5장

그리디 알고리즘

GREEDY ALGORITHM

지금 이 순간 가장 좋아 보이는 것을 고른다

이번 장에서 배울 내용

5.1 그리디의 기본

5.2 거스름돈 문제

5.3 활동 선택 문제

5.4 배낭 문제

5.5 그리디가 통하는 조건

5.6 크루스칼 알고리즘

5.7 TSP와 그리디

5.8 다익스트라 알고리즘

5.9 코딩 테스트 예제

5.1

그리디의 기본

"눈앞의 마시멜로 하나를 지금 먹을 것인가,
15분을 기다려 두 개를 먹을 것인가?"

그리디: 지금 가장 좋아 보이는 것을 고른다

그리디(greedy)는 '탐욕스러운'이라는 뜻입니다.
매 단계 지금 기준으로
가장 좋아 보이는 것을 고릅니다.

후보를 다 알아도 지금 기준 하나로

앞을 못 봐서 틀리는 것이 아닙니다.
정해 둔 기준으로만 고르기 때문입니다.

고른 것은 확정, 되돌리지 않음

앞의 선택을 다시 검토하는 단계가 없습니다.

완성 해를 모두 비교하지 않음

모든 해를 만들어 비교하는 것은 완전탐색입니다.

**되돌아보지 않아서 빠르고,
그래서 틀릴 수도 있습니다.**



손님은 '지금 맛있어 보이는 것' 기준 하나로 접시를 채웠고, 내려놓지 않아 스테이크 자리가 없습니다.

지역 최적해와 전역 최적해

안개 낀 산에서 가장 높은 봉우리를 찾습니다.
가장 단순한 전략은 지금보다 높은 쪽으로
한 걸음씩 오르는 것입니다.

지역 최적해 (local optimum)

주변 어디보다 높은 곳.
작은 언덕 꼭대기일 수 있습니다.

전역 최적해 (global optimum)

산 전체에서 가장 높은 곳.
우리가 원하는 답입니다.

매 걸음의 최선이 모여도
전체의 최선이라는 보장은 없습니다.



마시멜로 하나를 지금 먹는 것도 지금만 보면 최선입니다.
기다려서 두 개를 받는 쪽이 전체로는 더 좋습니다.

그림의 '국소적 최적해(Local Maximum)'는 지역 최적해(local optimum)와 같은 말입니다.

그리디 알고리즘의 세 단계

해답이 완성될 때까지 세 단계를 되풀이합니다.

① 선택

selection
남은 후보 중 지금 기준으로
가장 좋은 것 하나를 고릅니다.

② 적정성 검사

feasibility
그 선택을 넣어도 제약을
어기지 않는지 확인합니다.

③ 해답 검사

solution
모은 것이 완전한 해답인지
확인합니다. 아니면 ①로.

greedy.pseudo

```
GREEDY(C):                                          // C: 후보 집합
    S <- 빈 집합                                     // 지금까지 확정한 선택
    while C가 비어 있지 않고 S가 해답이 아니면:        // 해답 검사
        C에서 지금 가장 좋은 후보 x를 꺼낸다          // 선택
        if S + {x}가 제약을 지키면:                   // 적정성 검사
            S <- S + {x}                            // 확정, 되돌리지 않음
    if S가 해답이면 return S else return "해 없음"
```

한 번 확정한 선택은 다시 검토하지 않습니다.

세 단계를 이 장의 예제에 대응하면

문제마다 '지금 가장 좋은 것'의 기준과 지켜야 할 제약이 다릅니다.
되풀이하는 틀은 같습니다.

예제	① 선택	② 적정성 검사	③ 해답 검사
거스름돈 (5.2)	남은 돈 이하의 가장 큰 동전	남은 돈을 넘지 않는가	남은 돈이 0원인가
회의 배정 (5.3)	가장 먼저 끝나는 회의	시작 ≥ 마지막 종료 시각	더 볼 회의가 없는가
분할 배낭 (5.4)	무게당 가치가 가장 높은 보물	남은 용량 안에 들어가는가	배낭이 가득 찼거나 남은 보물이 없는가

어느 예제든 한 번 고른 것은 끝까지 그대로 둡니다.

각 절에서 이 표의 칸을 하나씩 다시 확인합니다.

이 장의 지도: 통하는 문제와 통하지 않는 문제

모두 '지금 가장 좋은 것 고르기'이지만,
최적해를 보장하는 문제와 그렇지 않은 문제가 있습니다.

절	문제	매 단계의 선택	최적해
5.2	거스름돈 (500·100·50·10원)	가장 큰 동전	**보장**
5.2	거스름돈 (400원 동전 추가)	가장 큰 동전	**보장 못 함**
5.3	활동 선택 (회의 배정)	가장 먼저 끝나는 회의	**보장**
5.4	분할 배낭 (쪼갤 수 있음)	무게당 가치가 가장 높은 보물	**보장**
5.4	0/1 배낭 (쪼갤 수 없음)	무게당 가치가 가장 높은 보물	**보장 못 함**
5.6	최소 신장 트리 (크루스칼)	사이클을 만들지 않는 가장 싼 간선	**보장**
5.7	외판원 순회 (TSP)	가장 가까운 미방문 도시	**보장 못 함**
5.8	최단 경로 (다익스트라)	시작점에서 가장 가까운 미확정 정점	**보장 (간선 비용 ≥ 0)**

왜 어떤 문제에서는 통하고 어떤 문제에서는 통하지 않는지는 5.5절에서 두 조건으로 정리합니다.

확인: 그리디의 기본

① 그리디 알고리즘의 진행 방식을 한 문장으로 말해 보세요.

② "매 단계에서 남은 모든 완성 해를 비교한 뒤 하나를 확정한다."
이것은 그리디입니까?

③ 매 단계에서 지역적으로 가장 좋은 것을 고르면
전역 최적해가 항상 보장됩니까?

④ 거스름돈 문제에서 세 단계(선택, 적정성 검사, 해답 검사)는
각각 무엇입니까?

5.2

거스름돈 문제

500원, 100원, 50원, 10원짜리 동전으로
1,260원을 가장 적은 개수로 거슬러 주려면?

손님에게 1,260원을 거슬러 줍니다

동전은 500원, 100원, 50원, 10원
네 종류가 충분히 있습니다.

목표: 동전 개수를 최소로

그리디의 기준

남은 돈 이하의 가장 큰 동전부터 고릅니다.

한 번에 몫만큼

그 동전으로 낼 수 있는 만큼 한꺼번에 냅니다.

나머지로 반복

남은 돈으로 다음 큰 동전을 봅니다.



만화의 점원도 가장 큰 500원부터 집습니다.

몫과 나머지로 한 번에 셉니다

500원을 하나씩 빼지 않습니다.
나눗셈 한 번으로 쓸 개수와 남은 돈을 함께 구합니다.

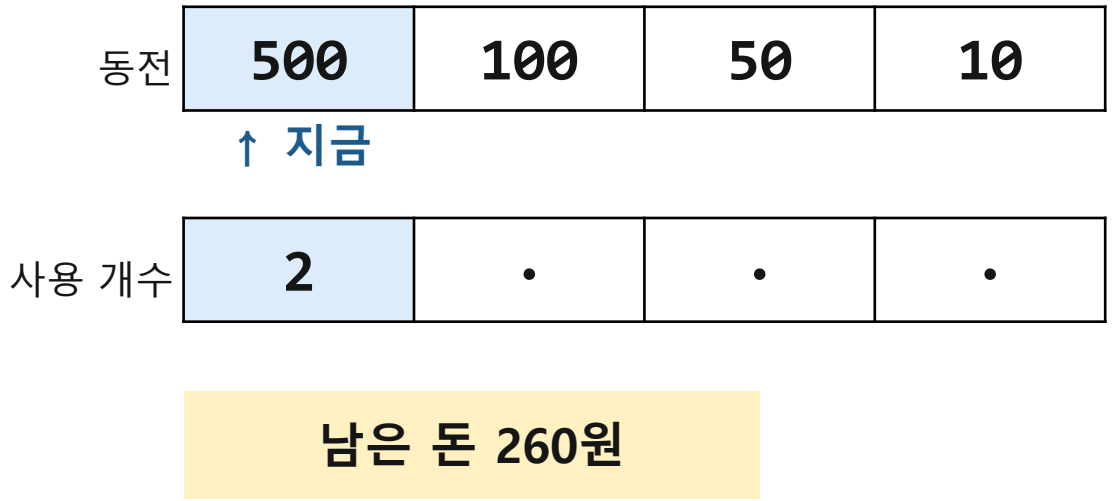
식 (Python)	뜻	1,260원에서
`1260 // 500`	몫: 500원을 몇 개 쓰는가	**2**
`1260 % 500`	나머지: 다음 동전으로 넘기는 돈	**260**

1260 ÷ 500 = 2.52처럼 소수로 보지 않습니다. 500원 2개를 쓰고 260원이 남습니다.

그리디의 세 단계	거스름돈에서
① 선택	남은 돈 이하의 가장 큰 동전
② 적정성 검사	몫만큼 써도 남은 돈을 넘지 않는다
③ 해답 검사	남은 돈이 0원이면 끝

① 500원 · 1,260원에서 2개

1,260원 이하의 가장 큰 동전은 500원입니다.
1260 // 500 = 2, 1260 % 500 = 260



단계	이번 판단
① 선택	500원
② 적정성	2개 = 1,000원 ≤ 1,260원
③ 해답 검사	260원 남음 → 계속

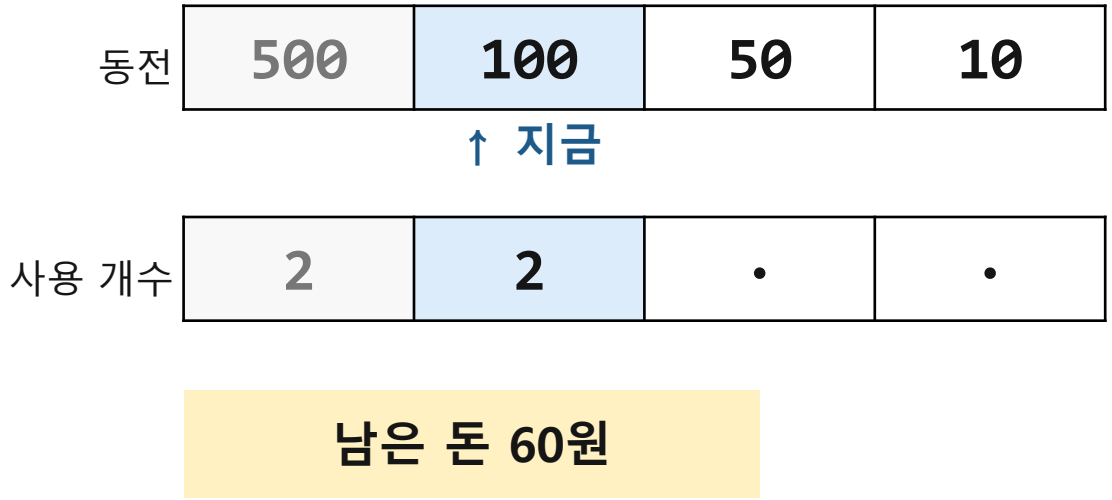
지금까지 확정한 동전

500원 2개

파랑: 지금 보는 동전　　회색: 이미 처리한 동전

② 100원 · 260원에서 2개

500원은 260원보다 커서 더 쓸 수 없습니다.
260 // 100 = 2, 260 % 100 = 60



단계	이번 판단
① 선택	100원
② 적정성	2개 = 200원 ≤ 260원
③ 해답 검사	60원 남음 → 계속

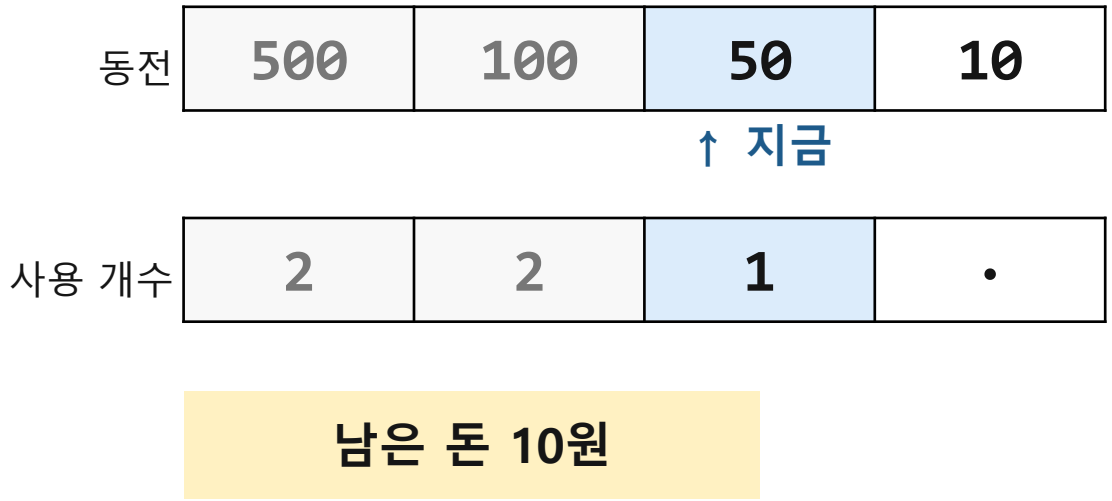
지금까지 확정한 동전

500원 2개 + 100원 2개

파랑: 지금 보는 동전　　회색: 이미 처리한 동전

③ 50원 · 60원에서 1개

100원은 60원보다 큽니다. 다음 50원입니다.
60 // 50 = 1, 60 % 50 = 10



단계	이번 판단
① 선택	50원
② 적정성	1개 = 50원 ≤ 60원
③ 해답 검사	10원 남음 → 계속

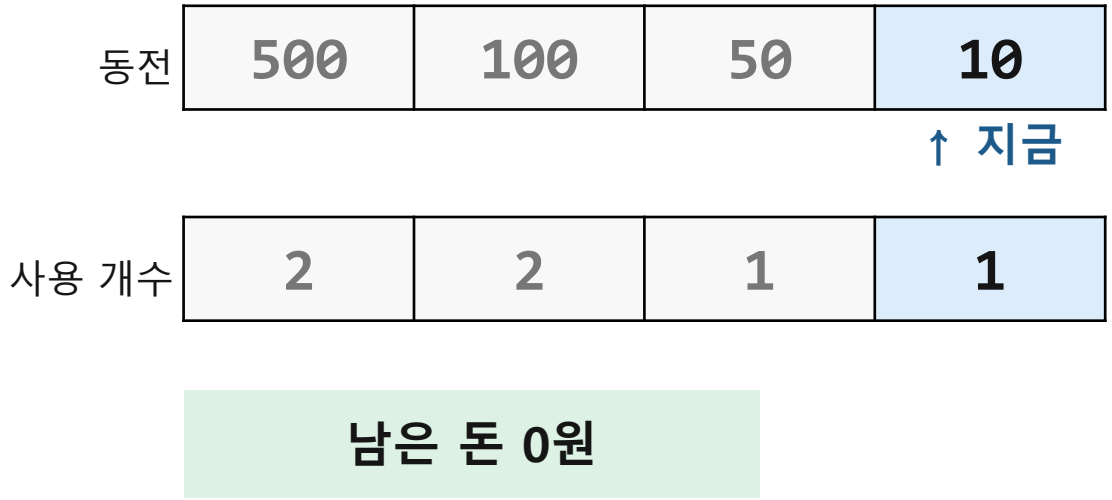
지금까지 확정한 동전

500원 2개 + 100원 2개 + 50원 1개

파랑: 지금 보는 동전　　회색: 이미 처리한 동전

④ 10원 · 남은 돈 0원, 모두 6개

마지막 10원입니다. 10 // 10 = 1, 10 % 10 = 0
남은 돈이 0원이 되어 해답이 완성됩니다.



단계	이번 판단
① 선택	10원
② 적정성	1개 = 10원 ≤ 10원
③ 해답 검사	0원 → 완성

지금까지 확정한 동전

2 + 2 + 1 + 1 = 6개

이 6개가 최소인 이유는
뒤의 배수 조건에서 봅니다.

파랑: 지금 보는 동전 회색: 이미 처리한 동전

Python: 큰 단위부터 몫으로

change.py

```
n = 1260            # 거슬러 줘야 할 돈
count = 0

# 큰 단위부터 확인 — 반드시 내림차순이어야 한다
coin_types = [500, 100, 50, 10]

print(f"거스름돈: {n}원 시작")
print("-" * 30)
for coin in coin_types:
    num_coins = n // coin    # 몫 = 이 동전을 몇 개 쓰는가
    count += num_coins
    n %= coin                # 나머지 = 남은 돈
    print(f"{coin}원짜리: {num_coins}개 사용 (남은 돈: {n}원)")
print("-" * 30)
print(f"최소 동전 개수: {count}개")
```

coin_types를 [10, 50, 100, 500]으로 적으면 10원 126개가 나옵니다. 큰 단위부터 적어야 합니다.

C ①: 변수와 동전 배열

change.c

```
#include <stdio.h>

int main(void) {
    int n = 1260;           /* 거슬러 줘야 할 돈 */
    int count = 0;

    /* 큰 단위부터 확인 – 반드시 내림차순이어야 한다 */
    int coin_types[] = {500, 100, 50, 10};
```

C ②: 큰 단위부터 몫으로

change.c

```
    printf("거스름돈: %d원 시작\n", n);
    printf("-----------------------------\n");
    for (int i = 0; i < 4; i++) {
        int coin = coin_types[i];
        int num_coins = n / coin;   /* 몫 = 이 동전을 몇 개 쓰는가 */
        count += num_coins;
        n %= coin;                  /* 나머지 = 남은 돈 */
        printf("%d원짜리: %d개 사용 (남은 돈: %d원)\n",
               coin, num_coins, n);
    }
    printf("-----------------------------\n");
    printf("최소 동전 개수: %d개\n", count);
    return 0;
}
```

실행 결과

Python과 C의 출력이 같습니다.

OUTPUT

```
거스름돈: 1260원 시작
------------------------------
500원짜리: 2개 사용 (남은 돈: 260원)
100원짜리: 2개 사용 (남은 돈: 60원)
50원짜리: 1개 사용 (남은 돈: 10원)
10원짜리: 1개 사용 (남은 돈: 0원)
------------------------------
최소 동전 개수: 6개
```

남은 돈은 260원 → 60원 → 10원 → 0원으로 줄고, 동전은 2 + 2 + 1 + 1 = 6개입니다.

복잡도: 금액이 아니라 동전 종류 수

K = 동전 종류의 수. 지금은 500, 100, 50, 10원이라 K = 4입니다.

거슬러 줄 돈	몫과 나머지: 반복 횟수	하나씩 빼기: 빼는 횟수
1,260원	4번	6번
1,260,000원	**4번**	**2,520번**

반복 한 번: 몫과 나머지 한 번씩 → O(1)

금액이 커도 나눗셈 한 번이면 개수가 나옵니다.

전체: K × O(1) = O(K)

목록이 이미 큰 단위부터 적혀 있을 때입니다. 정렬부터 해야 하면 O(K log K)가 더해집니다.

하나씩 빼는 방법은 반복 횟수가 금액에 비례해서 늘어납니다.

동전 종류가 고정이면 K는 상수이므로 O(1)이라고도 씁니다.

왜 큰 동전부터가 최적인가: 배수 조건

한국 동전은 큰 단위가 바로 아래 단위의 배수입니다.

큰 동전 1개	같은 금액의 작은 동전	개수
500원	100원 5개	1개 ↔ 5개
100원	50원 2개	1개 ↔ 2개
50원	10원 5개	1개 ↔ 5개

작은 동전이 쌓이면 큰 동전 하나로 바꿀 수 있습니다

100원 5개를 500원 1개로 바꾸면 금액은 같고 개수는 5개에서 1개로 줄어듭니다.

그래서 큰 동전을 덜 쓰면 개수만 늘어납니다

큰 동전을 낼 수 있는 만큼 쓰는 그리디의 선택이 손해가 되지 않습니다.

배수 조건이면 그리디가 최적입니다(충분조건).

배수 조건이 깨지면 이 보장이 사라집니다.
늘 틀린다는 뜻이 아니라 틀리는 입력이 생길 수 있다는 뜻입니다.

400원 동전이 생기면: 800원 거슬러 주기

편의점에 400원 동전이 새로 생겼다고 합시다.
동전은 500, 400, 100, 50, 10원입니다.

800원을 최소 개수로?

그리디

남은 돈 이하의 가장 큰 500원부터 집습니다.

다른 방법

500원을 쓰지 않고 400원 두 개를 씁니다.

동전 종류만 늘었고 그리디의 규칙은 그대로입니다.



만화: 그리디는 동전 4개, 400원 두 개를 쓰면 2개입니다.

그리디: 500 + 100 + 100 + 100 = 4개

800원 이하의 가장 큰 500원 1개 → 300원 남음.
300원에 400원은 들어가지 않아 100원 3개.

동전	500	400	100	50	10
그리디	1	0	3	0	0

남은 돈 0원 · 동전 4개

동전	몫	남은 돈
500원	800 // 500 = 1	300원
400원	300 // 400 = 0	300원
100원	300 // 100 = 3	0원

500원을 고르는 순간 남은 돈이 300원이 되어 400원을 쓸 수 없게 됩니다.

파랑: 그리디가 쓴 동전 빨강 글자: 남은 돈보다 커서 못 쓴 동전

최적: 400 + 400 = 2개

500원을 쓰지 않으면 400원 2개로 끝납니다.
2개 < 4개, 그리디의 답은 최적이 아닙니다.

동전	500	400	100	50	10
그리디	1	0	3	0	0
최적	0	2	0	0	0

그리디 4개 vs 최적 2개

왜 그리디가 틀렸나

400원 두 개 안에는
합이 500원인 묶음이 없어
500원으로 바꿔 넣을 수
없습니다.

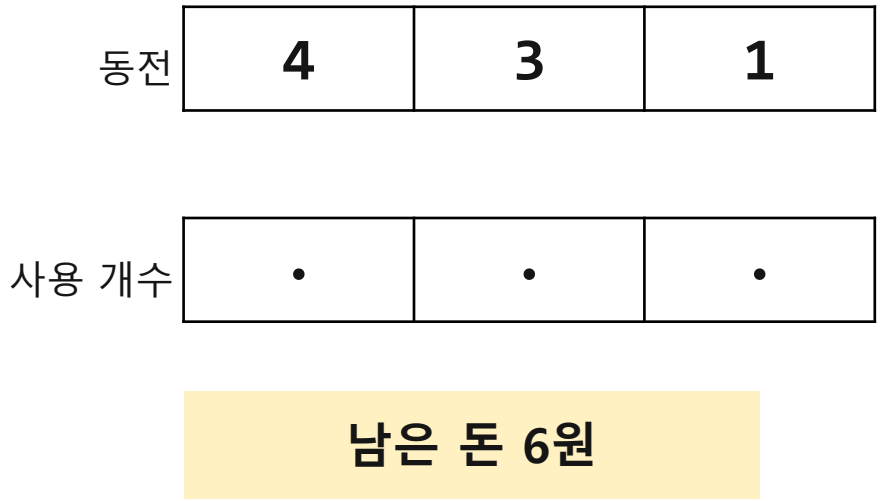
500은 400의 배수가 아닙니다.

배수 조건이 깨지자
큰 동전을 최대한 쓰는 선택이
손해가 되었습니다.

파랑: 그리디가 쓴 동전 초록: 최적해가 쓴 동전

연습: 동전 1, 3, 4원으로 6원 · 시작

주장: "남은 돈 이하의 가장 큰 동전을 하나씩 고르면 어떤 화폐 체계에서도 개수가 최소다."



문제 조건

동전: 1원, 3원, 4원
(각각 무제한)

목표: 정확히 6원을
가장 적은 개수로

그리디: 남은 돈 이하의
가장 큰 동전 하나를 고르고
0원이 될 때까지 반복

이 주장을 반박해 봅시다.

파랑: 지금 고른 동전　　회색: 남은 돈보다 커서 고를 수 없는 동전

① 4원 · 6원에서 4원 하나

6원 이하의 가장 큰 동전은 4원입니다.
4원 하나를 확정합니다. 남은 돈 6 − 4 = 2원.



