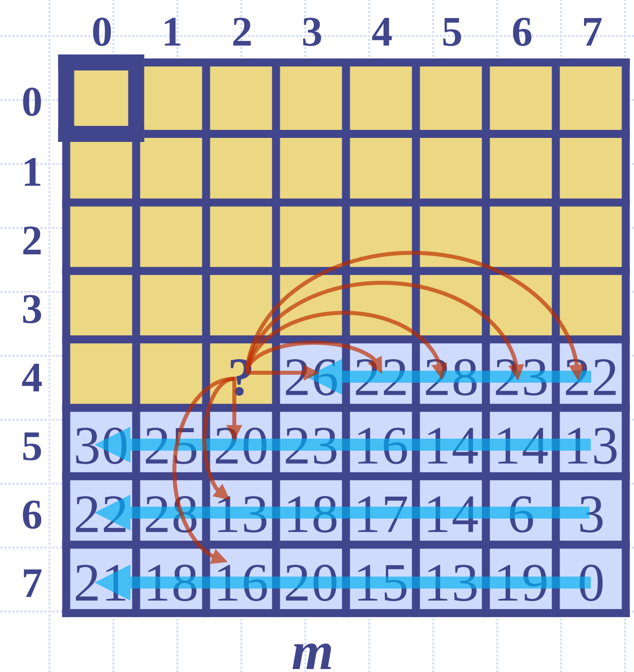
  $k(i+1 \le k \le n-1)$에 대해, 최소의
  $A[i,j] + m[k,j]$가 $mindown$이면,

  $m[i,j]$는 $minright$와 $mindown$ 가운데 최소값

- **초기화**

  $m[n-1, n-1] = A[n-1, n-1]$

- n^2개의 부문제가 존재하고 각각의 해결에 $O(n)$ 시간을 소요하므로, 전체적으로 $O(n^2)$ 공간, $O(n^3)$ 시간 소요

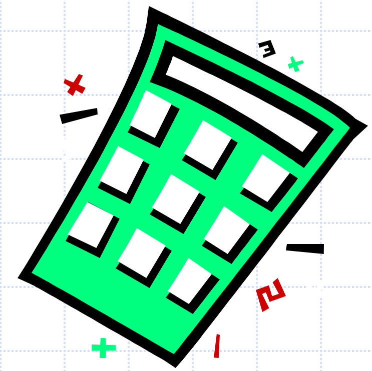


해결: 동적프로그래밍 (역방향)

```
Alg minGold(A, n)                          2. for i ← n – 1 downto 0
      {dynamic programming                        for j ← n – 1 downto 0
      backward ver.}                                  if (i = j = n – 1)
   input array A of n × n gold                            continue
      coins                                           minright ← ∞
   output minimum possible                            for k ← j + 1 to n – 1       {move right}
      gold coins moving from [0,                          cost ← A[i, j] + m[i, k]
      0] to [n – 1, n – 1]                                minright ← min(minright, cost)
                                                      mindown ← ∞
1. m[n – 1, n – 1] ← A[n – 1, n – 1]                  for k ← i + 1 to n – 1       {move down}
                                                          cost ← A[i, j] + m[k, j]
                                                          mindown ← min(mindown, cost)
                                                      m[i, j] ← min(minright, mindown)
                                           3. return m[0, 0]
                                                                          {Total O(n^3)}
```

- $m[i, j]$ = 셀 $[i, j]$에서 출발하여 셀 $[n-1, n-1]$에 도달할 때까지 뺏길 수 있는 최소 금화량
- $\therefore\ m[0, 0]$ = 셀 $[0, 0]$에서 출발하여 셀 $[n-1, n-1]$에 도달할 때까지 뺏길 수 있는 최소 금화량

응용문제: 부배열의 최대 구간합



- A는 크기 n의 실수 배열이다
- **부배열**의 **구간합** $\Sigma A[i{:}j]$가 **최대**가 되는 구간 $i{:}j$ $(i \le j)$와 해당 구간합을 찾기 위한 가장 빠른 알고리즘을 작성하라
- **예:** 아래 배열 A에서,
  - $i = 2, j = 6$
  - 최대합 $= 187$