단계	이번 판단
① 선택	4원
② 적정성	4원 ≤ 6원
③ 해답 검사	2원 남음 → 계속

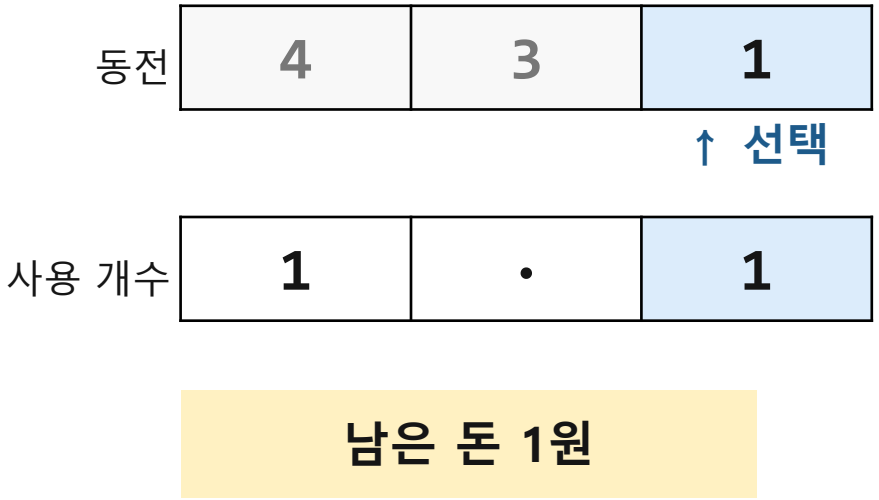
지금까지 고른 동전

4원 (1개)

파랑: 지금 고른 동전　　회색: 남은 돈보다 커서 고를 수 없는 동전

② 1원 · 2원에서 1원 하나

4원과 3원은 남은 2원보다 커서 고를 수 없습니다.
1원을 고릅니다. 남은 돈 2 − 1 = 1원.



단계	이번 판단
① 선택	1원
② 적정성	1원 ≤ 2원
③ 해답 검사	1원 남음 → 계속

지금까지 고른 동전

4원 + 1원 (2개)

파랑: 지금 고른 동전　　회색: 남은 돈보다 커서 고를 수 없는 동전

③ 1원 · 남은 돈 0원, 모두 3개

남은 1원 이하의 동전은 1원뿐입니다. 남은 돈 0원.
그리디의 답은 4 + 1 + 1 = 3개입니다.



단계	이번 판단
① 선택	1원
② 적정성	1원 ≤ 1원
③ 해답 검사	0원 → 완성

지금까지 고른 동전

4 + 1 + 1 = 3개

파랑: 지금 고른 동전 회색: 남은 돈보다 커서 고를 수 없는 동전

최적: 3 + 3 = 2개 · 반례 완성

4원을 쓰지 않으면 3원 두 개로 6원이 됩니다.
2개 < 3개, 그리디의 답은 최소가 아닙니다.



그리디: 4 + 1 + 1 = 6원 → 3개

최적: 3 + 3 = 6원 → 2개

**이 입력 하나로
주장이 깨집니다.**

반박에 쓸 것:
그리디의 결과, 더 좋은 해,
두 개수의 비교

파랑: 그리디가 쓴 동전 초록: 최적해가 쓴 동전

'항상 최적'은 반례 하나로 반박합니다

'항상 최적이다'라는 주장은 맞지 않는 입력이 하나만 있어도 거짓입니다.

① 문제의 조건을 지키는 입력

동전 1, 3, 4원과 금액 6원은 문제가 허용하는 입력입니다.
조건을 어긴 입력에서 틀리는 것은 반박이 아닙니다.

② 그리디의 결과보다 더 좋은 해

그리디 3개보다 적은 2개를 보였습니다. 그리디와 개수가 같은 다른 해는 반례가 아닙니다.

③ 하나면 충분

반례를 여러 개 모을 필요가 없습니다. 실행이 느린 입력은 최적성의 반례가 아닙니다.

반대로, 예제 몇 개에서 맞았다고 '항상 최적'이 증명되지는 않습니다.

맞다는 것을 보이는 방법은 5.3절 끝과 5.5절에서 다룹니다.

연습: coin_count · 1원까지 다섯 종류

coin_count(amount): amount원을 거슬러 줄
최소 동전 개수를 돌려줍니다. 큰 동전부터
몫만큼 쓰고 나머지를 다음 동전으로 넘깁니다.

동전	500	100	50	10	1
몫 (개수)	2	2	1	1	3
남은 돈	263	63	13	3	0

coin_count(1263) = 2 + 2 + 1 + 1 + 3 = 9

coin_count.pseudo

```
COIN_COUNT(amount):
  count <- 0
  for coin in [500, 100, 50, 10, 1]:
    count <- count + 몫(amount, coin)
    amount <- 나머지(amount, coin)
  return count
```

몫: Python //, C / 나머지: %

금액과 관계없이 5번 → O(1)

K = 동전 종류 수로 쓰면 O(K)
하나씩 빼면 O(amount)
amount = 0이면 0개

확인: 거스름돈

① 동전 500, 100, 50, 10원으로 2,370원을 거슬러 줍니다.
동전마다 몫과 나머지, 전체 개수는?

② 동전 1, 5, 6원으로 10원을 만듭니다. 그리디의 결과와 더 좋은 해를 쓰고,
'항상 최소'라는 주장을 반박해 보세요.

③ coin_count에 넣는 금액이 1,000배가 되면 for 반복 횟수는 어떻게 됩니까?

④ change.py의 coin_types를 [100, 500, 50, 10]으로 적으면
동전은 몇 개가 나옵니까?

5.3

활동 선택 문제

회의를 가장 많이 넣는 법
회의실은 하나, 겹치지 않게 회의를 최대한 많이 배정하려면?

회의실 하나, 회의는 최대한 많이

회의실은 하나, 요청은 여러 건입니다.
회의마다 시작 시각과 종료 시각이 있습니다.

회의 하나 = 활동(activity)

그래서 활동 선택 문제라고 부릅니다.

한 번에 한 회의만

시간이 겹치는 두 회의는 함께 넣을 수 없고,
회의를 도중에 끊을 수도 없습니다.

목표: 회의의 개수를 최대로

사용 시간의 합이 아니라 넣은 회의의 수입니다.

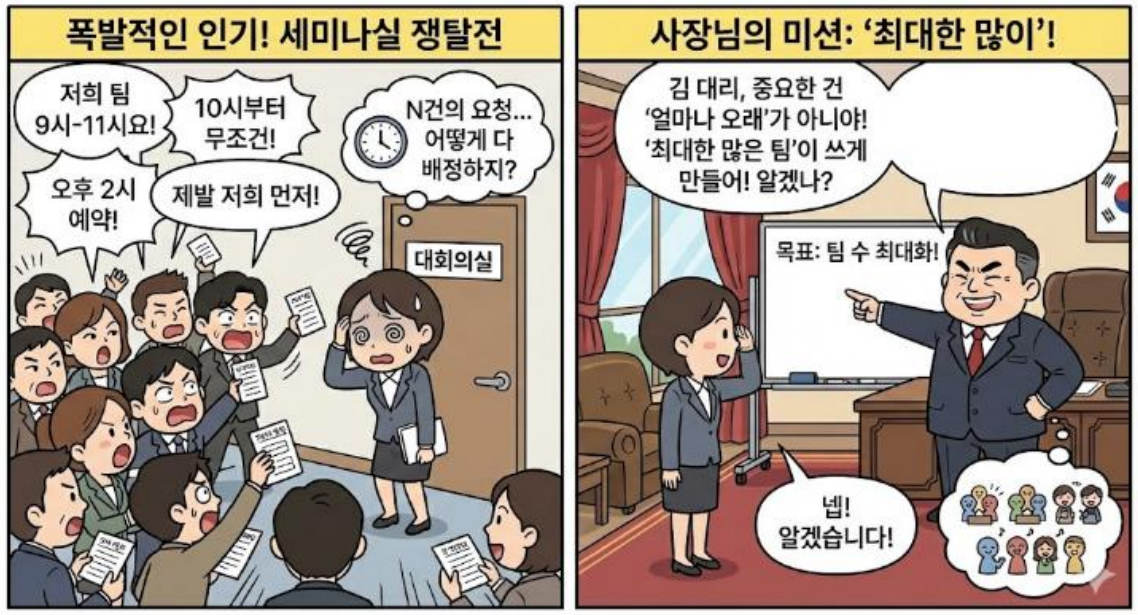
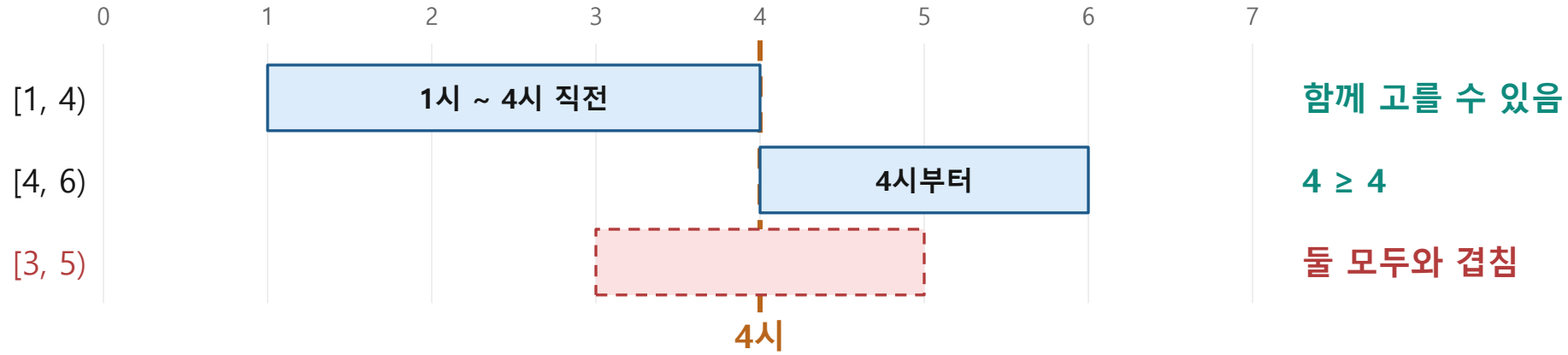


그림 속 '세미나실'은 이 강의의 회의실입니다. 목표는 '오래'가 아니라 '많이'입니다.

[시작, 종료) 표기: 끝나는 시각에 바로 시작

[1, 4)는 1시에 시작해 4시에 끝나는 회의입니다.
[는 1시를 포함하고,)는 4시를 포함하지 않습니다.



규칙: 다음 회의의 시작 ≥ 앞 회의의 종료이면 겹치지 않습니다.

[1, 4)와 [4, 6)은 함께 고를 수 있습니다. 4시에 하나가 끝나고 다른 하나가 시작합니다.
[3, 5)는 4시 전에 시작해 4시 뒤에 끝나므로 [1, 4), [4, 6) 둘 모두와 겹칩니다.

실패 ① 일찍 시작하는 회의부터? · 1개

시작이 빠른 순으로 보며 겹치지 않으면 고릅니다.
[1, 10)을 먼저 고르자 나머지 둘이 모두 겹칩니다.



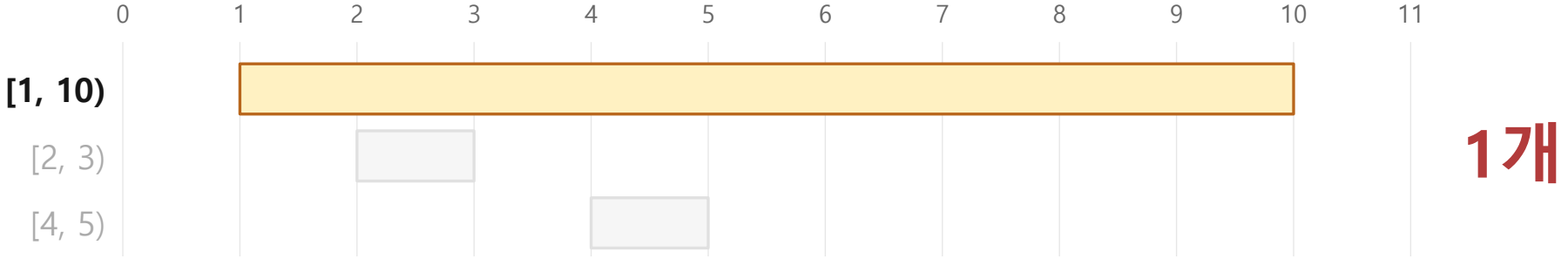
시작 순 그리디: 1개

노랑: 고른 회의 빨강 점선: 겹쳐서 버린 회의 주황 점선: 회의실이 비는 시각

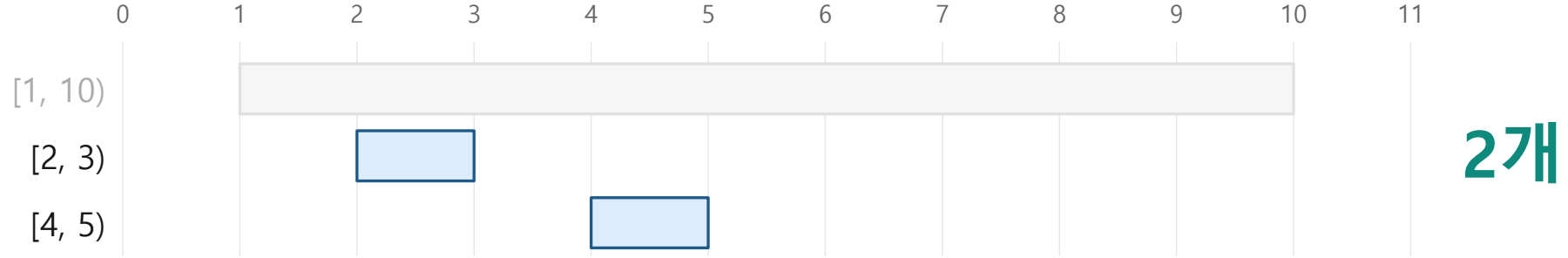
반례 완성: 시작 순 1개 < 최적 2개

[1, 10)을 버리면 [2, 3)과 [4, 5)를 함께 넣을 수 있습니다.
그리디 1개 < 최적 2개, 이 기준은 최적을 보장하지 않습니다.

시작 순 그리디: [1, 10) 하나



최적: [2, 3)과 [4, 5) 두 개

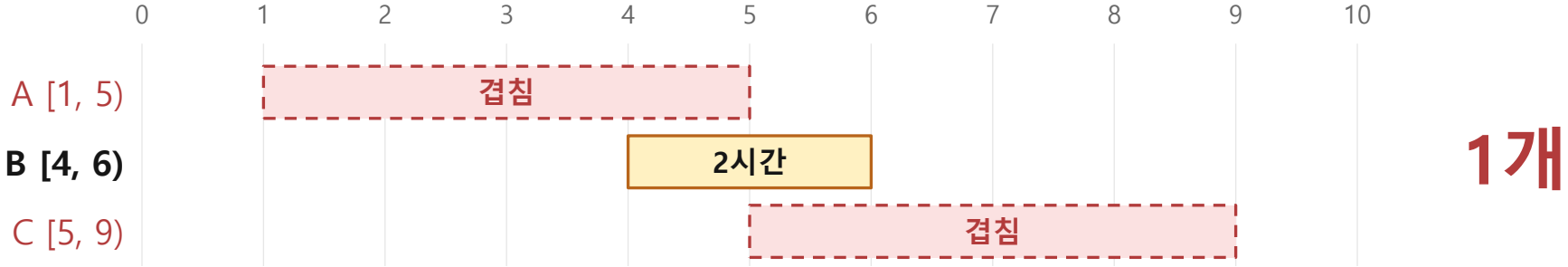


노랑: 그리디가 고른 회의 파랑: 최적해의 회의

실패 ② 짧은 회의부터? · 1개

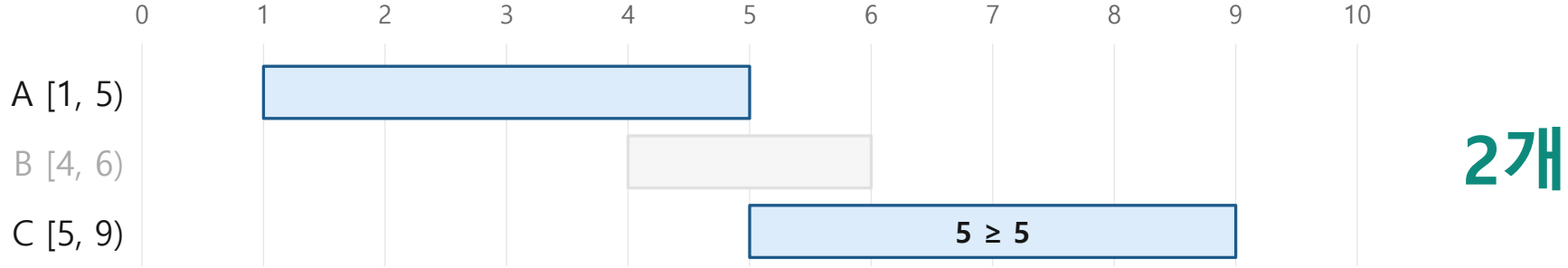
가장 짧은 B [4, 6)을 먼저 고르면 A와 C가 모두 겹칩니다.
A가 끝나는 5시에 C가 시작하므로 A와 C는 함께 고를 수 있습니다.

짧은 순 그리디: B 하나



1개

최적: A와 C 두 개



2개

노랑: 그리디가 고른 회의 파랑: 최적해의 회의 빨강 점선: 겹쳐서 버린 회의

성공 전략: 가장 먼저 끝나는 회의부터

회의실을 가장 빨리 비워 주는 회의부터 고릅니다.
last_end는 마지막으로 고른 회의의 종료 시각입니다.

① 정렬

회의를 종료 시각이
빠른 순으로 늘어놓습니다.

② 고르기

시작 ≥ last_end이면 고르고
last_end를 그 회의의
종료 시각으로 옮깁니다.

③ 반복

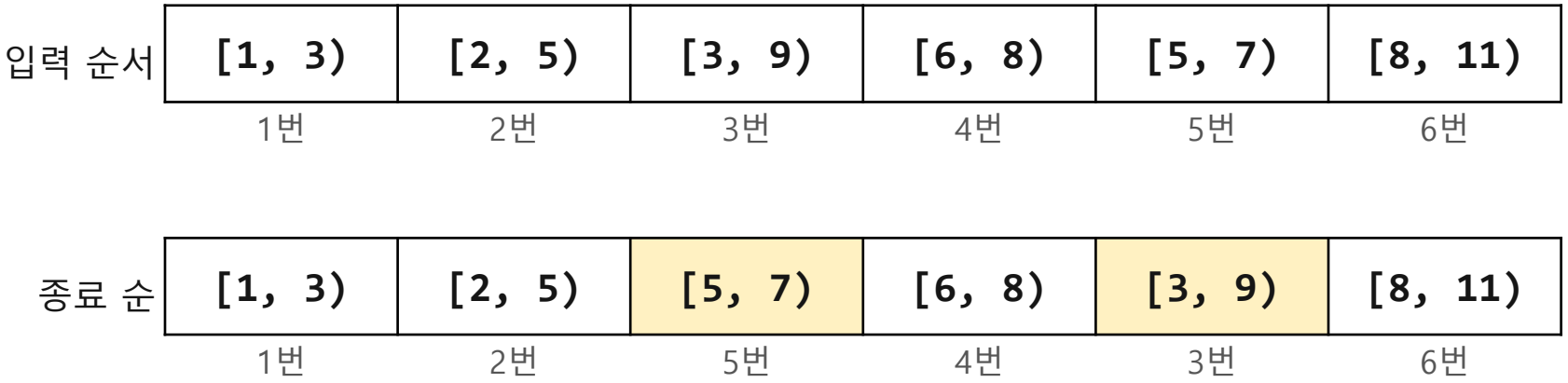
끝까지 봅니다.
조건을 처음 만족하는 회의가
곧 겹치지 않는 회의 중
가장 먼저 끝나는 회의입니다.

앞의 반례	시작 순	짧은 순	종료 순	최적
[1, 10), [2, 3), [4, 5)	**1개**	2개	**2개**	2개
[1, 5), [4, 6), [5, 9)	2개	**1개**	**2개**	2개

두 예에서 맞았다고 해서 증명된 것은 아닙니다. 왜 맞는지는 이 절 끝에서 봅니다.
'안전한 선택'(그렇게 골라도 최적해를 놓치지 않는 선택)의 정확한 뜻은 5.5절에서 정의합니다.

예제: 회의 여섯 건을 종료 시각 순으로

회의 여섯 건을 종료 시각이 빠른 순으로 정렬합니다.
칸 아래의 작은 글자는 입력 순서로 붙인 회의 번호입니다.



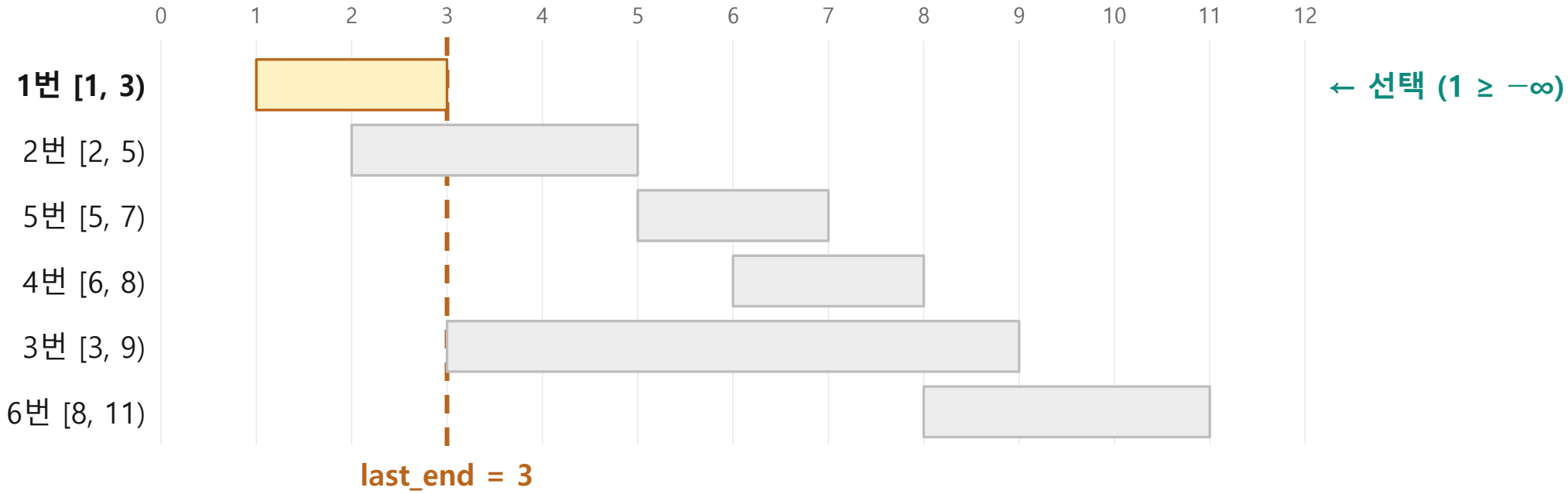
종료 시각: 3 < 5 < 7 < 8 < 9 < 11

last_end = −∞ (아직 고른 회의 없음)

노랑 칸: 입력 순서와 자리가 바뀐 회의

① 1번 [1, 3) · 선택

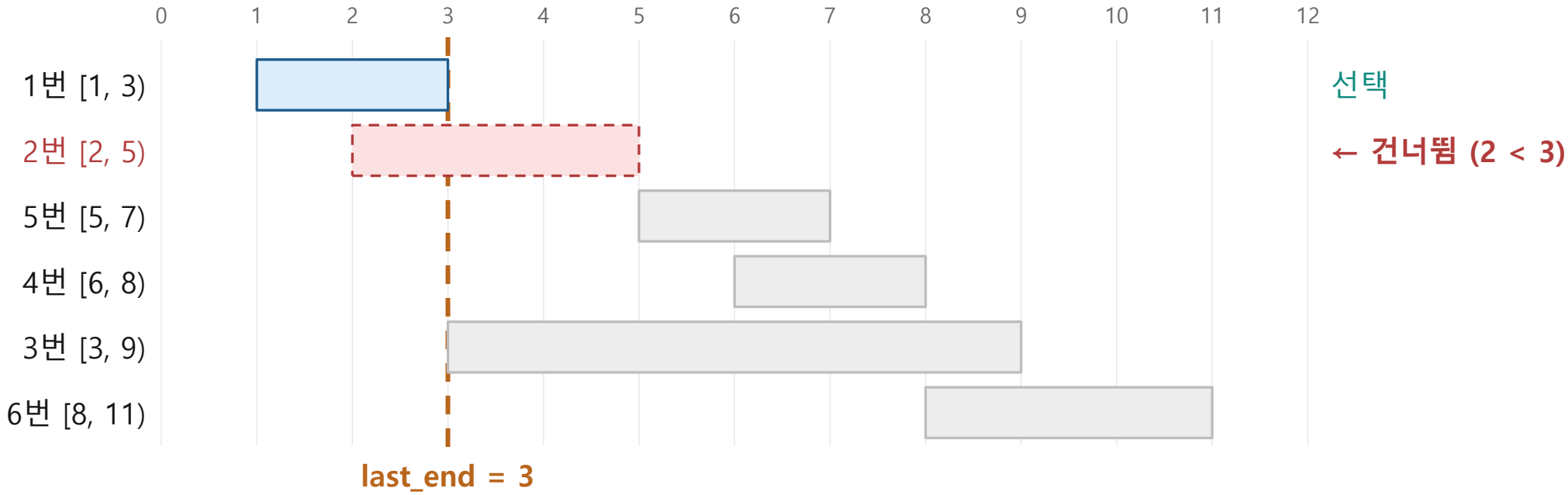
last_end = −∞이므로 1 ≥ −∞가 참입니다. 겹칠 회의가 없습니다.
1번을 고르고 last_end를 3으로 옮깁니다.



파랑: 고른 회의 노랑: 지금 고른 회의 빨강 점선: 건너뜀 회색: 아직 보지 않은 회의 주황 점선: last_end

② 2번 [2, 5) · 건너뜀

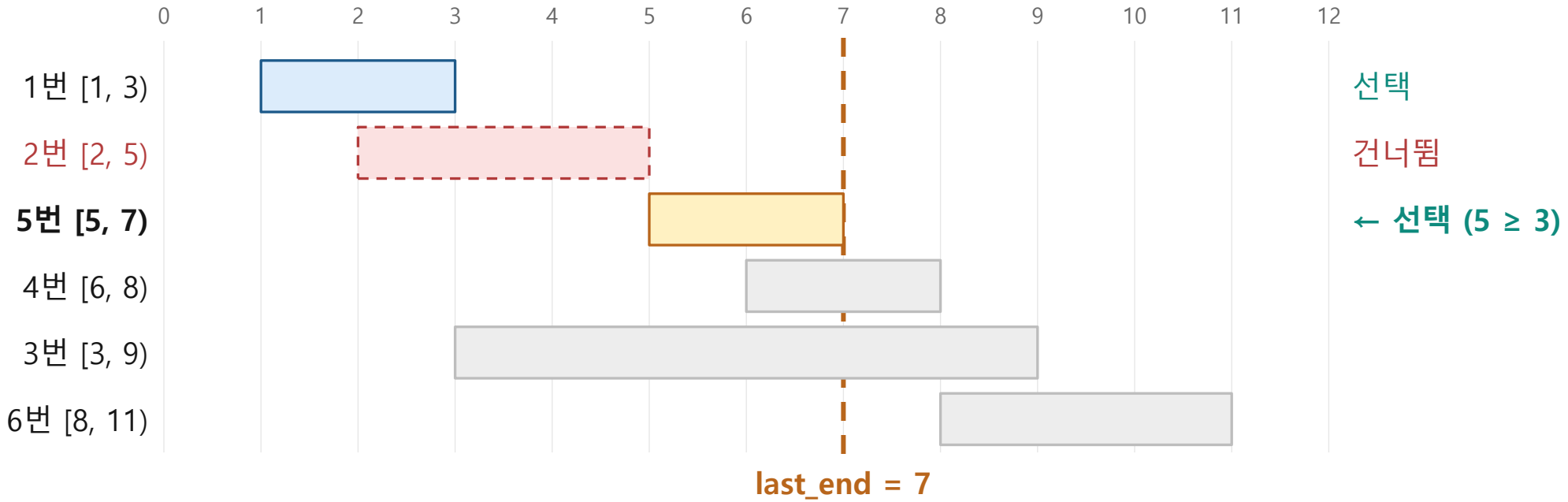
2번은 2시에 시작하는데, 직전에 고른 1번은 3시에 끝납니다.
2 ≥ 3을 만족하지 못해 건너뜁니다. last_end는 3 그대로입니다.



파랑: 고른 회의 노랑: 지금 고른 회의 빨강 점선: 건너뜀 회색: 아직 보지 않은 회의 주황 점선: last_end

③ 5번 [5, 7) · 선택

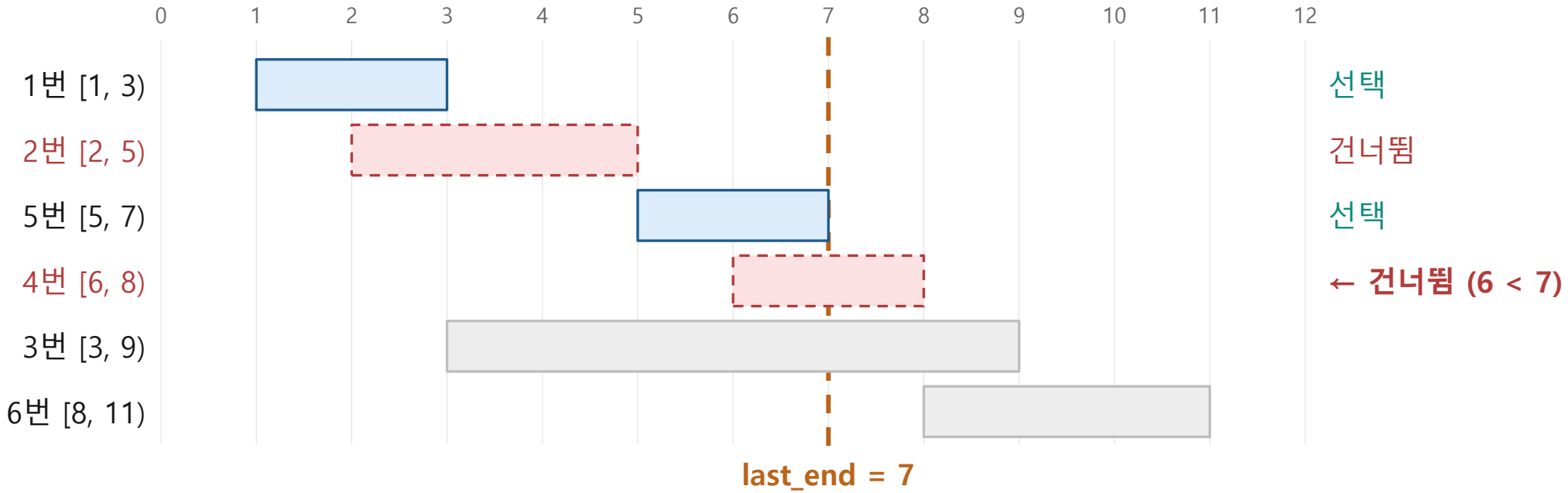
5번은 5시에 시작합니다. 5 ≥ 3이므로 1번과 겹치지 않습니다.
5번을 고르고 last_end를 7로 옮깁니다.



파랑: 고른 회의 노랑: 지금 고른 회의 빨강 점선: 건너뜀 회색: 아직 보지 않은 회의 주황 점선: last_end

④ 4번 [6, 8) · 건너뜀

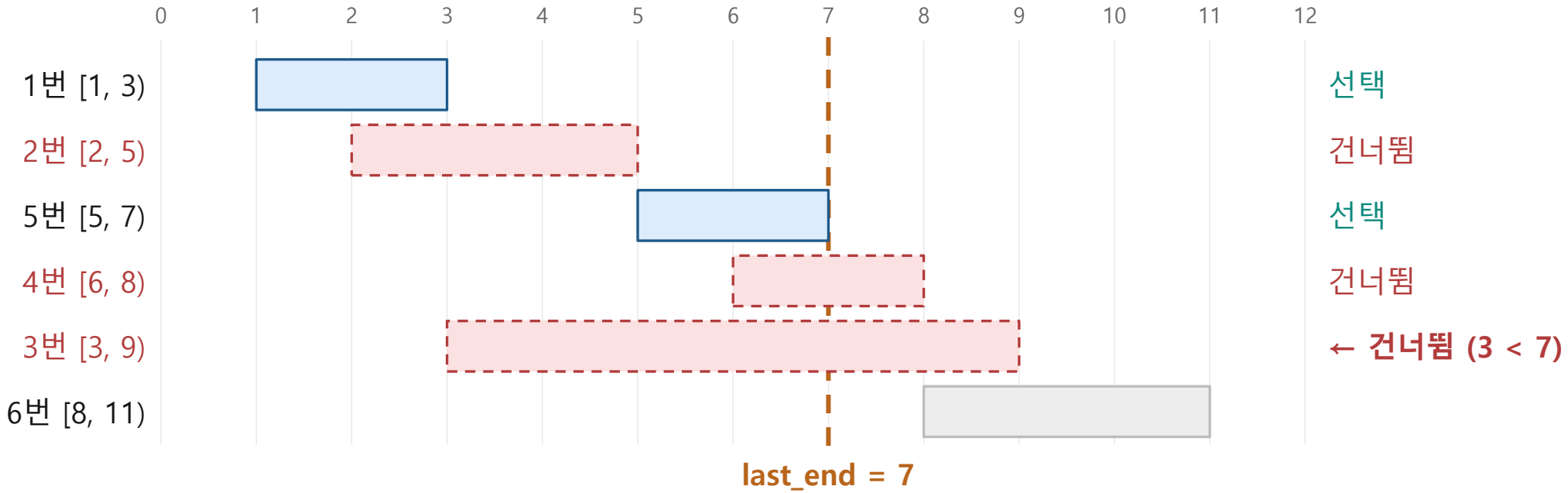
4번은 6시에 시작하는데, 직전에 고른 5번은 7시에 끝납니다.
6 ≥ 7을 만족하지 못해 건너뜁니다.



파랑: 고른 회의 노랑: 지금 고른 회의 빨강 점선: 건너뜀 회색: 아직 보지 않은 회의 주황 점선: last_end

⑤ 3번 [3, 9) · 건너뜀

3번은 3시에 시작합니다. 3 ≥ 7을 만족하지 못해 건너뜁니다.
일찍 시작하지만 9시에 끝나서 정렬 순서로는 다섯 번째입니다.

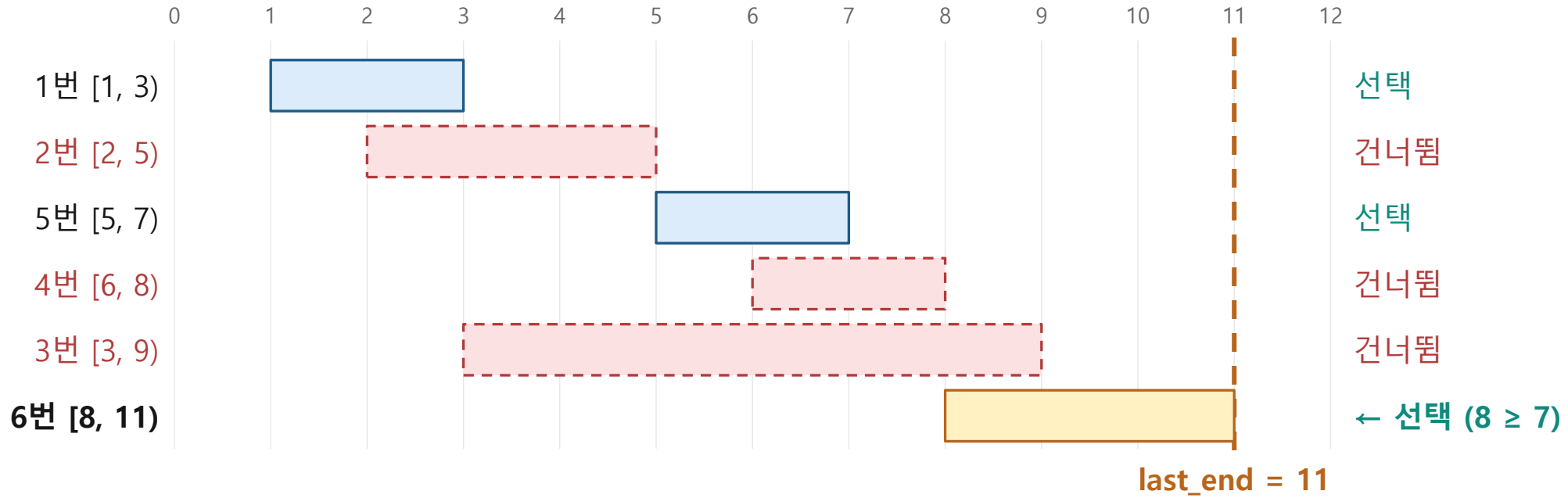


시작 순으로 골랐다면 1번 다음에 [3, 9)인 3번을 골라 2개에 그쳤을 것입니다.

파랑: 고른 회의 노랑: 지금 고른 회의 빨강 점선: 건너뜀 회색: 아직 보지 않은 회의 주황 점선: last_end

⑥ 6번 [8, 11) · 선택

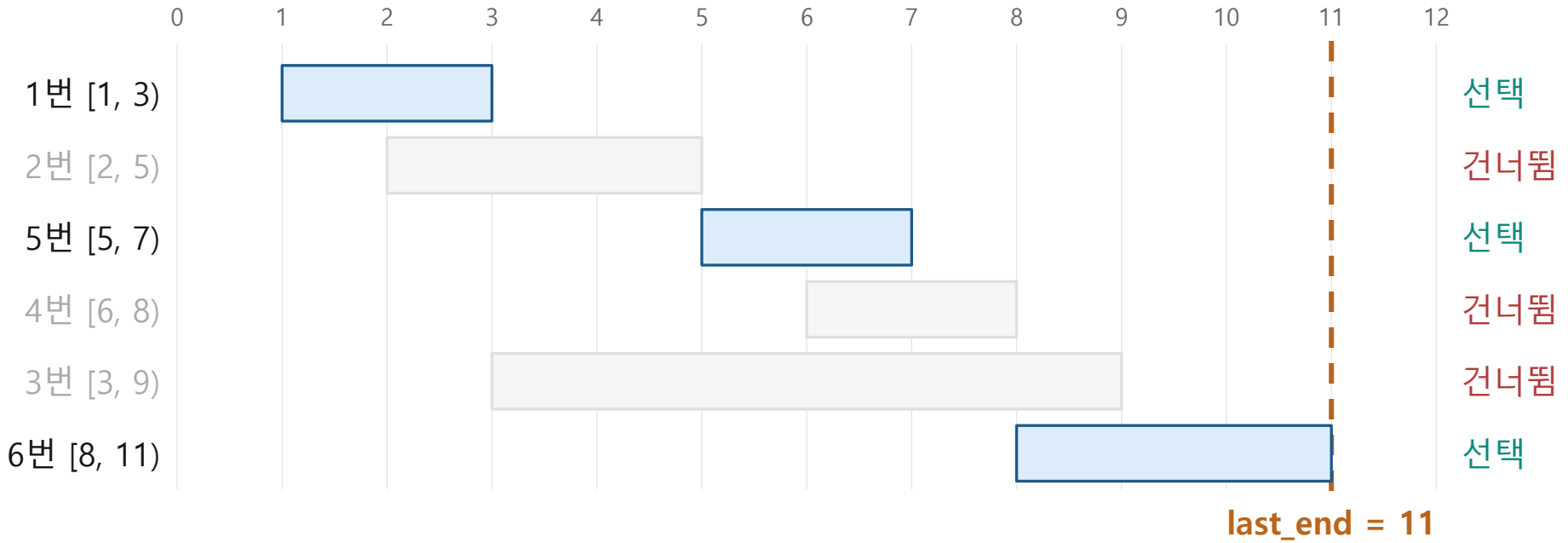
6번은 8시에 시작합니다. 8 ≥ 7이므로 5번과 겹치지 않습니다.
6번을 고르고 last_end를 11로 옮깁니다.



파랑: 고른 회의 노랑: 지금 고른 회의 빨강 점선: 건너뜀 회색: 아직 보지 않은 회의 주황 점선: last_end

완료: 1번, 5번, 6번 · 3개

고른 회의는 1번 [1, 3), 5번 [5, 7), 6번 [8, 11), 모두 3개입니다.
앞의 판정을 되돌린 적은 한 번도 없습니다.



이 3개가 최대인지는 뒤의 '왜 옳은가'에서 확인합니다.

파랑: 고른 회의 회색: 건너뛴 회의 주황 점선: last_end

활동 선택 의사코드

activity_selection.pseudo

```
ACTIVITY_SELECTION(A):              // 회의 a = (start, end)
    A를 end 오름차순으로 정렬한다
    S <- 빈 목록
    last_end <- -∞                  // 첫 회의가 무조건 통과하도록
    for a in A (정렬된 순서):
        if a.start >= last_end:     // 겹치지 않으면
            S에 a를 추가한다
            last_end <- a.end
    return S
```

정렬 한 줄이 그리디의 기준. end 대신 start로 정렬하면 앞의 반례처럼 틀립니다.

last_end 하나만 기억. 고른 회의는 시간 순서로 이어져 마지막 종료 시각만 보면 됩니다.

last_end를 −∞로 시작하는 이유. 어떤 시작 시각도 −∞ 이상이므로 첫 회의가 그대로 통과합니다.

Python ①: 정렬하고 고르는 함수

activity.py

```
def solve_activity_selection(meetings):
    # meetings: (회의ID, 시작시간, 종료시간) 튜플의 리스트
    # 종료 시간 기준 오름차순 정렬 – 그리디의 핵심
    meetings.sort(key=lambda x: x[2])
    selected = []
    last_end = float("-inf")    # 첫 회의가 무조건 통과하도록
    for m in meetings:
        _, start, end = m
        # 직전 회의가 끝난 뒤에 시작하면 채택
        if start >= last_end:
            selected.append(m)
            last_end = end
    return selected
```

key=lambda x: x[2]는 튜플의 인덱스 2, 곧 종료 시각을 정렬 기준으로 씁니다.

Python ②: 데이터 준비와 출력

activity.py

```
# 데이터 준비: (ID, 시작 시간, 종료 시간)
data = [
    (1, 1, 3), (2, 2, 5), (3, 3, 9),
    (4, 6, 8), (5, 5, 7), (6, 8, 11)
]
result = solve_activity_selection(data)

print(f"총 배정된 회의 수: {len(result)}개")
print("-" * 40)
for m in result:
    print(f"[{m[0]}번 회의] {m[1]}시 ~ {m[2]}시")
```

C ①: 구조체와 비교 함수

activity.c

```
#include <limits.h>
#include <stdio.h>
#include <stdlib.h>

typedef struct { int id, start, end; } Meeting;

int by_end(const void *a, const void *b) {
    const Meeting *x = a, *y = b;
    return x->end - y->end;
}
```

C ②: 정렬하고 고르는 함수

activity.c

```
/* 고른 회의를 sel에 담고 그 개수를 돌려준다 */
int solve_activity_selection(Meeting m[], int n, Meeting sel[]) {
    /* 종료 시간 기준 오름차순 정렬 – 그리디의 핵심 */
    qsort(m, n, sizeof(Meeting), by_end);
    int count = 0;
    int last_end = INT_MIN;     /* 첫 회의가 무조건 통과하도록 */
    for (int i = 0; i < n; i++) {
        /* 직전 회의가 끝난 뒤에 시작하면 채택 */
        if (m[i].start >= last_end) {
            sel[count++] = m[i];
            last_end = m[i].end;
        }
    }
    return count;
}
```

C ③: 데이터 준비와 출력

activity.c

```
int main(void) {
    /* 데이터 준비: (ID, 시작 시간, 종료 시간) */
    Meeting data[] = {
        {1, 1, 3}, {2, 2, 5}, {3, 3, 9},
        {4, 6, 8}, {5, 5, 7}, {6, 8, 11}
    };
    Meeting result[6];
    int k = solve_activity_selection(data, 6, result);

    printf("총 배정된 회의 수: %d개\n", k);
    printf("-------------------------------------\n");
    for (int i = 0; i < k; i++)
        printf("[%d번 회의] %d시 ~ %d시\n",
               result[i].id, result[i].start, result[i].end);
    return 0;
}
```

실행 결과

Python과 C의 출력이 같습니다.

OUTPUT

```
총 배정된 회의 수: 3개
----------------------------------------
[1번 회의] 1시 ~ 3시
[5번 회의] 5시 ~ 7시
[6번 회의] 8시 ~ 11시
```

고른 순서 1번, 5번, 6번은 종료 시각 순서이고, 앞의 타임라인 단계와 같습니다.
번호는 입력 순서로 붙인 번호입니다.

복잡도: 정렬 O(n log n) + 한 번 훑기 O(n)

n = 회의의 수

단계	하는 일	시간
정렬	종료 시각 순으로 정렬 (sort, qsort)	O(n log n)
훑기	앞에서부터 한 번, 회의마다 비교 한 번	O(n)
공간	고른 회의 목록	O(n)

O(n log n) + O(n) = O(n log n)

Python의 sort는 최악에도 O(n log n), C의 qsort는 보통 O(n log n)

전체 비용은 정렬이 정합니다.

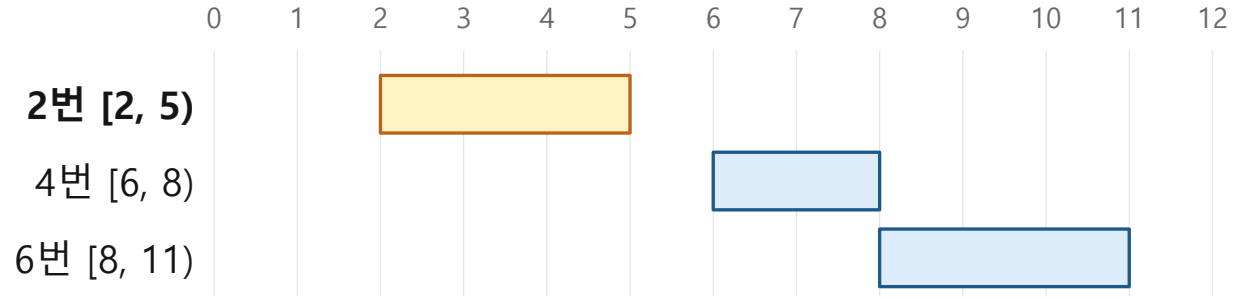
입력이 이미 종료 시각 순이면 O(n)

정렬을 빼고 한 번 훑기만 하면 됩니다.

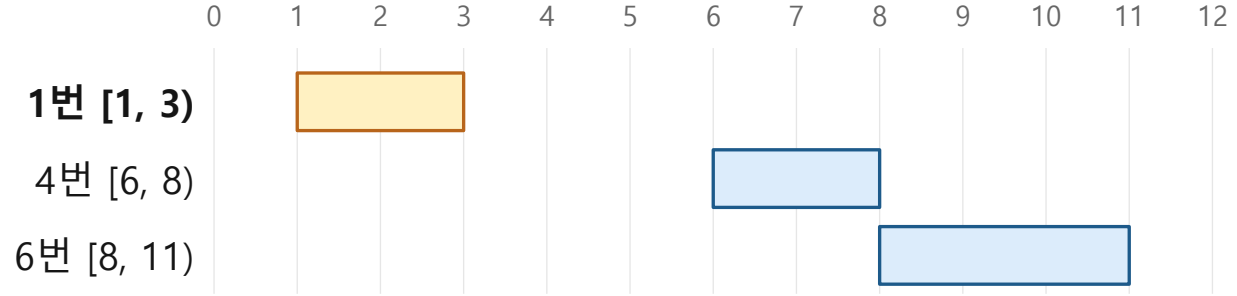
왜 옳은가: 첫 회의를 바꿔 넣어도 최적

최적해 O를 하나 가져옵니다.
예: 2번, 4번, 6번(3개). O의 첫 회의 2번을
그리디의 첫 회의 1번으로 바꿔 봅니다.

최적해 O의 예: 2번, 4번, 6번 (3개)



O의 2번을 1번으로 바꾼 O′: 3개 그대로



노랑: 바꾸는 회의 파랑: 그대로 두는 회의

① g = 그리디의 첫 회의 (1번)
= 전체에서 가장 먼저 끝나는 회의
x = O에서 가장 먼저 끝나는 회의 (2번)
→ g의 종료 ≤ x의 종료 (3 ≤ 5)

② O의 나머지는 x가 끝난 뒤 시작
→ g와도 겹치지 않습니다.