	0	1	2	3	4	5	6	7	8	9
A	31	–41	59	26	–53	58	97	–93	–23	84

해결: 단순직선적

- **단순직선적**인 해결책
- 모든 가능한 $i{:}j$ 구간을 검사
  - $\mathbf{O}(n^2)$개의 구간 존재
  - 각 구간에서 구간합 $\Sigma A[i{:}j]$을 구하는데 $\mathbf{O}(n)$ 시간 소요
  - $\mathbf{O}(1)$ 공간, $\mathbf{O}(n^3)$ 시간

```
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)}
```

해결: 누적합을 사용

- $\Sigma A[i{:}j] = \Sigma A[i{:}j-1] + A[j]$ 에 착안한 해결책
- 구간합 $\Sigma A[i{:}j]$ 계산부를 제거하고, 이를 **누적합**(running sum)으로 대체
  - 각 구간에서의 작업량이 $\mathbf{O}(1)$ 시간으로 감소
  - $\mathbf{O}(1)$ 공간, $\mathbf{O}(n^2)$ 시간

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)}

해결: 초기구간합을 사용

- 초기구간합 $s[i] = \Sigma A[0{:}i]$ 를 사용,

  $\Sigma A[i{:}j] = \Sigma A[0{:}j] - \Sigma A[0{:}i-1]$

  $= s[j] - s[i-1]$

  에 착안한 해결책
- $\Sigma A[0{:}i]$를 $s[i]$에 미리 저장
  - 계산편의 상 $s[-1] = 0$ 으로 정의
- maxSubarray v.1의 내부 반복문을 구간합 $\Sigma A[i{:}j]$이 $O(1)$ 시간에 계산되도록 수정
  - $O(n)$ 공간, $O(n^2)$ 시간

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 $O(n^2)$}

해결: 초기구간합을 사용 (conti.)

- **예:** 아래 배열 A에서,
  - $maxi = 2$, $maxj = 6$
  - 최대합 = 187

			maxi				*maxj*			
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
A	31	–41	59	26	–53	58	97	–93	–23	84

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
s	31	**–10**	49	75	22	80	**177**	84	61	145

$$maxSum = s[6] - s[1] = 177 - (-10) = 187$$

해결: 동적프로그래밍을 사용

- **동적프로그래밍**에 기초한 해결책
  - 셀 0(원점)에서 출발하여 셀 $n-1$(목표점) 방향으로 진행하면서 해를 구함
- $s[i] = \Sigma A[?:i]$를 사용, $s[i] = max(s[i-1] + A[i], A[i])$ 에 착안한 해결책
- $s[i-1]$이 **음수**면,
  - (잠정적) 최대 부배열 구간의 시작 위치 k를 i로 설정
- 배열 위치 i에서 새로운 최대합 $s[i]$를 찾으면,
  - 최대합을 $s[i]$로, 최대 부배열 구간을 $k:i$로 각각 갱신
- 배열 A의 각 원소를 한번만 검사
  - $\mathbf{O}(n)$ 공간, $\mathbf{O}(n)$ 시간
  - $\mathbf{O}(1)$ 공간으로 개선 가능

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

1. $s[-1] \leftarrow 0$
2. $maxSum, maxi, k \leftarrow -\infty, 0, 0$
3. $i \leftarrow 0$
4. **while** ($i < n$) {$\mathbf{O}(n)$}
   - $s[i] \leftarrow max(s[i-1] + A[i], A[i])$
   - **if** ($s[i-1] < 0$)
     - $k \leftarrow i$
   - **if** ($maxSum < s[i]$)
     - $maxSum \leftarrow s[i]$
     - $maxi, maxj \leftarrow k, i$
   - $i \leftarrow i + 1$
5. **return** $maxSum, maxi, maxj$

{Total $\mathbf{O}(n)$}

해결: 동적프로그래밍을 사용 (conti.)

- **예**: 아래 배열 A에서,
  - $\boldsymbol{maxi} = 2$, $\boldsymbol{maxj} = 6$
  - 최대합 = 187

	0	1	2	3	4	5	6	7	8	9
A	31	–41	59	26	–53	58	97	–93	–23	84

	0	1	2	3	4	5	6	7	8	9
s	31	–10	**59**	**85**	32	**90**	**187**	94	71	155

$maxi \longrightarrow maxi$

$maxj \longrightarrow maxj$