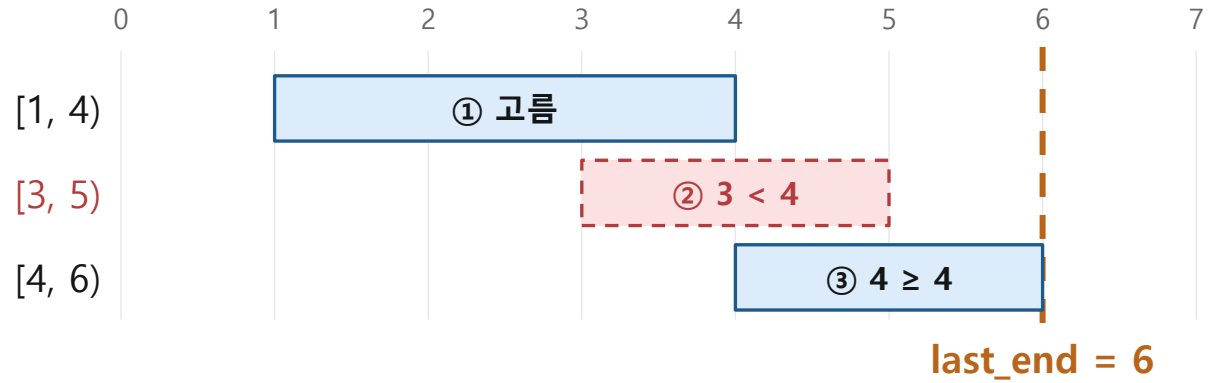
③ 개수가 같으므로 O′도 최적해
→ 남은 회의에도 되풀이합니다.

첫 회의를 포함하는 최적해가 적어도 하나 있습니다.

이 방법이 교환 논증입니다:
최적해의 일부를 탐욕 선택으로
바꿔도 여전히 최적해임을 보입니다.

연습: activity_count · [1, 4), [3, 5), [4, 6)

종료 시각 순으로 정렬되어 들어온 활동의 최대 개수를 셉니다.
시작 ≥ last_end이면 고르고 count를 1 늘립니다.



activity_count = 2

[1, 4)와 [4, 6)

차례	활동	시작 ≥ last_end ?	count	last_end
처음			0	−∞
1	[1, 4)	1 ≥ −∞ → 고름	1	4
2	[3, 5)	3 ≥ 4 아님 → 건너뜀	1	4
3	[4, 6)	4 ≥ 4 → 고름	**2**	6

activity_count: 빈 목록은 0, 시간 O(n)

activity_count.pseudo

```
ACTIVITY_COUNT(activities):     // 종료 시각 순으로 정렬되어 있다
    count <- 0
    last_end <- -∞
    for (start, end) in activities:
        if start >= last_end:   // 끝나는 시각에 바로 시작해도 된다
            count <- count + 1
            last_end <- end
    return count                // 빈 목록이면 0
```

빈 목록. for 문이 한 번도 돌지 않아 count = 0을 그대로 돌려줍니다.
목록[0]을 먼저 넣는 방식은 빈 목록에서 오류가 납니다.

시각이 0 이상으로 주어지면 last_end를 0으로 시작해도 됩니다. 첫 활동이 그대로 통과합니다.

시간 복잡도. 정렬된 입력을 한 번 훑으므로 O(n), 직접 정렬부터 하면 O(n log n)입니다.

확인: 활동 선택

① 회의 [1, 4), [2, 3), [3, 5), [5, 6)을 종료 시각 순 그리디로 고르세요.
고른 회의와 개수는?

② '시작이 빠른 순' 그리디가 틀리는 예를 활동 3개로 만들어 보세요.
강의와 다른 숫자로, 그리디의 결과와 최대 개수를 함께 쓰세요.

③ [2, 4)와 [4, 7)은 함께 고를 수 있습니까? [시작, 종료) 표기로 설명해 보세요.

④ activity_count에 빈 목록을 넣으면 무엇을 돌려줍니까? 그 이유는?

5.4

배낭 문제

보물을 쪼갤 수 있느냐 없느냐,
그 차이 하나로 그리디의 결과가 갈립니다.

배낭 문제: 용량 안에서 가치를 최대로

용량이 정해진 배낭에 보물을 담습니다. 담은 무게의 합이 용량을 넘지 않게 하면서 가치의 합을 가장 크게 만드는 문제가 **배낭 문제**(knapsack problem)입니다.

보물	무게	가치
A	20kg	14,000 G
B	15kg	9,000 G
C	15kg	9,000 G

배낭 용량은 **30kg**입니다. G(골드)는 가치의 단위입니다.

쪼갤 수 있다 → 분할 배낭 (fractional knapsack)
가루처럼 원하는 만큼 덜어 담을 수 있습니다.

쪼갤 수 없다 → 0/1 배낭 (0/1 knapsack)
통째로 담거나(1) 아예 담지 않습니다(0).

이 절은 분할 배낭을 끝까지 본 다음, 같은 숫자로 0/1 배낭을 봅니다.

① 무게당 가치(가성비) 순으로 정렬

1kg에 얼마인지, 즉 무게당 가치를 구해
높은 순서로 세웁니다. 코드에서는 ratio라고 부릅니다.

보물	무게	가치	무게당 가치	담은 무게	얻은 가치
A	20kg	14,000 G	**700 G/kg**	·	·
B	15kg	9,000 G	**600 G/kg**	·	·
C	15kg	9,000 G	**600 G/kg**	·	·

배낭 (용량 30kg)

빈 공간 30kg

무게당 가치 = 가치 ÷ 무게

A: 14,000 ÷ 20 = 700

B: 9,000 ÷ 15 = 600

C: 9,000 ÷ 15 = 600

순서: A → B → C

B와 C는 같으므로
입력 순서대로 B를 먼저 봅니다.

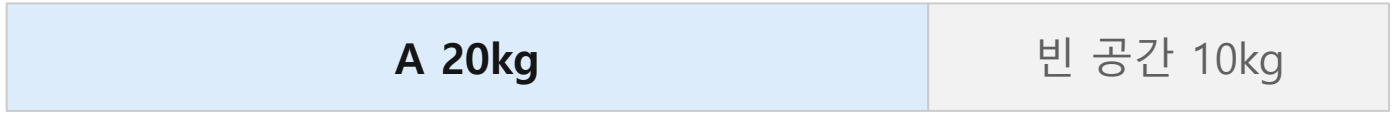
노랑: 정렬 기준인 무게당 가치　　회색 막대: 비어 있는 배낭

② A · 20kg 전부 담기

A는 20kg이고 남은 용량은 30kg입니다.
20 ≤ 30이므로 통째로 담습니다.

보물	무게	가치	무게당 가치	담은 무게	얻은 가치
A	20kg	14,000 G	700 G/kg	**20kg**	**14,000 G**
B	15kg	9,000 G	600 G/kg	·	·
C	15kg	9,000 G	600 G/kg	·	·

배낭 (용량 30kg)



적정성 검사: 20kg ≤ 남은 30kg
→ 통째로 담습니다.

남은 용량

30 − 20 = 10kg

가치 합

14,000 G

파랑: 통째로 담은 보물 회색: 남은 빈 공간

③ B · 남은 10kg만 덜어 담기

B는 15kg인데 남은 용량은 10kg입니다.
통째로는 들어가지 않으니 10kg만 덜어 담습니다.

보물	무게	가치	무게당 가치	담은 무게	얻은 가치
A	20kg	14,000 G	700 G/kg	**20kg**	**14,000 G**
B	15kg	9,000 G	600 G/kg	**10kg (2/3)**	**6,000 G**
C	15kg	9,000 G	600 G/kg	·	·

배낭 (용량 30kg)

A 20kg	B 10kg

적정성 검사: 15kg > 남은 10kg
→ 10kg만 덜어 담습니다.

덜어 담는 비율

10 ÷ 15 = 2/3

얻는 가치

9,000 × 2/3 = 6,000 G

= 10kg × 600 G/kg

파랑: 통째로 담은 보물 초록: 일부만 덜어 담은 보물

④ 배낭이 가득 참 · 20,000 G

남은 용량이 0이 되었으므로 끝냅니다.
C는 확인할 필요가 없습니다.

보물	무게	가치	무게당 가치	담은 무게	얻은 가치
A	20kg	14,000 G	700 G/kg	**20kg**	**14,000 G**
B	15kg	9,000 G	600 G/kg	**10kg (2/3)**	**6,000 G**
C	15kg	9,000 G	600 G/kg	확인하지 않음	·

배낭 (용량 30kg)

A 20kg	B 10kg

해답 검사: 남은 용량 0

→ 반복을 끝냅니다.

무게 합

20 + 10 = 30kg

가치 합

14,000 + 6,000 = 20,000 G

파랑: 통째로 담은 보물　초록: 일부만 덜어 담은 보물　회색 글자: 확인하지 않은 보물

왜 최적인가: 1kg씩 바꿔 보기

그리디 배낭(A 20kg + B 10kg) 안의 1kg은 모두 600 G 이상이고,
밖에 남은 1kg(B 5kg, C 15kg)은 모두 600 G입니다.

바꾸는 1kg	가치 변화
안의 A 1kg을 빼고 밖의 C 1kg을 넣기	−700 + 600 = −100 G
안의 B 1kg을 빼고 밖의 C 1kg을 넣기	−600 + 600 = 0 G
안의 A 1kg을 빼고 비워 두기	−700 G

어떻게 바꿔도 가치가 늘지 않습니다. 그래서 20,000 G가 최대입니다.

다른 어떤 배낭도 그리디 배낭에서 1kg씩 바꿔 만들 수 있습니다.
바꿀 때마다 가치가 줄거나 그대로이므로 그리디보다 클 수 없습니다.
5.3절 '왜 옳은가'와 같은 교환 논증입니다. 5.5절에서 정리합니다.

의사코드: 무게당 가치 순으로 채우기

fractional_knapsack.pseudo

```
FRACTIONAL_KNAPSACK(items, capacity):
    items를 ratio(무게당 가치)가 큰 순서로 정렬
    total_value <- 0
    current_weight <- 0
    for each item in items:
        if current_weight + item.weight <= capacity then   // 통째로
            current_weight += item.weight
            total_value += item.value
        else                                                // 덜어 담기
            room <- capacity - current_weight               // 남은 공간
            total_value += room × item.ratio
            break
    return total_value
```

단계	의사코드에서
① 선택	정렬된 순서대로, 남은 보물 중 무게당 가치가 가장 높은 것을 봅니다.
② 적정성 검사	통째로 들어가는가? 아니면 남은 공간(room)만큼만 덜어 담습니다.
③ 해답 검사	덜어 담으면 배낭이 가득 차므로 break. 덜어 담기는 많아야 한 번입니다.

Python ①: 보물 하나를 나타내는 Item

knapsack.py

```
class Item:
    def __init__(self, name, weight, value):
        self.name = name
        self.weight = weight
        self.value = value
        self.ratio = value / weight     # 가성비
```

Python ②: 배낭 채우기 함수

knapsack.py

```
def solve_fractional_knapsack(capacity, items):
    # 가성비 내림차순 정렬 – 그리디의 핵심
    items.sort(key=lambda x: x.ratio, reverse=True)
    total_value = 0.0
    current_weight = 0.0
    print(f"--- 배낭 채우기 시작 (용량: {capacity}kg) ---")
    for item in items:
        if current_weight + item.weight <= capacity:
            current_weight += item.weight       # 통째로
            total_value += item.value
            print(f"-> {item.name} 전체 추가",
                  f"(현재 무게: {current_weight:g}kg)")
        else:
            room = capacity - current_weight    # 남은 공간
            total_value += room * item.ratio
            print(f"-> {item.name} {room:g}kg만 쪼개서 추가 (가득 참!)")
            break
    return total_value
```

Python ③: 보물 세 개로 실행

knapsack.py

```
items = [Item("A(황금 가루)", 20, 14000),
         Item("B(은 가루)", 15, 9000),
         Item("C(보석 가루)", 15, 9000)]
best = solve_fractional_knapsack(30, items)
print("-" * 36)
print(f"배낭의 최대 가치 합계: {best:g} 골드")
```

C ①: 구조체와 make_item

knapsack.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct {
    const char *name;
    int weight, value;
    double ratio;                          /* 가성비 */
} Item;

Item make_item(const char *name, int weight, int value) {
    Item it = {name, weight, value, (double)value / weight};
    return it;
}
```

C ②: 비교 함수와 정렬

knapsack.c

```
/* 가성비 내림차순. 같으면 이름순(B가 C보다 앞) */
int by_ratio(const void *a, const void *b) {
    const Item *x = a, *y = b;
    if (x->ratio > y->ratio) return -1;
    if (x->ratio < y->ratio) return 1;
    return strcmp(x->name, y->name);
}

double solve_fractional_knapsack(int capacity, Item items[], int n) {
    qsort(items, n, sizeof(Item), by_ratio);   /* 그리디의 핵심 */
    double total_value = 0, current_weight = 0;
    printf("--- 배낭 채우기 시작 (용량: %dkg) ---\n", capacity);
```

C ③: 배낭을 채우는 반복문

knapsack.c

```
    for (int i = 0; i < n; i++) {
        Item it = items[i];
        if (current_weight + it.weight <= capacity) {
            current_weight += it.weight;              /* 통째로 */
            total_value += it.value;
            printf("-> %s 전체 추가 (현재 무게: %gkg)\n",
                   it.name, current_weight);
        } else {
            double room = capacity - current_weight;  /* 남은 공간 */
            total_value += room * it.ratio;
            printf("-> %s %gkg만 쪼개서 추가 (가득 참!)\n", it.name, room);
            break;
        }
    }
    return total_value;
}
```

C ④: main에서 보물 세 개로 실행

knapsack.c

```
int main(void) {
    Item items[] = {make_item("A(황금 가루)", 20, 14000),
                    make_item("B(은 가루)", 15, 9000),
                    make_item("C(보석 가루)", 15, 9000)};
    double best = solve_fractional_knapsack(30, items, 3);
    printf("----------------------------------\n");
    printf("배낭의 최대 가치 합계: %g 골드\n", best);
    return 0;
}
```

실행 결과: 20,000 골드

OUTPUT

```
--- 배낭 채우기 시작 (용량: 30kg) ---
-> A(황금 가루) 전체 추가 (현재 무게: 20kg)
-> B(은 가루) 10kg만 쪼개서 추가 (가득 참!)
-----------------------------------
배낭의 최대 가치 합계: 20000 골드
```

A 20kg 전부 + B 10kg = 30kg, 14,000 + 6,000 = 20,000 골드입니다.
Python과 C의 출력이 같습니다.

Python에서 reverse=True를 빼면 무게당 가치가 낮은 B, C부터 담아 18,000 골드가 됩니다.
정렬 방향이 그리디의 기준입니다.

복잡도: 정렬이 O(n log n)

n은 보물의 개수입니다.

정렬	한 번 훑기	전체
무게당 가치로 n개를 정렬 O(n log n)	보물마다 한 번씩 보고 덜어 담기는 많아야 한 번 O(n)	O(n log n) + O(n) = O(n log n)

정렬이 전체 비용을 결정합니다.

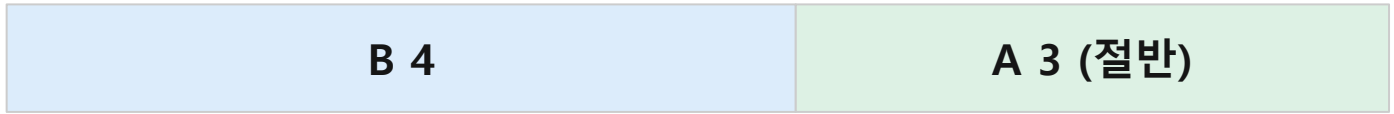
보물이 백만 개여도 정렬 한 번과 훑기 한 번이면 최적해가 나옵니다. 단, 쪼갤 수 있을 때만입니다.

연습: 용량 7 · A(6, 18) · B(4, 20)

쪼갤 수 있는 배낭, 보물은 (무게, 가치)입니다.
무게당 가치 A 18 ÷ 6 = 3, B 20 ÷ 4 = 5 → B부터 봅니다.

보물	무게	가치	무게당 가치	담은 무게	얻은 가치
B	4	20	**5**	**4 (전부)**	**20**
A	6	18	**3**	**3 (3/6)**	**9**

배낭 (용량 7)



① B: 4 ≤ 7 → 전부, 가치 20

② 남은 용량 7 − 4 = 3

③ A: 6 > 3 → 3만 덜어 담기

3 ÷ 6 = 1/2 → 18 × 1/2 = 9

합계 20 + 9 = 29

파랑: 통째로 담은 보물　초록: 일부만 덜어 담은 보물　노랑: 무게당 가치

연습: 용량 5 · A(2, 8) · B(4, 16)

무게당 가치는 A 8 ÷ 2 = 4, B 16 ÷ 4 = 4로 같습니다.
같으면 앞에 있는 A부터 봅니다.

보물	무게	가치	무게당 가치	담은 무게	얻은 가치
A	2	8	**4**	**2 (전부)**	**8**
B	4	16	**4**	**3 (3/4)**	**12**

배낭 (용량 5)



① A: 2 ≤ 5 → 전부, 가치 8

② 남은 용량 5 − 2 = 3

③ B: 4 > 3 → 3만 덜어 담기

3 ÷ 4 = 3/4 → 16 × 3/4 = 12

합계 8 + 12 = 20

B부터 담아도 16 + 4 = 20.
무게당 가치가 모두 4라서 5 × 4 = 20입니다.

파랑: 통째로 담은 보물 초록: 일부만 덜어 담은 보물 노랑: 무게당 가치

0/1 배낭: 같은 숫자, 쪼갤 수만 없다

이번 보물은 조각상이나 갑옷처럼 덜어 낼 수 없습니다.
통째로 담거나(1) 담지 않습니다(0).

보물	무게	가치	무게당 가치
A	20kg	14,000 G	700 G/kg
B	15kg	9,000 G	600 G/kg
C	15kg	9,000 G	600 G/kg

용량 30kg과 보물 표가 앞과 똑같습니다.
달라진 규칙은 "쪼갤 수 없다" 하나뿐입니다.

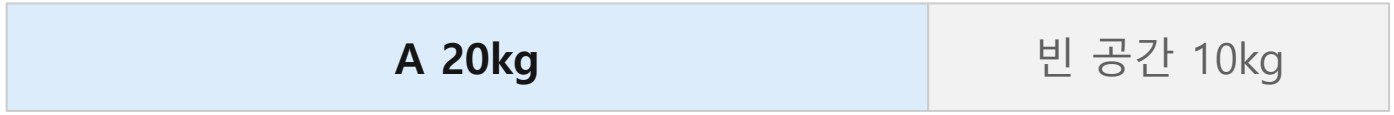
그리디 규칙: 무게당 가치 순서로 보며,
남은 용량에 통째로 들어가면 담고 아니면 건너뜁니다.

0/1 그리디: A만 담고 14,000 G

A(20kg)를 담으면 남은 용량은 10kg입니다.
B와 C는 15kg이라 통째로 들어가지 않아 건너뜁니다.

보물	무게	가치	무게당 가치	결정	얻은 가치
A	20kg	14,000 G	700 G/kg	**담음**	**14,000 G**
B	15kg	9,000 G	600 G/kg	**건너뜀**	0 G
C	15kg	9,000 G	600 G/kg	**건너뜀**	0 G

배낭 (용량 30kg)



A: 20kg ≤ 30kg → 담음

B: 15kg > 남은 10kg → 건너뜀

C: 15kg > 남은 10kg → 건너뜀

쪼갤 수 없으니 10kg이 빈 채로 끝납니다.

그리디의 답

14,000 G

파랑: 담은 보물 빨강: 들어가지 않아 건너뛴 보물 회색: 남은 빈 공간

가능한 조합 전부: B + C가 최대

보물마다 담거나 말거나 두 가지이므로 조합은 2 × 2 × 2 = 8가지입니다.

담은 보물	무게 합	가치 합	판정
없음	0kg	0 G	가능
A	20kg	14,000 G	**가능 (그리디의 답)**
B	15kg	9,000 G	가능
C	15kg	9,000 G	가능
A + B	35kg	—	30kg 초과
A + C	35kg	—	30kg 초과
B + C	**30kg**	**18,000 G**	**가능 · 최대**
A + B + C	50kg	—	30kg 초과

최적해는 B + C = 18,000 G입니다.
그리디(14,000 G)보다 4,000 G 많습니다.

연습: 0/1 용량 50 · 그리디는 160

A(10, 60), B(20, 100), C(30, 120)입니다(무게, 가치).
무게당 가치 A 6, B 5, C 4 → A, B, C 순서로 봅니다.

보물	무게	가치	무게당 가치	결정	얻은 가치
A	10	60	**6**	**담음**	**60**
B	20	100	**5**	**담음**	**100**
C	30	120	**4**	**건너뜀**	0

배낭 (용량 50)

A 10	**B 20**	빈 공간 20

① A: 10 ≤ 50 → 담음, 남은 40

② B: 20 ≤ 40 → 담음, 남은 20

③ C: 30 > 20 → 건너뜀

그리디의 답

60 + 100 = 160

파랑: 담은 보물　빨강: 건너뛴 보물　노랑: 무게당 가치　회색: 남은 빈 공간

연습: 조합을 다 보면 B + C = 220

담은 보물	무게 합	가치 합	판정
없음	0	0	가능
A	10	60	가능
B	20	100	가능
C	30	120	가능
A + B	30	160	**가능 (그리디의 답)**
A + C	40	180	가능
B + C	**50**	**220**	**가능 · 최대**
A + B + C	60	—	50 초과

최적해는 B + C = 220입니다. 그리디의 답 160보다 60 큽니다.

무게당 가치 1등인 A를 담는 순간, 남은 40에는 B(20)와 C(30)를 함께 넣을 수 없습니다.
첫 선택부터 최적해에서 벗어난 것입니다.

0/1 배낭에서 그리디가 실패하는 이유

① 첫 선택이 최적해에 없다

A를 담으면 남은 10kg에 B도 C도 들어가지 않습니다. 최적해 B + C에는 A가 없습니다.

② 분할 배낭과의 차이

쪼갤 수 있으면 남은 10kg을 B로 채워 손해가 없습니다.
쪼갤 수 없으면 1kg 교환을 쓸 수 없고, 이 예처럼 빈 공간이 손해로 남습니다.

③ 모든 조합을 보면?

보물 n개를 담거나 말거나: 2^n가지. 보물이 30개면 10억 가지가 넘습니다.

④ 6장 동적 계획법

같은 부분 문제의 답을 표에 저장해 다시 계산하지 않습니다. 0/1 배낭의 정확한 답은 6장에서 구합니다.

0/1 배낭에서 무게당 가치 그리디는 최적해를 보장하지 못합니다.

확인: 배낭 문제

보물 세 개와 배낭 용량 6kg이 있습니다.

보물	무게	가치
X	4kg	20 G
Y	3kg	18 G
Z	2kg	8 G

① 쪼갤 수 있을 때(분할 배낭) 최대 가치는 얼마입니까?

② 쪼갤 수 없을 때(0/1 배낭) 무게당 가치 그리디의 답은 얼마이고, 실제 최대는 얼마입니까?

③ 분할 배낭에서 무게당 가치 순서가 최적인 이유를 한 문장으로 말해 보세요.

5.5

그리디가 통하는 조건

지금의 최선을 확정해도 괜찮으려면
문제에 어떤 성질이 있어야 할까요?

그리디가 최적해를 보장하는 두 조건

그리디는 매 단계 지금 기준의 최선을 확정하고 되돌리지 않습니다.
이 방식이 최적해를 낸다고 보장하려면 두 성질을 보여야 합니다.

① 탐욕적 선택 속성

greedy choice property
지금 가장 좋아 보이는 선택을
포함하는 최적해가 있다

② 최적 부분 구조

optimal substructure
그 선택을 하고 남은 문제의 최적해가
전체 최적해의 나머지가 된다

①로 첫 선택이 안전하고, ②로 남은 문제를 같은 방법으로 풀 수 있습니다.

이 둘을 단계마다 이어 붙이면 그리디의 답이 전체 최적해가 됩니다. 둘 중 하나라도 보이지 못하면 보장이 사라집니다.

① 탐욕적 선택 속성 (greedy choice property)

지금 기준으로 가장 좋아 보이는 선택을 해도, 그 선택을 포함하는 최적해가 적어도 하나 있다는 성질입니다.

판별 질문: 이 선택을 포함하는 최적해가 있는가?

예

그 선택을 확정하고 남은 문제로 넘어가도 최적해를 놓치지 않습니다.

아니오

그 선택을 하는 순간 최적해로 가는 길이 모두 막힙니다.

"안전하다"는 다른 선택보다 더 좋다는 뜻이 아니라, 이 선택을 해도 최적해로 가는 길이 남아 있다는 뜻입니다. 최적해가 여러 개라면 그중 하나만 이 선택을 포함해도 됩니다.

판별 질문 적용: 예와 아니오

표준 동전 — 예

1,260원의 최적해(동전 6개)에 그리디가 먼저 고른 500원이 들어 있습니다.

400원 동전 — 아니오

800원의 최적해는 400 + 400 하나뿐입니다. 그리디가 먼저 고른 500원이 없습니다.

활동 선택 — 예

가장 먼저 끝나는 1번 [1, 3)을 포함하는 최적해가 있습니다. 5.3에서 교환으로 확인했습니다.

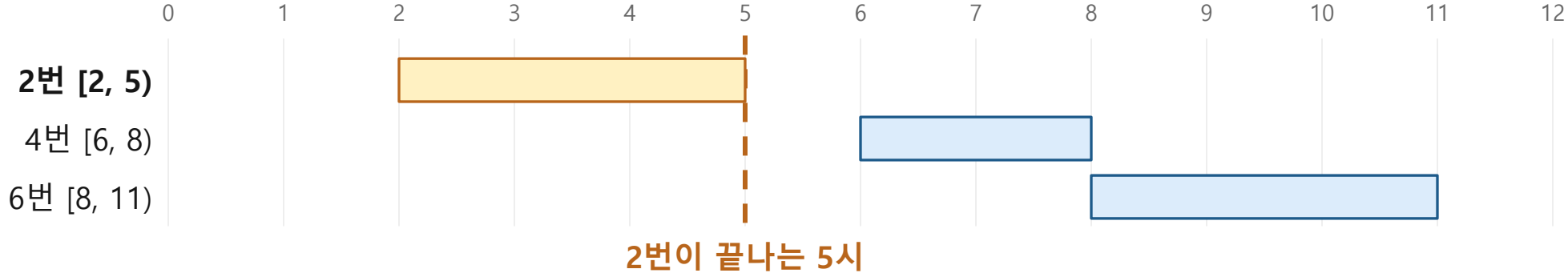


왼쪽 그림: 표준 동전 — 500원을 먼저 써도 안전　　오른쪽 그림: 400원 동전 — 500원을 잡는 순간 400 + 400을 놓칩니다

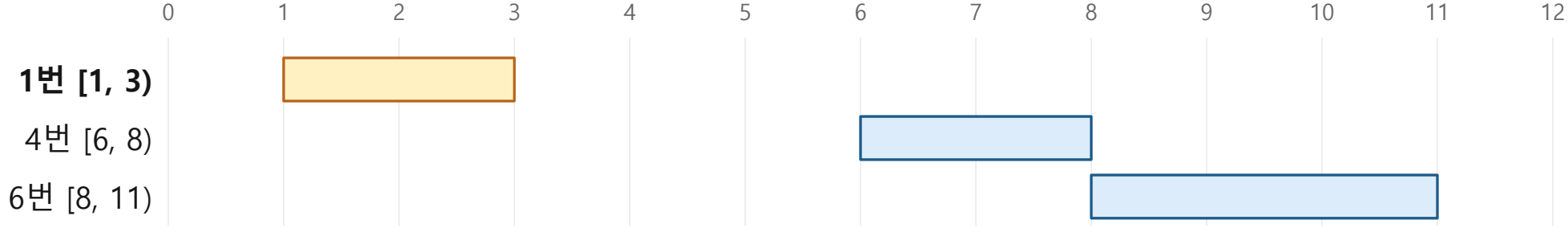
5.3에서 본 교환: 첫 회의를 1번으로

5.3의 교환 논증: 최적해 O = {2번, 4번, 6번}의 첫 회의 2번을
가장 먼저 끝나는 1번으로 바꾸면 O′ = {1번, 4번, 6번}입니다.

O = {2번, 4번, 6번}: 3개



O′ = {1번, 4번, 6번}: 여전히 3개, 겹침 없음



노랑: 바꾼 회의 (2번 → 1번) 파랑: 그대로 둔 회의

교환 논증의 핵심

최적해의 일부를 탐욕 선택으로 바꿔도 여전히 최적해임을 보입니다.

① 어떤 최적해 O에서 시작합니다

O에 탐욕 선택 g가 없다고 합시다.

② O의 일부를 g로 바꿉니다

바꾼 O′도 문제의 조건(겹치지 않음, 용량, 금액)을 지킵니다.

③ 값이 나빠지지 않음을 보입니다

O′의 값이 O보다 나쁘지 않으면 O′도 최적해이고, O′에는 g가 들어 있습니다.

예	바꾸는 것	나빠지지 않는 이유
활동 선택 (5.3)	O의 첫 회의 → 1번 [1, 3)	1번이 더 일찍 끝나 겹치지 않고, 개수 그대로
분할 배낭 (5.4)	안의 B·C 1kg → 밖의 A 1kg	1kg당 600 G → 700 G, 가치가 늘어남
표준 동전 (5.2)	100원 5개 → 500원 1개	같은 금액을 동전 4개 적게 (다음 슬라이드)

결론은 늘 같습니다. 탐욕 선택을 포함하는 최적해가 있다 → 탐욕적 선택 속성이 성립합니다.

교환 예: 표준 동전에서 500원이 안전한 이유

예: 1,260원을 100원 12개 + 50원 1개 + 10원 1개(14개)로 주면, 100원 5개를 500원 1개로 두 번 바꿔 6개가 됩니다.

바꿀 수 있는 묶음	개수 변화
100원 5개 → 500원 1개	5개 → 1개
50원 2개 → 100원 1개	2개 → 1개
10원 5개 → 50원 1개	5개 → 1개

최적해에는 이런 묶음이 없습니다. 그래서 100원은 4개 이하, 50원은 1개 이하, 10원은 4개 이하입니다. 그래서 최적해에서 500원을 뺀 나머지 동전의 합은 많아야 400 + 50 + 40 = 490원입니다.

500원 이상이면 최적해에 500원이 반드시 있습니다 → 500원을 먼저 골라도 안전

400원 동전이 생기면 이 교환이 깨집니다. 800원의 최적해 400 + 400에는 500원 1개로 바꿀 묶음이 없습니다.

② 최적 부분 구조 (optimal substructure)

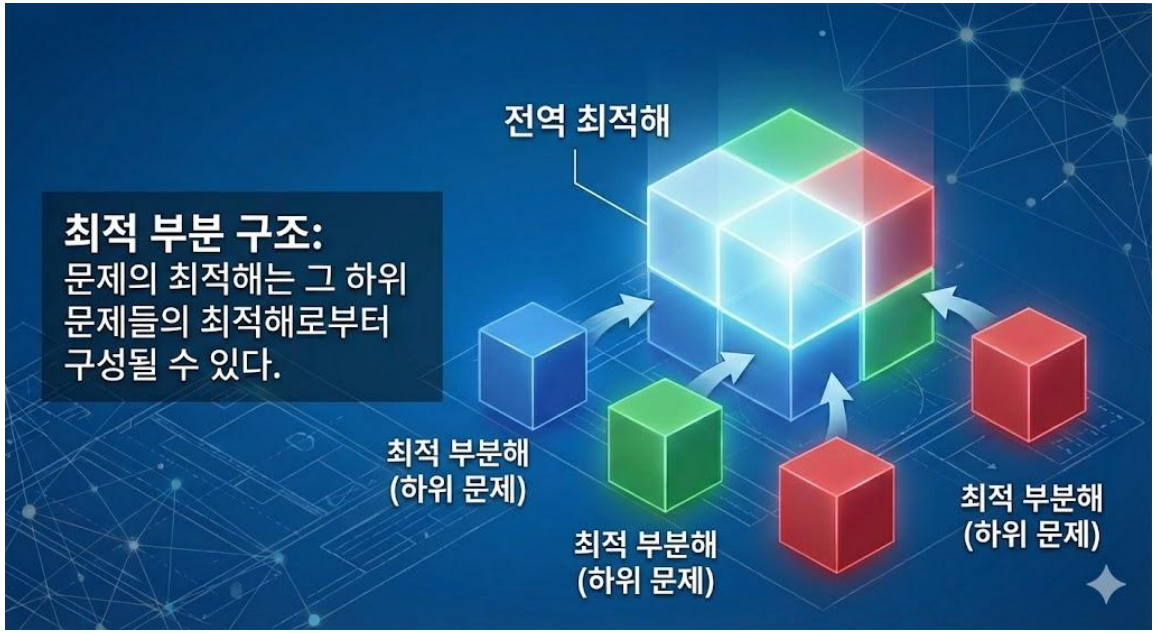
선택하고 남은 부분 문제(하위 문제)의 최적해에 그 선택을 더하면 전체 최적해가 된다는 성질입니다. 거꾸로 말하면 전체 최적해 안에 부분 문제의 최적해가 들어 있습니다.

예: 활동 선택

1번 [1, 3)을 고르면 남은 문제는 "3시 이후에 시작하는 회의 중 최대 개수 고르기"입니다. 그 최적해에 1번을 더하면 전체 최적해가 됩니다.

그리디는 선택을 확정한 뒤 남은 문제를 같은 방법으로 풉니다. 그래서 이 성질이 필요합니다.

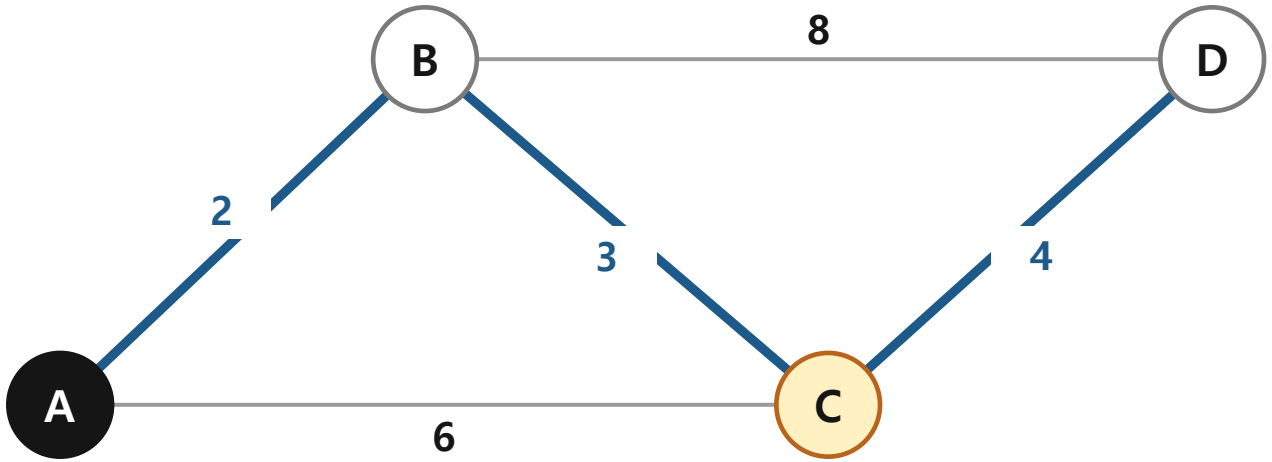
4장의 최대 구간 합처럼 분할 정복으로 푸는 최적화 문제, 6장의 동적 계획법도 이 성질을 씁니다.



작은 문제의 최적해(그림의 최적 부분해)를 모아 전체 최적해(그림의 전역 최적해)를 만듭니다.

② 성립 예: 최단 경로의 부분 경로

A → D 최단 경로는 A → B → C → D = 9입니다. 그 안의 A → B → C도 A에서 C까지의 최단일까요?



A → D: A → C → D = 10, A → B → D = 10

A → C로 가는 길

A → B → C = 2 + 3 = 5

A → C 직행 = 6

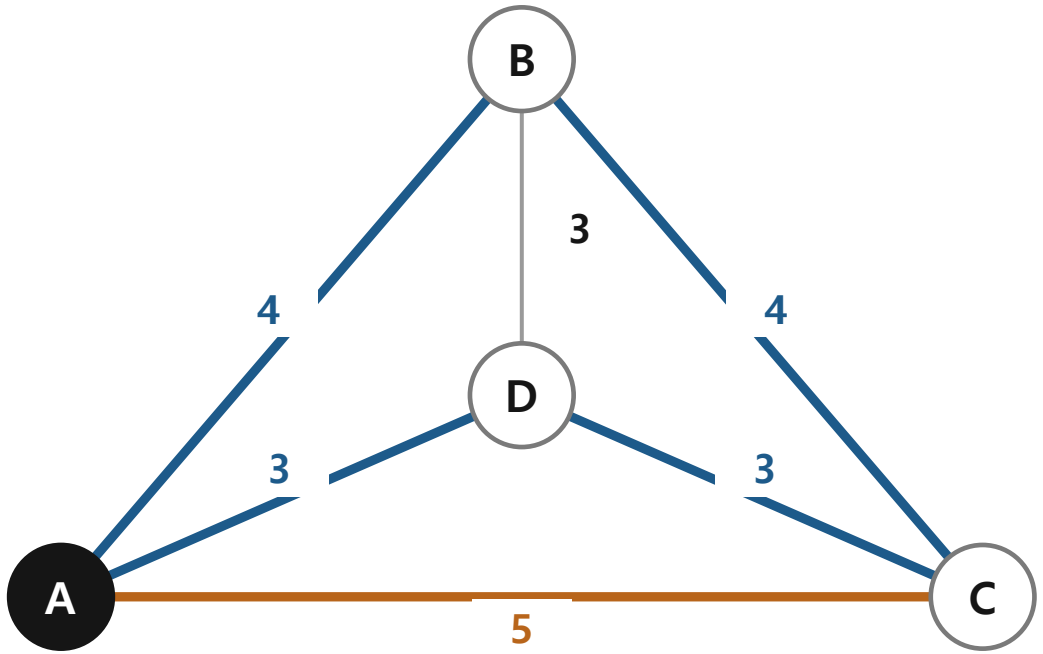
5가 A → C 최단

5보다 짧은 A → C 길이 있다면 그 길로 바꿔 A → D가 9보다 짧아집니다. 9가 최단이라는 사실과 모순입니다.

파랑: A에서 D까지의 최단 경로 검정 정점: 출발점 A 노랑 정점: 중간 지점 C

TSP: '두 도시 사이 최단'으로 쪼갤 수 없다

최적 순회는 A → B → C → D → A = 14입니다(다른 두 순회는 15). 그 안의 A → B → C를 떼어 봅니다.



파랑: 최적 순회 주황: 더 짧지만 B를 건너뛰는 A → C 검정 정점: 출발지 A

외판원 순회(TSP): 모든 도시를 한 번씩 들르고 출발지로 돌아오는 가장 짧은 순회. 5.7절에서 자세히 봅니다.

A → B → C = 4 + 4 = 8

A → C 직행 = 5가 더 짧습니다

바꾸면 A → C → D → A: B를 건너뜁니다

부분 문제에 '들른 도시'가 들어가야 합니다

현재 도시와 들른 도시 집합을 함께 기억하면 DP로 풉니다(6장).

주의: ②만으로는 그리디를 보장할 수 없다

문제 (그리디의 기준)	① 탐욕적 선택	② 최적 부분 구조	그리디의 결과
활동 선택 (종료가 빠른 순)	성립	성립	**최적**
분할 배낭 (무게당 가치 순)	성립	성립	**최적**
400원 동전 (큰 동전부터)	**깨짐**	성립	보장 못 함 → 6장 DP
0/1 배낭 (무게당 가치 순)	**깨짐**	성립	보장 못 함 → 6장 DP
최단 경로, 간선 비용 ≥ 0 (다익스트라)	성립	성립	**최적 (5.8)**
최단 경로, 음수 간선 (다익스트라)	**깨짐**	성립	틀릴 수 있음 (5.8)

400원 동전과 0/1 배낭은 ①이 깨진 예입니다. ②는 성립하므로 6장 동적 계획법으로 풀립니다.

최단 경로는 음수 간선이 있어도(한 바퀴 돌수록 짧아지는 순환이 없다면) ②가 성립합니다. 다익스트라가 옳으려면 "간선 비용이 모두 0 이상이면, 아직 확정하지 않은 곳 중 지금 가장 가까운 곳은 더 가까워지지 않는다"는 ①의 근거가 더 필요합니다.

반례 모음: '항상 최적'은 반례 하나로 깨진다

5.2에서 본 대로 "이 그리디는 항상 최적해를 준다"는 모든 입력에 대한 주장입니다. 이 장에서 만난 반례를 모아 봅니다.

반박은 반례 하나면 충분합니다. 조건을 지키는 입력 하나에서 그리디보다 더 좋은 해를 보이면 됩니다.

주장	반례 입력	그리디	더 좋은 해
큰 동전부터 고르면 최적	동전 {1, 3, 4}, 6원	4 + 1 + 1 → 3개	**3 + 3 → 2개**
큰 동전부터 고르면 최적	동전 {500, 400, 100, 50, 10}, 800원	500 + 100 × 3 → 4개	**400 + 400 → 2개**
무게당 가치 순이면 최적 (0/1 배낭)	30kg, A · B · C	A → 14,000 G	**B + C → 18,000 G**
일찍 시작하는 회의부터 고르면 최적	[1, 10), [2, 3), [4, 5)	[1, 10) → 1개	**[2, 3), [4, 5) → 2개**

반대로 "항상 최적"임을 보이려면 예를 아무리 많이 들어도 부족합니다. 교환 논증처럼 모든 입력에 통하는 논증이 필요합니다.

확인: 그리디가 통하는 조건

① 동전 {1, 3, 4}로 6원을 만들 때, 그리디의 첫 선택 4원은 탐욕적 선택 속성을 만족합니까?

② 교환 논증의 핵심을 한 문장으로 말해 보세요. 표준 동전에서는 무엇을 무엇으로 바꿉니까?

③ "무게당 가치 그리디는 0/1 배낭에서 항상 최적이다"를 반박하려면 무엇이 필요합니까?

④ 400원 동전 문제에서 두 조건 중 깨진 것은 무엇입니까? 이 문제를 정확히 풀려면 어떤 방법을 씁니까?

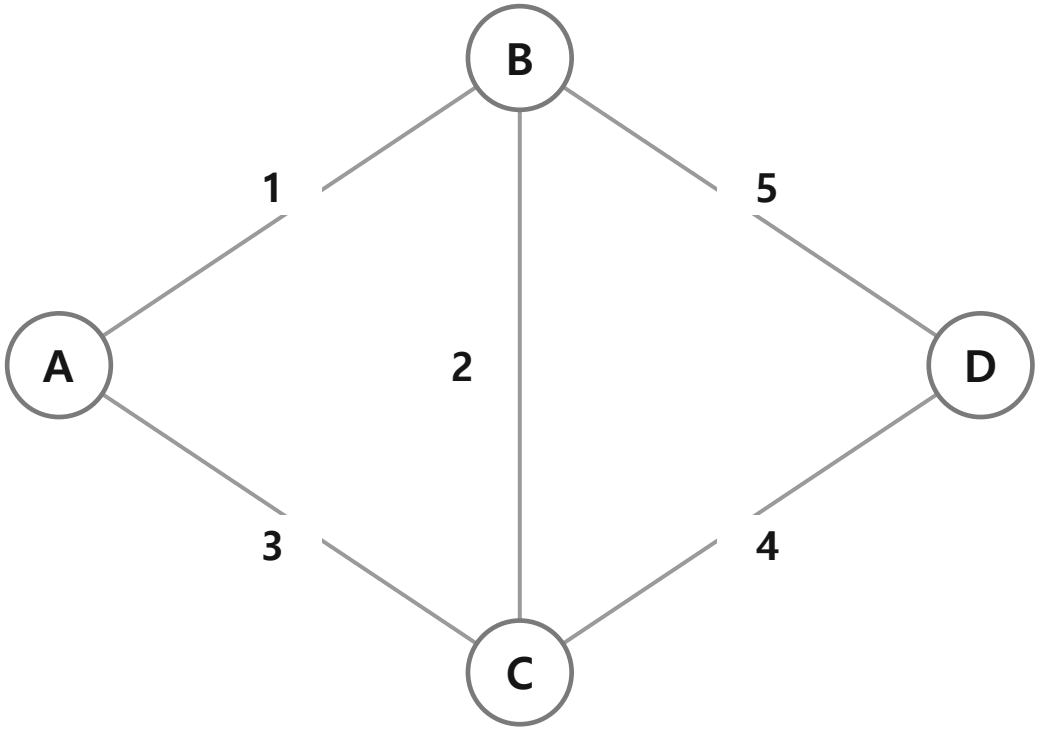
5.6

크루스칼 알고리즘

모든 도시를 잇되 도로 비용은 최소로,
가장 싼 도로부터 하나씩 골라 나갑니다.

그래프 용어: 정점, 간선, 가중치

도시와 도로를 그래프로 나타냅니다. 왼쪽은 정점 4개, 간선 5개짜리 그래프입니다.



원: 정점 선: 간선 선 위의 수: 가중치

용어	뜻	이 그림에서
정점 (vertex)	점. 예: 도시	V = 4 (A~D)
간선 (edge)	두 정점을 잇는 선. 예: 도로	E = 5
가중치 (weight)	간선에 붙은 비용	A – B 1, C – D 4
경로 (path)	간선을 따라 이어진 길	A – B – D
사이클 (cycle)	출발점으로 돌아오는 닫힌 길	A – B – C – A
신장 (spanning)	모든 정점을 포함한다	A~D 모두

V는 정점의 수, E는 간선의 수를 나타내는 기호입니다.

최소 신장 트리 (MST)

신장 트리: 모든 정점을 잇고, 사이클이 없는 간선 모음

최소 신장 트리(Minimum Spanning Tree, MST): 신장 트리 중 가중치 합이 가장 작은 것

모두 연결

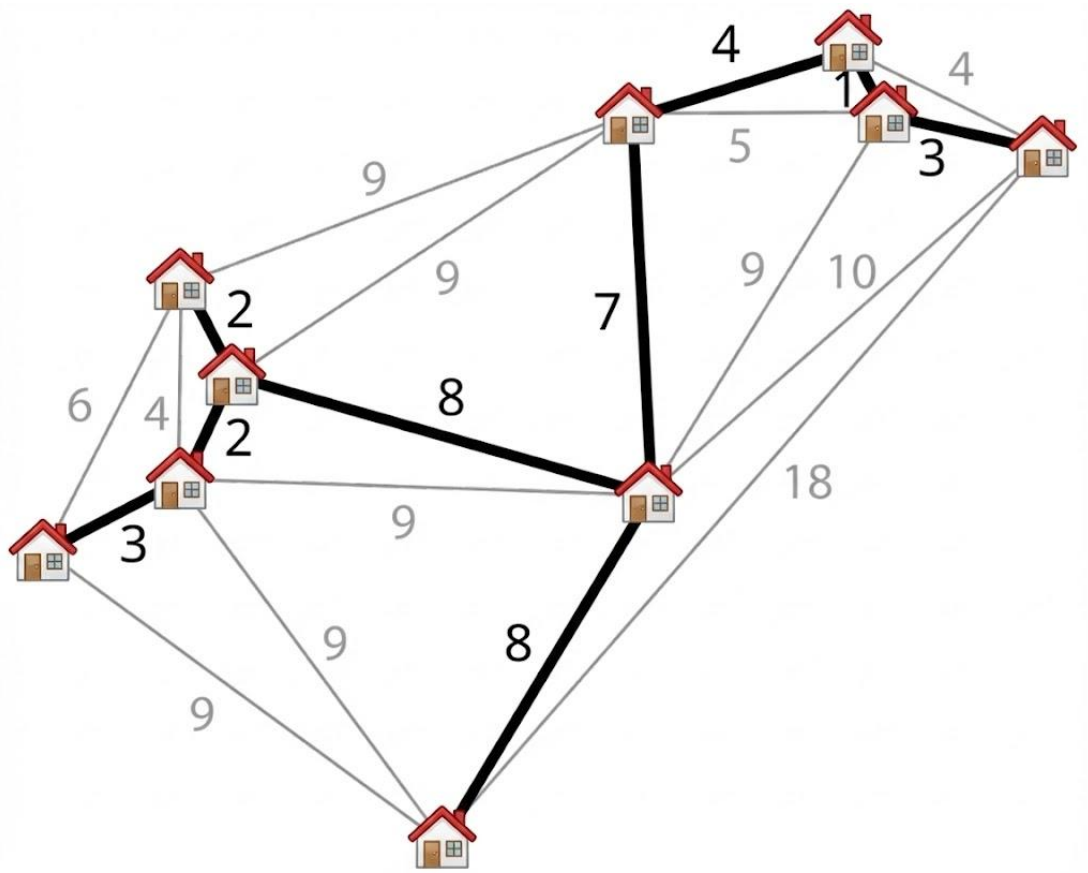
어느 정점도 떨어져 있으면 안 됩니다.

사이클 없음

사이클이 있으면 그중 간선 하나를 빼도 모두 연결된 채입니다. 모두 잇는 데 사이클은 필요 없습니다.

간선은 정확히 V − 1개

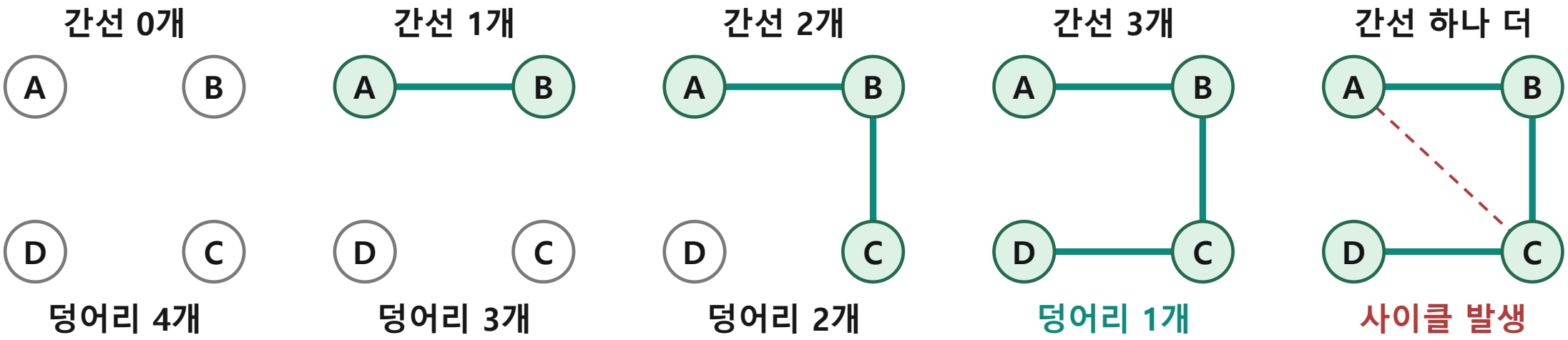
정점이 V개면 간선은 V − 1개입니다(다음 슬라이드).



굵은 선 9개 = 최소 신장 트리(정점 10개, 합 38). 집 = 정점, 선 = 간선, 수 = 가중치 (맨 위 짧은 굵은 선 = 1)

신장 트리의 간선은 V − 1개

덩어리 = 서로 경로로 이어진 정점들의 묶음입니다. 서로 다른 두 덩어리를 잇는 간선을 하나 더할 때마다 덩어리가 하나씩 줄어듭니다.



정점 V개를 한 덩어리로 이으려면 간선 V − 1개가 필요합니다.

간선을 하나 더 넣으면 이미 이어진 두 정점을 또 잇게 되어 덩어리는 줄지 않고 사이클만 생깁니다. 그래서 신장 트리의 간선은 정확히 V − 1개입니다.

크루스칼: 가장 싼 간선부터 잇는다

① 정렬
모든 간선을 가중치 오름차순으로 줄 세웁니다.

② 싼 간선부터 검사
사이클을 만들지 않으면 채택하고, 만들면 버립니다.

③ 멈춤
채택한 간선이 V − 1개가 되면 끝납니다.

사이클 판정: 간선의 두 끝이 이미 같은 덩어리이면 사이클

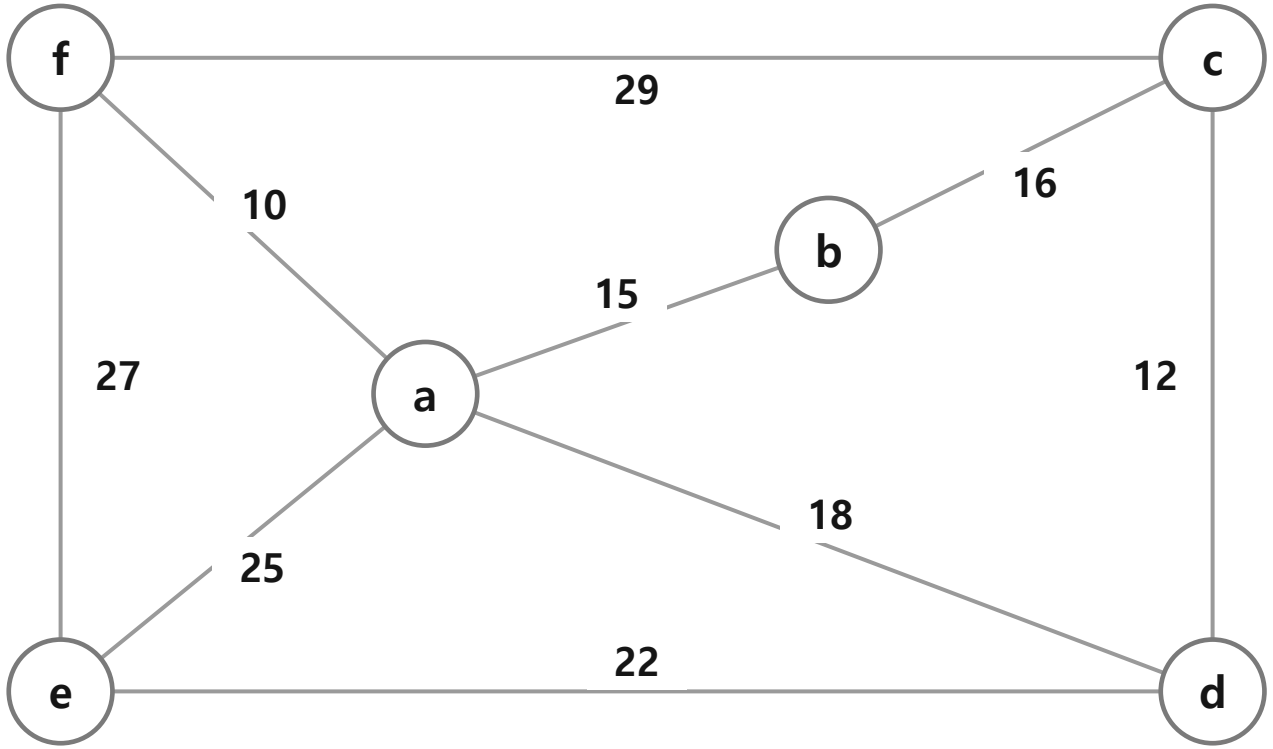
선택 = 남은 간선 중 가장 싼 것, 적정성 검사 = 사이클이 생기지 않는가, 해답 검사 = 간선이 V − 1개인가



왼쪽 그림: 가장 싼 간선부터 잇는다 오른쪽 그림: 이미 이어진 두 곳을 또 이으면 사이클이 되므로 버린다

예제: 정점 6개, 간선 9개

간선 9개를 가중치 오름차순으로 정렬했습니다.
코드에서는 a~f를 0~5번 정점으로 씁니다(a = 0, ..., f = 5).

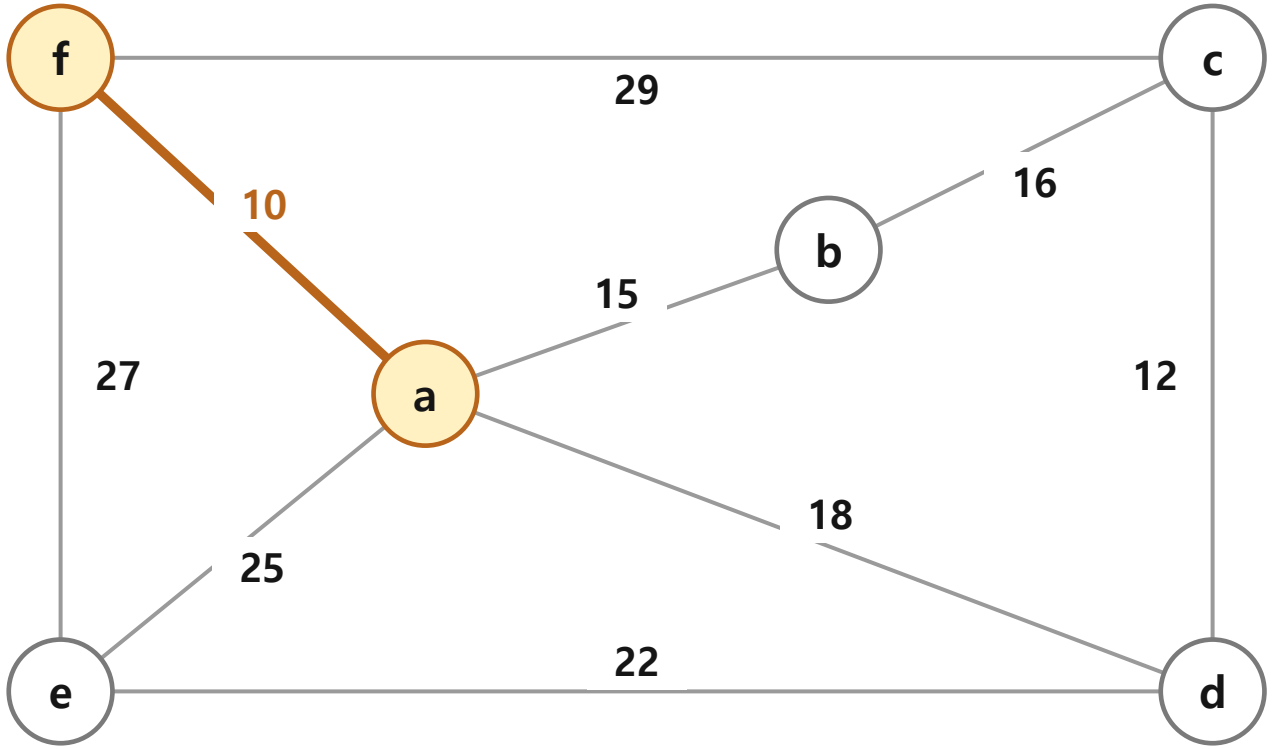


간선	가중치	결과
a – f	10	
c – d	12	
b – a	15	
b – c	16	
d – a	18	
d – e	22	
e – a	25	
e – f	27	
c – f	29	

선 위의 수: 가중치 오른쪽 표: 가중치 오름차순으로 정렬한 간선

① 10: a – f 채택

a와 f는 서로 다른 덩어리입니다 → 채택.
덩어리: {a, f} {b} {c} {d} {e} · 합 10

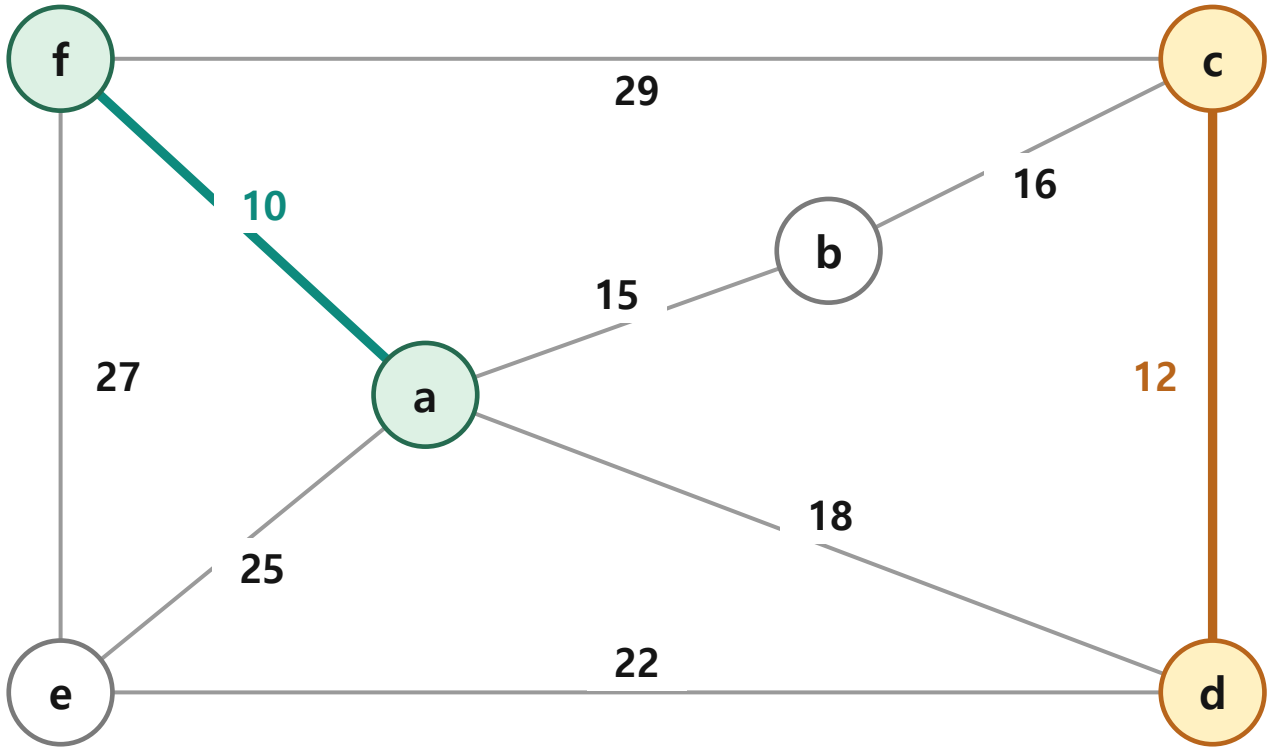


간선	가중치	결과
a – f	**10**	**채택**
c – d	12	
b – a	15	
b – c	16	
d – a	18	
d – e	22	
e – a	25	
e – f	27	
c – f	29	

청록: 채택한 간선 주황: 지금 보는 간선(노랑 정점: 그 두 끝) 빨강 점선: 버린 간선 초록 정점: 채택한 간선에 닿은 정점

② 12: c – d 채택

c와 d도 서로 다른 덩어리입니다 → 채택.
덩어리: {a, f} {c, d} {b} {e} · 합 22

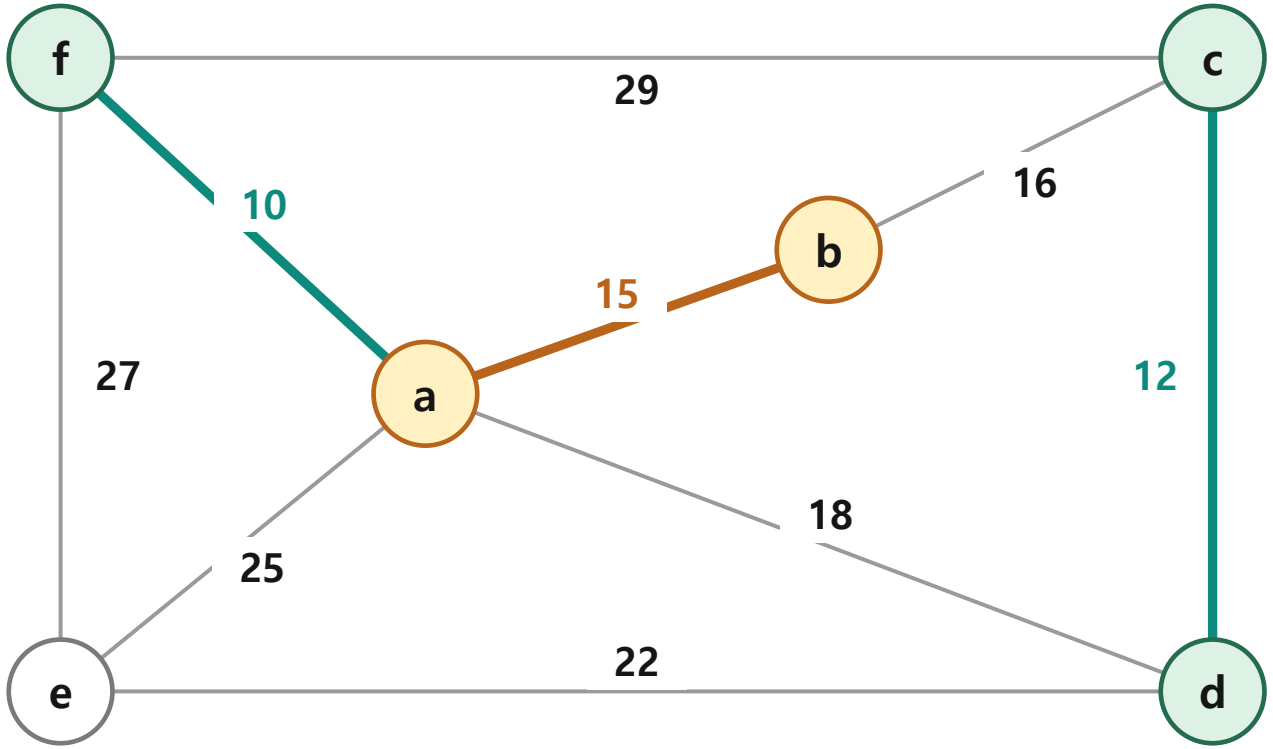


간선	가중치	결과
a – f	10	채택
c – d	**12**	채택
b – a	15	
b – c	16	
d – a	18	
d – e	22	
e – a	25	
e – f	27	
c – f	29	

청록: 채택한 간선 주황: 지금 보는 간선(노랑 정점: 그 두 끝) 빨강 점선: 버린 간선 초록 정점: 채택한 간선에 닿은 정점

③ 15: b – a 채택

b(혼자)와 a({a, f})는 서로 다른 덩어리입니다 → 채택.
덩어리: {a, b, f} {c, d} {e} · 합 37

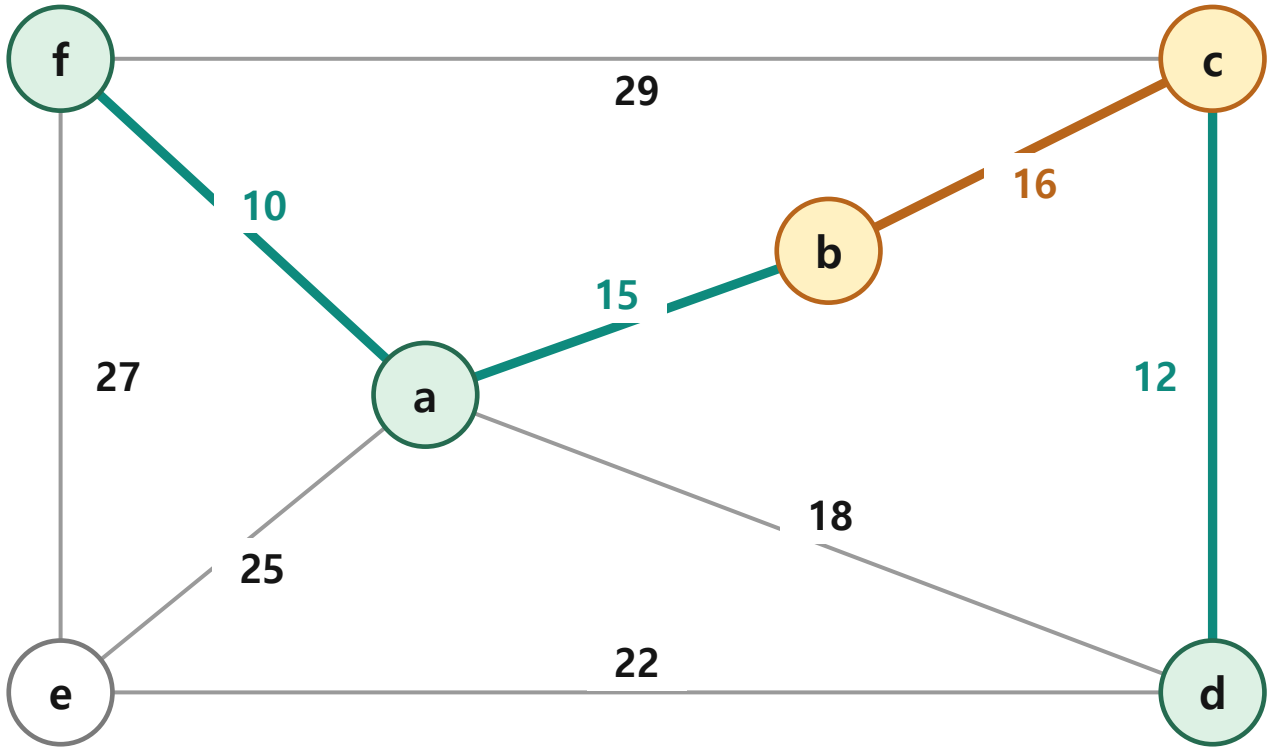


간선	가중치	결과
a – f	10	채택
c – d	12	채택
b – a	**15**	채택
b – c	16	
d – a	18	
d – e	22	
e – a	25	
e – f	27	
c – f	29	

청록: 채택한 간선 주황: 지금 보는 간선(노랑 정점: 그 두 끝) 빨강 점선: 버린 간선 초록 정점: 채택한 간선에 닿은 정점

④ 16: b – c 채택

b({a, b, f})와 c({c, d})는 서로 다른 덩어리입니다 → 채택.
덩어리: {a, b, c, d, f} {e} · 합 53

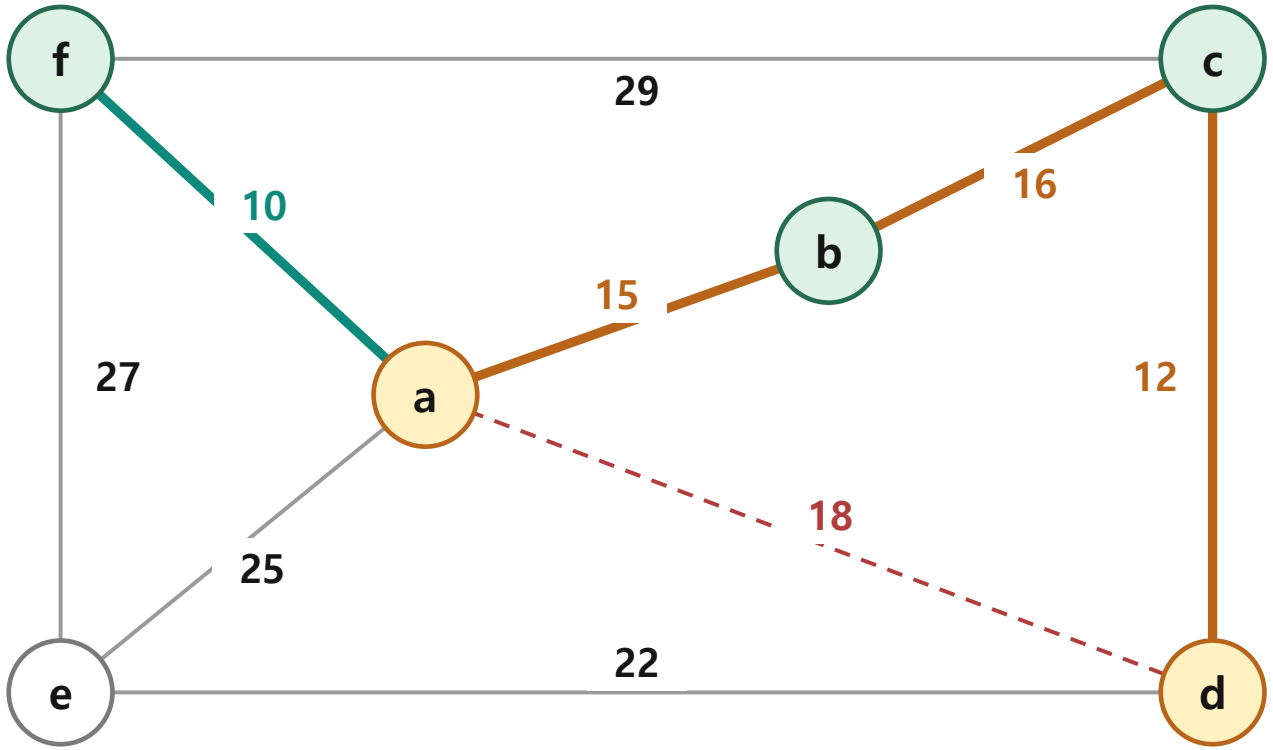


간선	가중치	결과
a – f	10	채택
c – d	12	채택
b – a	15	채택
b – c	**16**	채택
d – a	18	
d – e	22	
e – a	25	
e – f	27	
c – f	29	

청록: 채택한 간선 주황: 지금 보는 간선(노랑 정점: 그 두 끝) 빨강 점선: 버린 간선 초록 정점: 채택한 간선에 닿은 정점

⑤ 18: d – a 사이클 → 버림

d와 a는 이미 d – c – b – a로 이어져 있습니다.
d – a를 더하면 사이클이 생기므로 버립니다 · 합 53

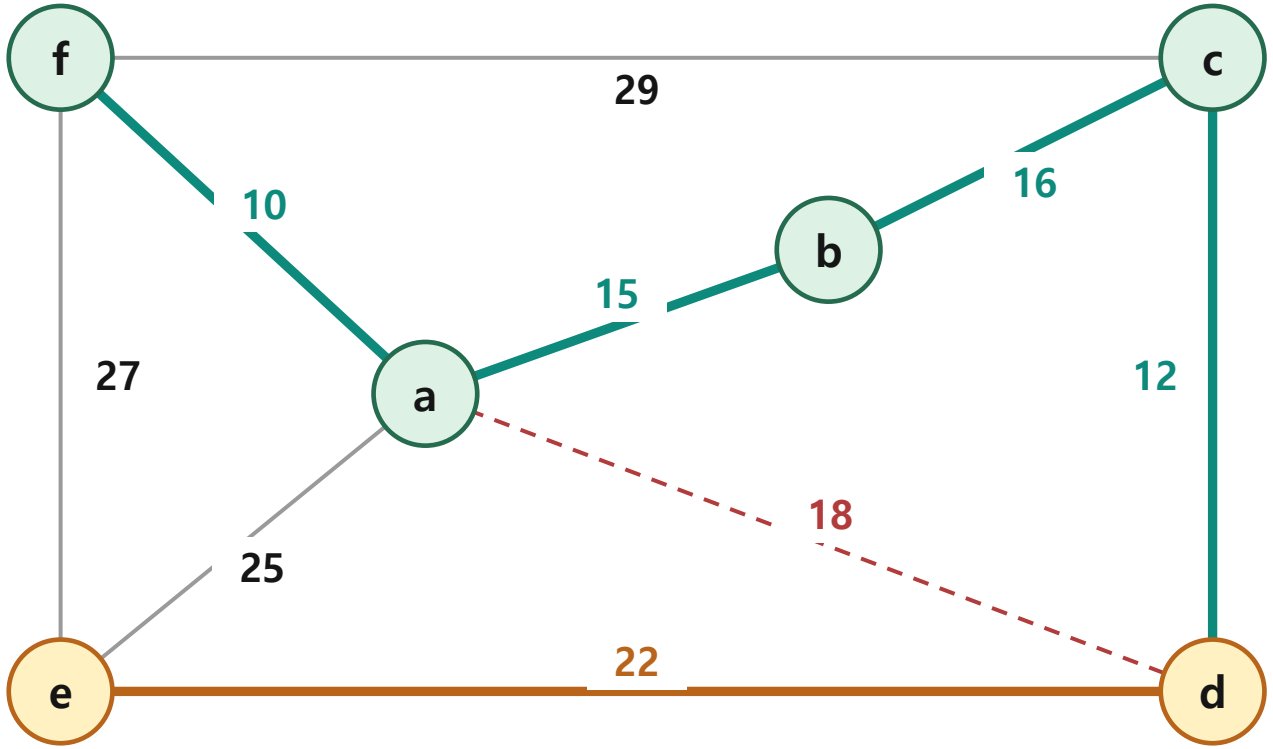


간선	가중치	결과
a – f	10	채택
c – d	12	채택
b – a	15	채택
b – c	16	채택
d – a	**18**	사이클
d – e	22	
e – a	25	
e – f	27	
c – f	29	

주황: d와 a를 이미 잇는 길 빨강 점선: 버린 d – a(노랑 정점: 그 두 끝) 청록: 채택한 간선 초록 정점: 채택한 간선에 닿은 정점

⑥ 22: d – e 채택 · 간선 5개

d와 e는 다른 덩어리 → 채택. 한 덩어리가 됩니다.
간선 5개(= V − 1)가 모였으므로 멈춥니다 · 합 75

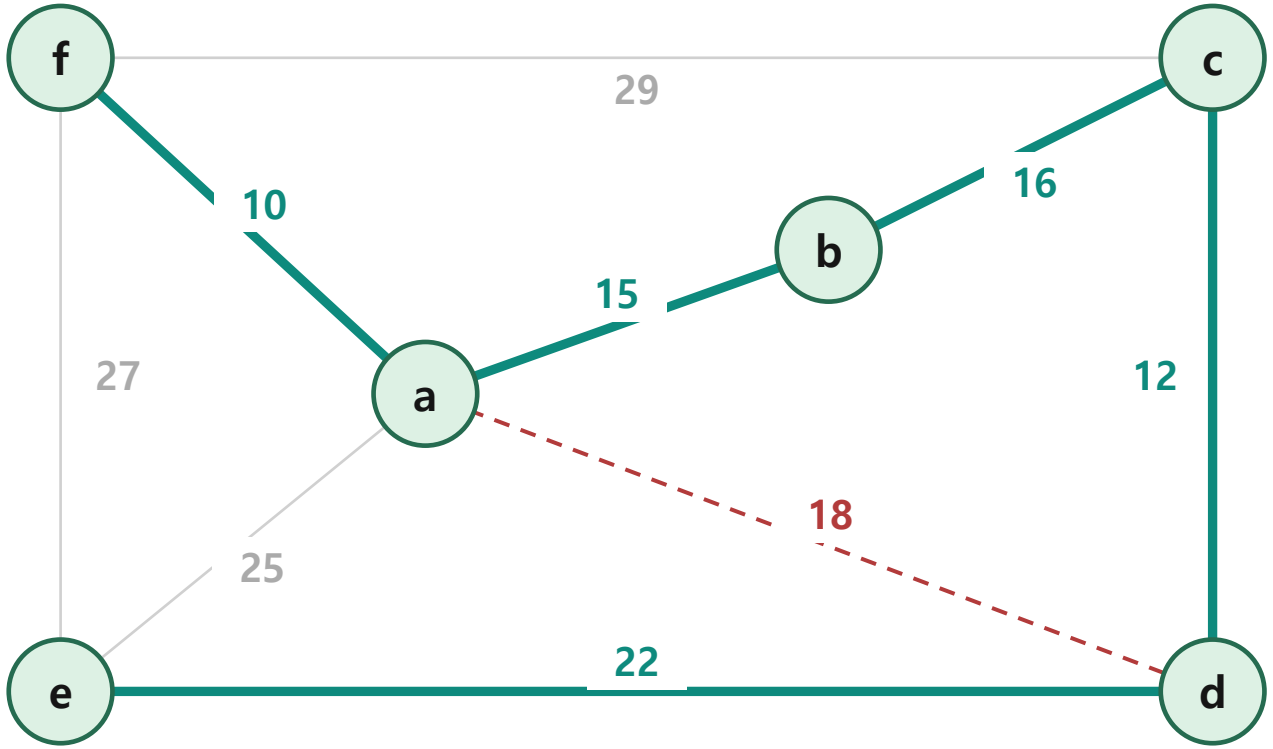


간선	가중치	결과
a – f	10	채택
c – d	12	채택
b – a	15	채택
b – c	16	채택
d – a	18	사이클
d – e	**22**	채택
e – a	25	확인하지 않음
e – f	27	확인하지 않음
c – f	29	확인하지 않음

청록: 채택한 간선 주황: 지금 보는 간선(노랑 정점: 그 두 끝) 빨강 점선: 버린 간선 초록 정점: 채택한 간선에 닿은 정점

⑦ 완료: 최소 신장 트리, 합 75

채택한 간선 5개: 10 + 12 + 15 + 16 + 22 = 75.
25, 27, 29는 확인하지 않았습니다.

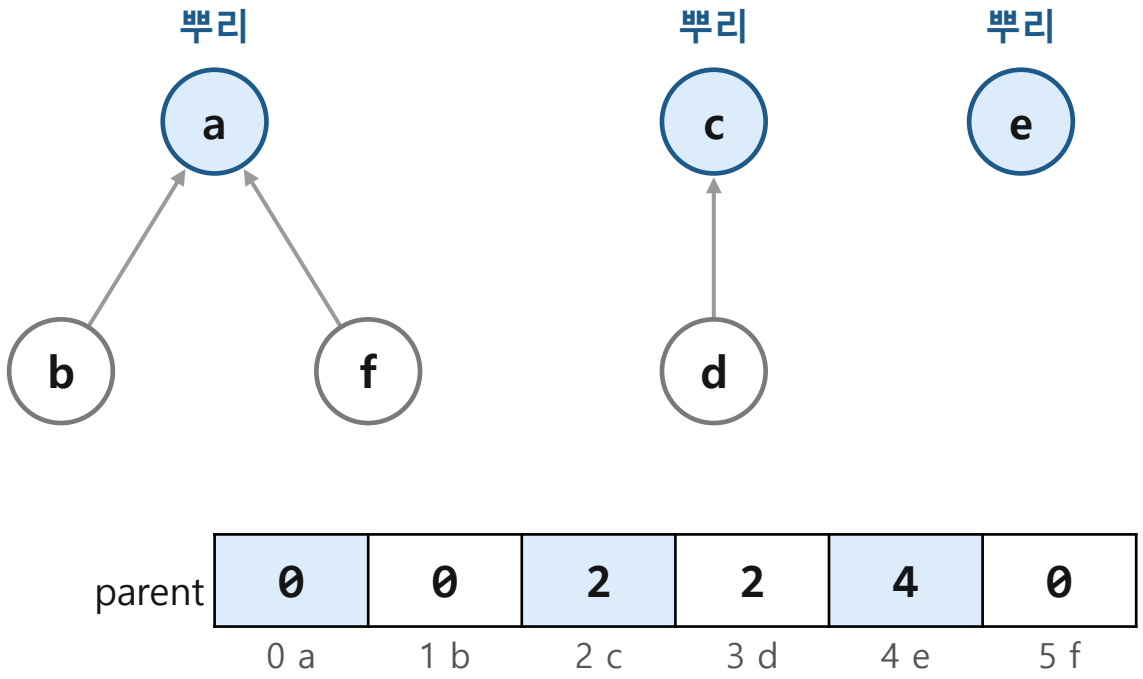


간선	가중치	결과
a – f	10	채택
c – d	12	채택
b – a	15	채택
b – c	16	채택
d – a	18	사이클
d – e	22	채택
e – a	25	확인하지 않음
e – f	27	확인하지 않음
c – f	29	확인하지 않음

청록: 최소 신장 트리의 간선 빨강 점선: 사이클이라 버린 간선 회색: 확인하지 않은 간선

서로소 집합: 덩어리 하나 = 나무 하나

덩어리를 나무(트리)로 저장해 "같은 덩어리인가?"에 빨리 답합니다. 그림은 ③ b – a를 채택한 직후입니다.



뿌리(루트) = 대표

parent[x] = x인 정점: a, c, e

find(x)

부모를 따라 뿌리까지 올라갑니다. 예: find(d) = c

union(x, y)

두 뿌리가 다르면 한 뿌리를 다른 뿌리 밑에 붙입니다.

사이클 판정

find(u) = find(v)이면 이미 같은 덩어리

화살표: 자식 → 부모 파랑: 뿌리(대표) 아래 칸: parent 배열 (칸 아래는 번호와 정점)

parent 배열: ① ~ ④ 채택

union은 두 뿌리 중 랭크(나무 높이의 상한)가 낮은 쪽을 높은 쪽 밑에 붙입니다. 랭크가 같으면 붙인 뒤 위쪽 랭크가 1 늘어납니다.

처음	**0**	**1**	**2**	**3**	**4**	**5**
	0 a	1 b	2 c	3 d	4 e	5 f
① a – f	**0**	**1**	**2**	**3**	**4**	**0**
② c – d	**0**	**1**	**2**	**2**	**4**	**0**
③ b – a	**0**	**0**	**2**	**2**	**4**	**0**
④ b – c	**0**	**0**	**0**	**2**	**4**	**0**

① union(a, f): 랭크 0 = 0 → f를 a 밑에, a의 랭크 1

② union(c, d): 같은 방식 → d를 c 밑에, c의 랭크 1

③ union(b, a): 랭크 0 < 1 → b를 a 밑에

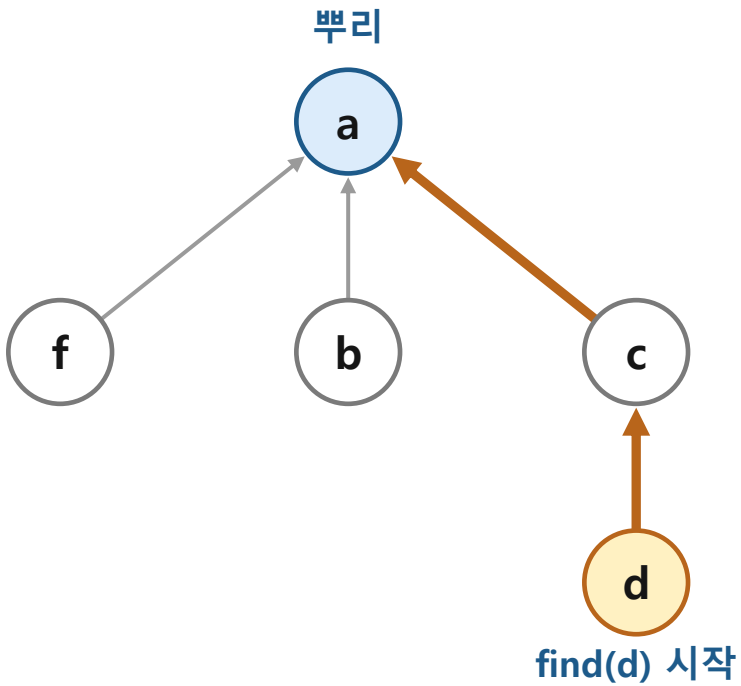
④ union(b, c): find(b) = a, find(c) = c. 랭크 1 = 1 → c를 a 밑에, a의 랭크 2

b와 c를 직접 잇지 않고, 두 뿌리 a와 c를 잇습니다.

파랑: 이번에 바뀐 칸 칸의 수: 부모의 번호 (a = 0, b = 1, c = 2, d = 3, e = 4, f = 5)

parent 배열: ⑤ 사이클 판정·경로 압축, ⑥ 채택

⑤ d – a: find(d)는 d → c → a를 따라 뿌리 a에 닿고, find(a) = a입니다. 뿌리가 같으므로 사이클입니다.



	0 a	1 b	2 c	3 d	4 e	5 f
④ 뒤	0	0	0	2	4	0
⑤ 뒤	0	0	0	0	4	0
⑥ 뒤	0	0	0	0	0	0

⑤ 경로 압축: find(d)가 지나간 d의 부모를 뿌리 a(0)로 바로 바꿉니다. 다음 find(d)는 한 번에 끝납니다.

⑥ d – e: find(d) = a, find(e) = e로 다르므로 채택. e(랭크 0)를 a(랭크 2) 밑에 붙입니다.

주황: find(d)가 따라간 길 노랑 칸: 바뀔 칸 파랑 칸: 바뀐 칸

의사코드: 간선 V − 1개가 되면 멈춘다

kruskal.pseudo

```
KRUSKAL(V, edges):          // V: 정점 수, edges: 간선 목록
    edges를 가중치 오름차순으로 정렬
    모든 정점 v에 대해 parent[v] <- v        // 각자 한 덩어리
    MST <- 빈 목록
    for each (w, u, v) in edges (정렬된 순서):
        if find(u) != find(v) then          // 서로 다른 덩어리
            MST에 (u, v, w) 추가;  union(u, v)
        if MST의 간선 수 = V - 1 then break   // 트리 완성
    return MST
```

① 선택

정렬된 순서에서 다음 간선(남은 것 중 가장 싼 간선)을 꺼냅니다.

② 적정성 검사

find(u) ≠ find(v)인가? 같으면 사이클이므로 건너뜁니다.

③ 해답 검사

채택한 간선이 V − 1개이면 멈춥니다.

Python ①: 서로소 집합 · 생성과 find

kruskal.py

```
class DisjointSet:
    def __init__(self, n):
        self.parent = list(range(n))  # 처음에는 모든 정점이 각자 집합
        self.rank = [0] * n           # 트리 높이의 상한

    def find(self, x):                # x가 속한 집합의 대표(루트)
        if self.parent[x] != x:
            self.parent[x] = self.find(self.parent[x])  # 경로 압축
        return self.parent[x]
```

Python ②: union (랭크로 합치기)

kruskal.py

```
def union(self, x, y):
    px, py = self.find(x), self.find(y)
    if px == py:
        return False                # 이미 같은 집합 -> 사이클
    if self.rank[px] < self.rank[py]:
        px, py = py, px             # px를 더 높은 트리로
    self.parent[py] = px            # 낮은 트리를 높은 트리 밑에
    if self.rank[px] == self.rank[py]:
        self.rank[px] += 1
    return True
```

Python ③: kruskal 함수

kruskal.py

```
def kruskal(n, edges):
    edges.sort()              # 가중치 오름차순 (튜플의 첫 값)
    ds = DisjointSet(n)
    mst = []
    for weight, u, v in edges:
        if ds.union(u, v):  # 사이클이 없을 때만 채택
            mst.append((u, v, weight))
        if len(mst) == n - 1:   # 간선 (정점 수 - 1)개면 완성
            break
    return mst
```

Python ④: 간선 목록과 실행

kruskal.py

```
names = "abcdef"              # 0~5번 정점의 이름
edges = [
    (10, 0, 5), (12, 2, 3), (15, 1, 0),
    (16, 1, 2), (18, 3, 0), (22, 3, 4),
    (25, 4, 0), (27, 4, 5), (29, 2, 5)
]
mst = kruskal(6, edges)
print("최소 신장 트리 구성:")
for u, v, w in mst:
    print(f"{names[u]} -- {names[v]} (가중치 {w})")
```

C ①: 간선 구조체와 make_sets

kruskal.c

```
#include <stdio.h>
#include <stdlib.h>

typedef struct { int w, u, v; } Edge;   /* (가중치, u, v) */
int parent[100];                        /* 정점 100개까지 */
int rank[100];                          /* 트리 높이의 상한 */

void make_sets(int n) {          /* 처음에는 모든 정점이 각자 집합 */
    for (int i = 0; i < n; i++) {
        parent[i] = i;
        rank[i] = 0;
    }
}
```

C ②: find와 union_sets

kruskal.c

```
int find(int x) {                /* x가 속한 집합의 대표(루트) */
    if (parent[x] != x)
        parent[x] = find(parent[x]);    /* 경로 압축 */
    return parent[x];
}

int union_sets(int x, int y) {  /* C에서 union은 예약어 */
    int px = find(x), py = find(y);
    if (px == py) return 0;      /* 이미 같은 집합 -> 사이클 */
    if (rank[px] < rank[py]) {  /* px를 더 높은 트리로 */
        int t = px; px = py; py = t;
    }
    parent[py] = px;             /* 낮은 트리를 높은 트리 밑에 */
    if (rank[px] == rank[py]) rank[px]++;
    return 1;
}
```

C ③: 비교 함수와 kruskal

kruskal.c

```
int by_weight(const void *a, const void *b) {
    const Edge *x = a, *y = b;
    return x->w - y->w;
}

int kruskal(int n, Edge edges[], int m, Edge mst[]) {
    qsort(edges, m, sizeof(Edge), by_weight);   /* 가중치 오름차순 */
    make_sets(n);
    int count = 0;
    for (int i = 0; i < m; i++) {
        if (union_sets(edges[i].u, edges[i].v))
            mst[count++] = edges[i];    /* 사이클이 없을 때만 채택 */
        if (count == n - 1) break;  /* 간선 (정점 수 - 1)개면 완성 */
    }
    return count;
}
```

C ④: main에서 실행

kruskal.c

```
int main(void) {
    const char *names = "abcdef";   /* 0~5번 정점의 이름 */
    Edge edges[] = {{10, 0, 5}, {12, 2, 3}, {15, 1, 0},
                    {16, 1, 2}, {18, 3, 0}, {22, 3, 4},
                    {25, 4, 0}, {27, 4, 5}, {29, 2, 5}};
    Edge mst[5];
    int k = kruskal(6, edges, 9, mst);  /* k = mst에 담긴 간선 수 */
    printf("최소 신장 트리 구성:\n");
    for (int i = 0; i < k; i++)
        printf("%c -- %c (가중치 %d)\n",
               names[mst[i].u], names[mst[i].v], mst[i].w);
    return 0;
}
```

실행 결과: 합 75

OUTPUT

```
최소 신장 트리 구성:
a -- f (가중치 10)
c -- d (가중치 12)
b -- a (가중치 15)
b -- c (가중치 16)
d -- e (가중치 22)
```

채택 순서가 단계 슬라이드와 같습니다. 합은 10 + 12 + 15 + 16 + 22 = 75입니다. Python과 C의 출력이 같습니다.

d – a(18)는 사이클이라 출력에 없습니다. 25, 27, 29는 간선 5개가 모인 뒤라 확인하지 않았습니다.

복잡도: O(E log E)

V는 정점 수, E는 간선 수입니다.

정렬

간선 E개를 가중치 순으로
O(E log E)

간선마다 find · union

경로 압축 + 랭크: 평균 O(α(V))
거의 상수 → E × O(α(V))

전체

O(E log E) + E × O(α(V))
= O(E log E)

정렬이 전체 비용을 결정합니다.

α(V)는 아주 느리게 자라는 함수(역 아커만 함수)입니다. 현실에서 쓰는 어떤 V에서도 4 이하라서 상수처럼 봅니다.

이 보장은 경로 압축과 랭크를 함께 쓸 때 성립합니다. 랭크로 합치면 나무 높이가 log V를 넘지 않습니다.

왜 옳은가: 교환 논증

크루스칼이 간선 g를 채택하는 순간, g는 서로 다른 덩어리를 잇는 간선 중 가장 쌉니다. 더 싼 간선은 이미 다 봤고, 본 간선은 모두 한 덩어리 안에 있기 때문입니다.

① 어떤 MST T를 생각합니다

지금까지 고른 간선을 모두 포함하는 MST T에 g가 없다고 합시다.

② T에 g를 더하면 사이클

그 사이클은 g의 한쪽 덩어리를 나갔다가 다시 들어와야 하므로, 서로 다른 덩어리를 잇는 다른 간선 h가 있습니다.

③ h를 빼고 g를 넣어도 손해가 없다

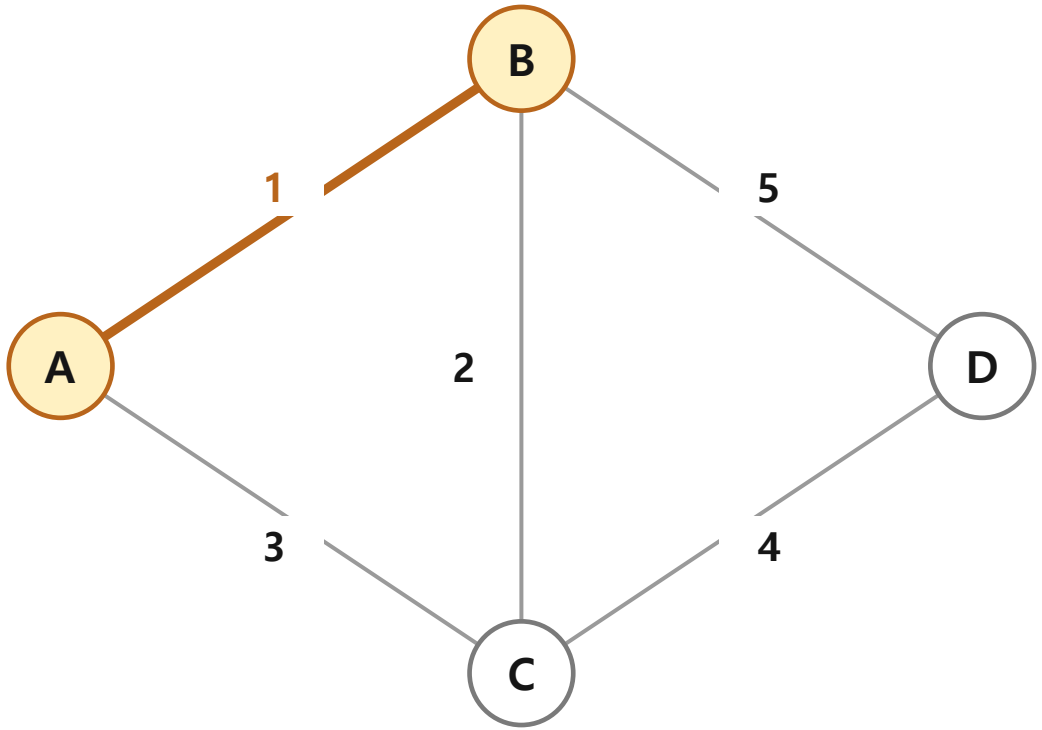
서로 다른 덩어리를 잇는 간선은 아직 보지 않은 간선이라 h의 가중치 ≥ g의 가중치입니다. 바꿔도 신장 트리이고 합은 늘지 않습니다.

→ 지금까지 고른 간선과 g를 모두 포함하는 MST가 있습니다.

예: ⑥에서 d – e(22)를 고를 때, 서로 다른 덩어리를 잇는 다른 간선 e – a(25), e – f(27)는 모두 22 이상입니다.

연습 ① A~D · AB 1 채택

간선을 정렬하면 AB 1, BC 2, AC 3, CD 4, BD 5입니다.
정점이 4개이므로 간선 3개(V − 1)에서 멈춥니다.



간선	가중치	결과
AB	**1**	**채택**
BC	2	
AC	3	
CD	4	
BD	5	

A와 B는 서로 다른 덩어리 → 채택

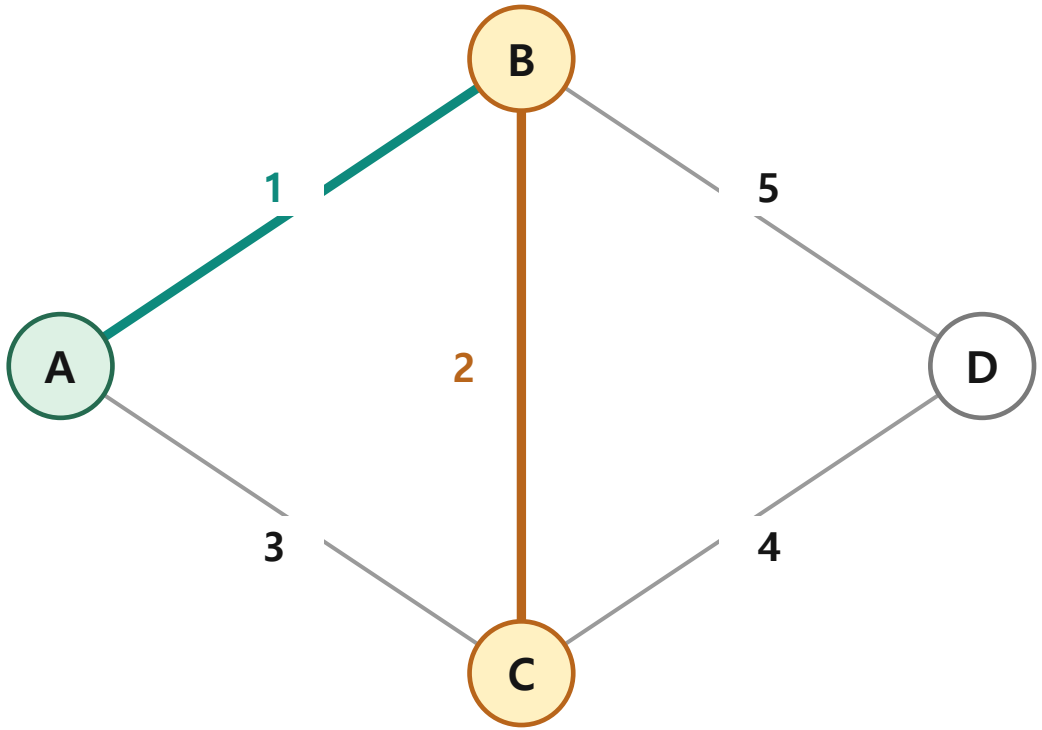
덩어리: {A, B} {C} {D}

합 1

청록: 채택한 간선 주황: 지금 보는 간선(노랑 정점: 그 두 끝) 빨강 점선: 버린 간선 초록 정점: 채택한 간선에 닿은 정점

연습 ② BC 2 채택

B({A, B})와 C(혼자)는 서로 다른 덩어리입니다 → 채택.
덩어리: {A, B, C} {D}



간선	가중치	결과
AB	1	채택
BC	**2**	채택
AC	3	
CD	4	
BD	5	

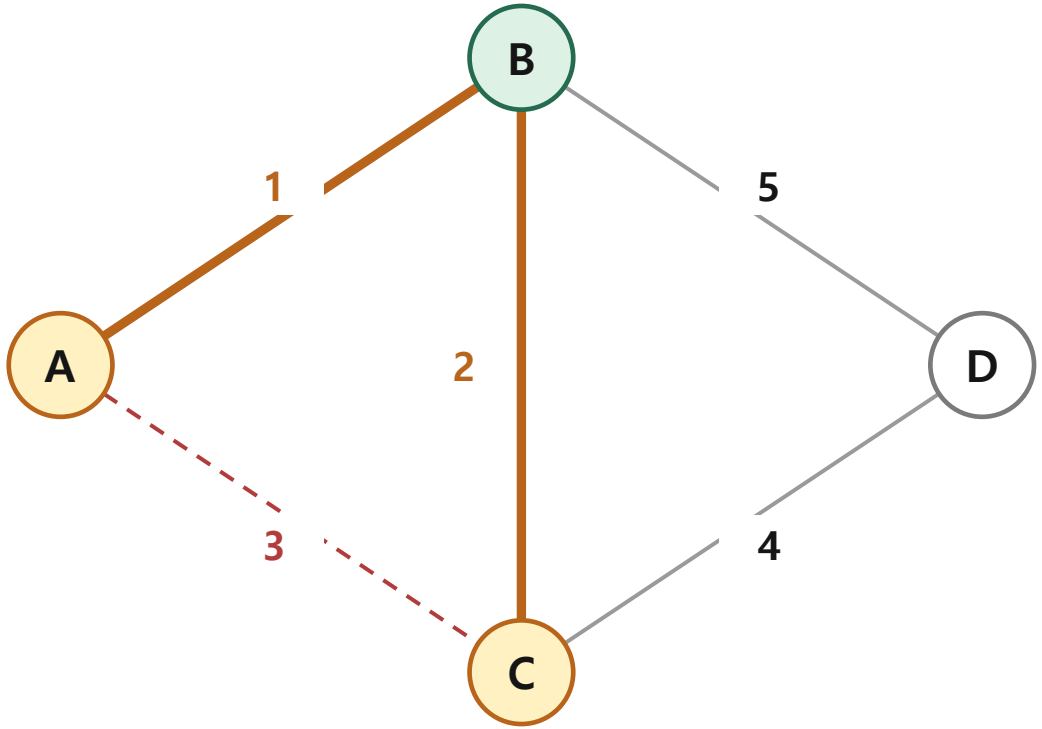
채택한 간선 2개

합 1 + 2 = 3

청록: 채택한 간선 주황: 지금 보는 간선(노랑 정점: 그 두 끝) 빨강 점선: 버린 간선 초록 정점: 채택한 간선에 닿은 정점

연습 ③ AC 3 사이클 → 버림

A와 C는 이미 A – B – C로 이어져 있습니다.
AC를 더하면 사이클 A – B – C – A가 생기므로 버립니다.



간선	가중치	결과
AB	1	채택
BC	2	채택
AC	**3**	사이클
CD	4	
BD	5	

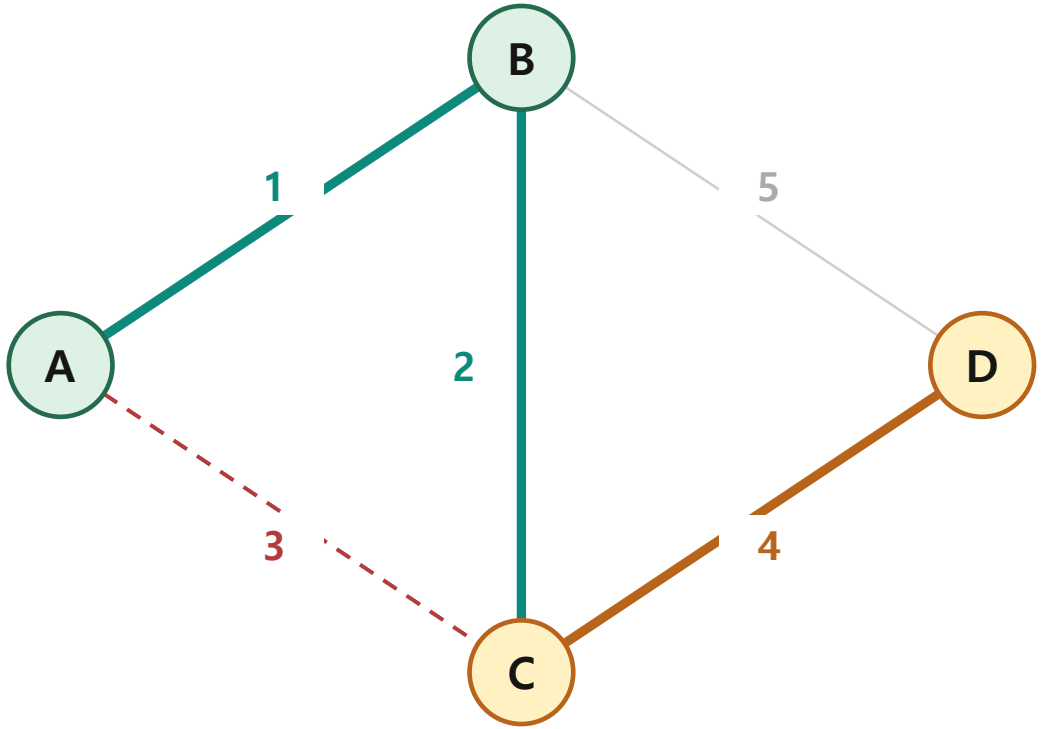
채택한 간선 그대로 2개

합 3

주황: A와 C를 이미 잇는 길 빨강 점선: 버린 AC(노랑 정점: 그 두 끝) 초록 정점: 채택한 간선에 닿은 정점

연습 ④ CD 4 채택 · 합 7

D가 연결되어 간선 3개(= V − 1)가 되었으므로 멈춥니다.
BD(5)는 확인하지 않습니다.



간선	가중치	결과
AB	1	채택
BC	2	채택
AC	3	사이클
CD	**4**	채택
BD	5	확인하지 않음

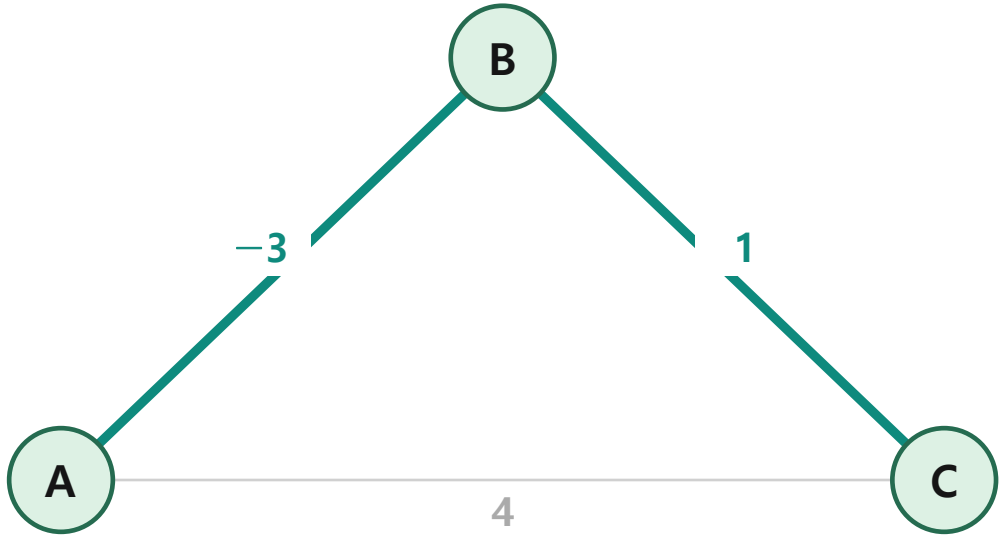
AB → BC → (AC 사이클) → CD → 멈춤

합 1 + 2 + 4 = 7

청록: 채택 주황: 지금 보는 간선(노랑 정점: 그 두 끝) 빨강 점선: 버림 회색: 확인하지 않음 초록 정점: 채택한 간선에 닿은 정점

연습: 음수 가중치 AB = −3

정점 A, B, C와 AB = −3, BC = 1, AC = 4.
정렬하면 AB(−3)가 맨 앞입니다.



간선	가중치	결과
AB	−3	① 채택
BC	1	② 채택
AC	4	확인하지 않음

간선 2개 = V − 1 → 멈춤

합 −3 + 1 = −2

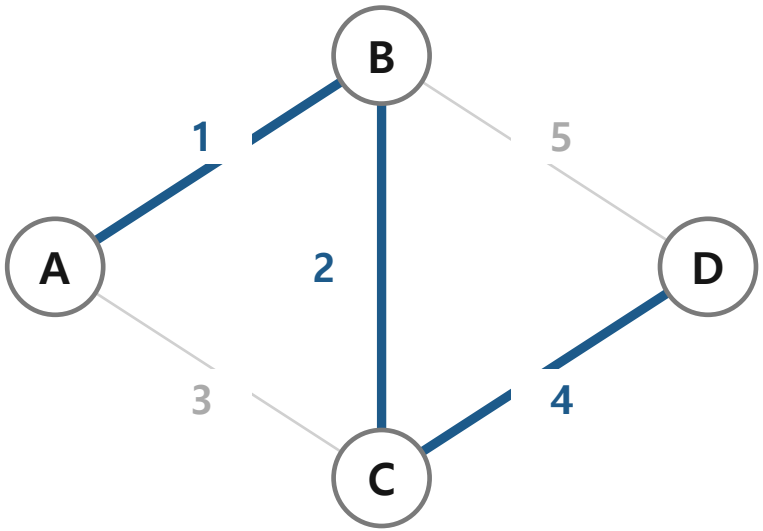
크루스칼은 가중치를 크기 비교에만 씁니다. 음수여도 그대로 성립합니다. 음수 간선에서 틀릴 수 있는 다익스트라(5.8)와 다릅니다.

청록: 채택한 간선 회색: 확인하지 않은 간선

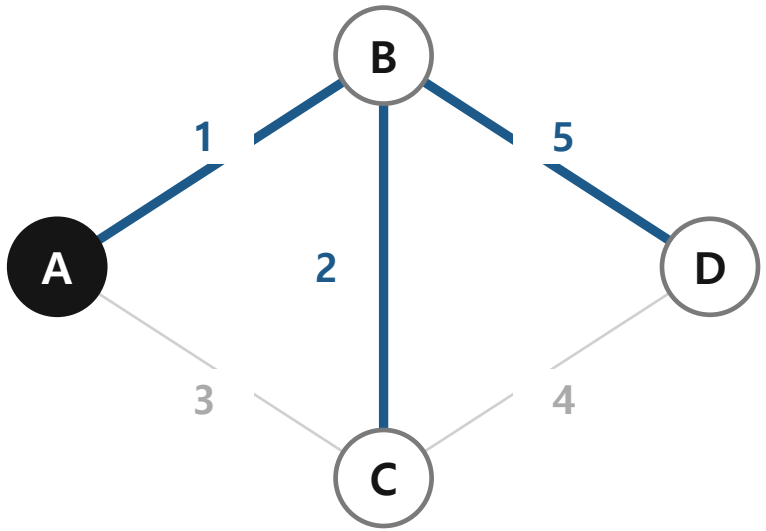
MST와 최단 경로 트리는 다를 수 있다

같은 A~D 그래프입니다. MST는 간선 가중치의 합을,
최단 경로 트리는 시작점 A에서 각 정점까지의 거리를 최소로 합니다.

최소 신장 트리: 합 7



A에서의 최단 경로 트리: 합 8



A에서 각 정점까지 거리

정점	MST 안	최단
B	1	1
C	3	3
D	**7**	**6**

D까지: MST 안의 길 A – B – C – D = 7, 최단 경로 A – B – D = 6

A – C를 넣은 트리(합 9)도 최단 경로 트리입니다(A – C 직행도 3).

검정: 거리를 재는 시작점 A(MST에는 시작점이 없습니다) 파랑: 트리에 넣은 간선 회색: 넣지 않은 간선

확인: 크루스칼 알고리즘

① 정점이 7개인 그래프에서 신장 트리의 간선은 몇 개입니까?

② 정점 P, Q, R, S와 간선 PQ = 2, QR = 3, PR = 4, RS = 1, QS = 5. 크루스칼이 채택하는 간선과 합은? 확인하지 않는 간선은?

③ parent = [0, 0, 1, 2]일 때 find(3)의 결과는? 경로 압축 뒤 parent는?

④ "MST에서 두 정점 사이의 길은 그 두 정점 사이의 최단 경로다." 맞습니까?

5.7

TSP와 그리디

"일단 지금 있는 곳에서 가장 가까운 도시로!"
가장 단순한 그리디 기준이 어디까지 통할까요?

외판원 순회 문제(TSP)

모든 도시를 **한 번씩** 방문하고 출발 도시로 **돌아오는** 가장 짧은 순회를 찾습니다.

순회(tour)
출발 도시로 돌아오는 닫힌 경로

비용
지나간 간선 거리의 합(양방향이 같음)

경우의 수(출발 도시 고정)
(n−1)!가지: 4개면 6, 11개면 3,628,800

TSP는 NP-난해 문제입니다

NP-난해: 다항 시간에 풀면 NP의 모든 문제를 다항 시간에 풀 수 있게 되는, 가장 어려운 부류의 문제. 다항 시간 방법은 아직 알려져 있지 않습니다.

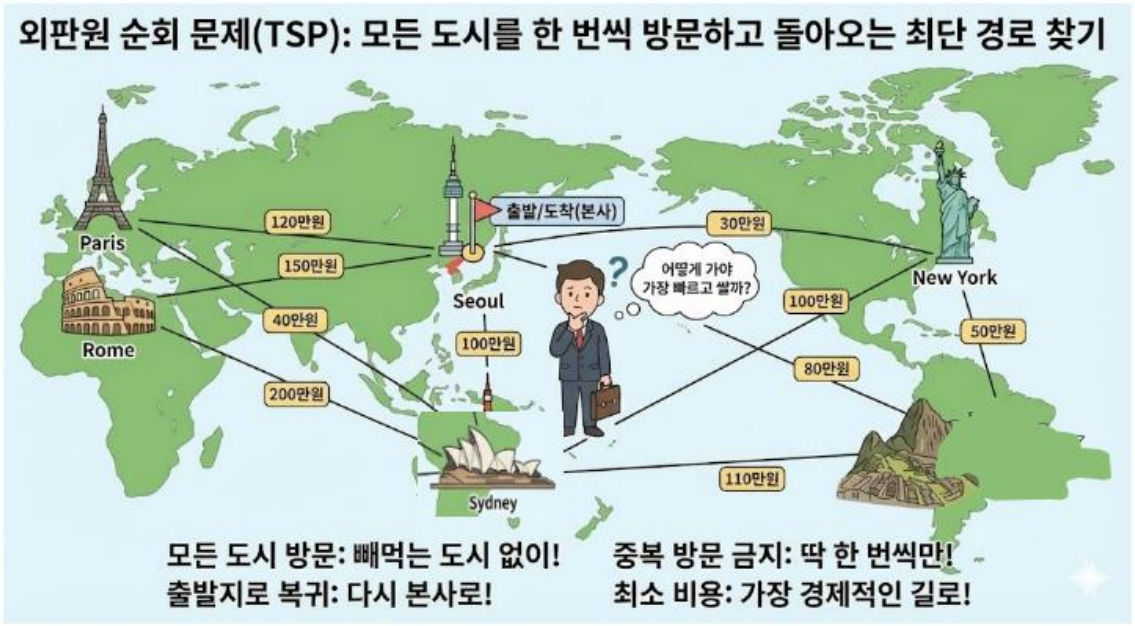


그림: 서울 본사에서 출발해 모든 도시를 한 번씩 들르고 서울로 돌아오는 외판원

가장 가까운 이웃(nearest neighbor)

지금 있는 도시에서 **아직 가지 않은 도시 중 가장 가까운 곳**으로 갑니다. 모든 도시를 돌았으면 출발 도시로 돌아옵니다.

① 선택 — 방문하지 않은 도시 중 current에서 거리가 가장 짧은 도시

② 적정성 검사 — 이미 방문한 도시(visited)는 다시 고르지 않습니다

③ 해답 검사 — n개를 모두 방문하면 출발 도시로 돌아가 끝냅니다

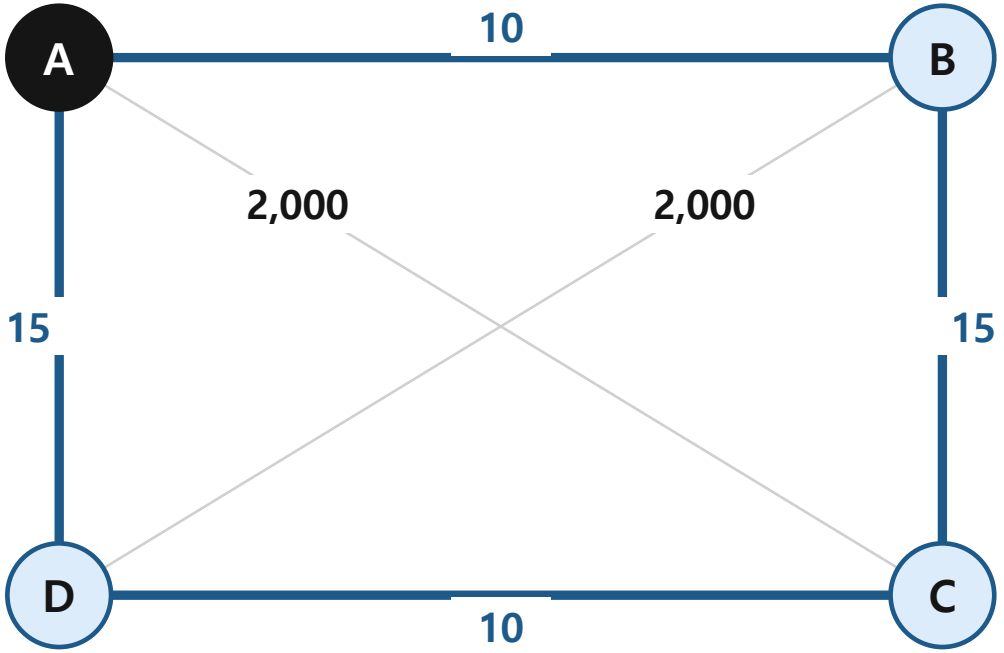
nearest_neighbor.pseudo

```
NEAREST_NEIGHBOR(adj, start):
    visited <- {start};  path <- [start];  current <- start;  total <- 0
    n - 1번 반복:
        next_city <- visited에 없는 i 중 adj[current][i]가 가장 작은 i
        visited와 path에 next_city 추가;  total <- total + adj[current][next_city];  current <- next_city
    return path + [start],  total + adj[current][start]        // 출발 도시로 복귀
```

adj[i][j]: 도시 i와 j 사이의 거리를 담은 표(인접 행렬)

성공 예: A → B → C → D → A = 50

A에서 출발해 매번 가장 가까운 도시로 갑니다. 이 그래프에서는 그리디의 순회가 최적입니다.



검정: 출발 도시 파랑 굵은 선: 순회에 쓴 길 연회색: 쓰지 않은 길

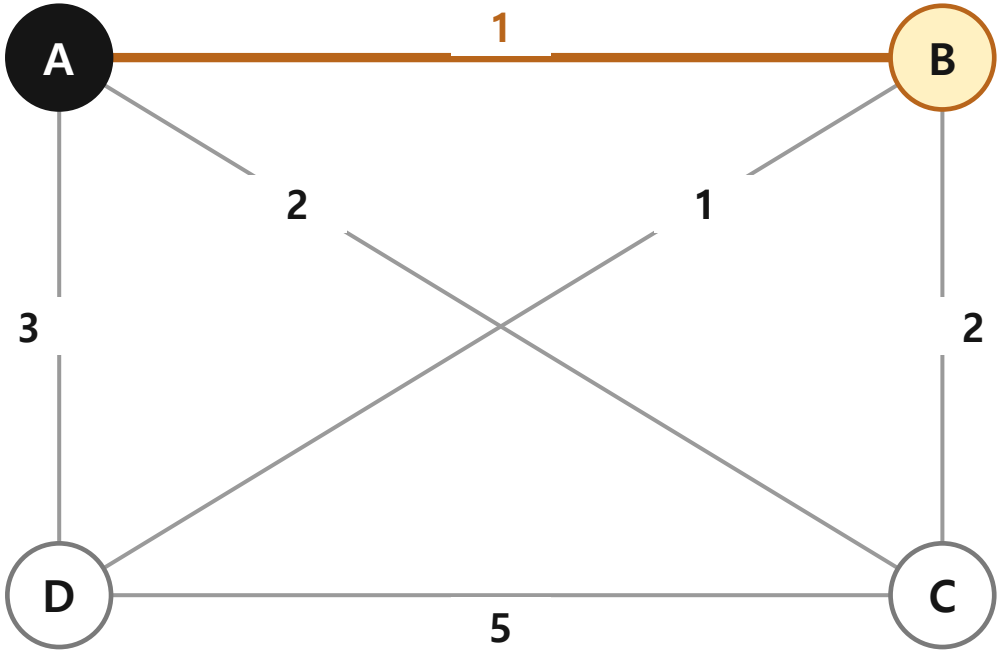
지금	후보와 거리	이동
A	B 10 · C 2,000 · D 15	→ B
B	C 15 · D 2,000	→ C
C	D 10	→ D
D	모두 방문함	→ A 15

10 + 15 + 10 + 15 = 50

방향만 다른 것을 하나로 세면 순회는 세 가지뿐: 50, 4,020, 4,030 → 50이 최적

실패 예 ① A에서 가장 가까운 B(1)

거리만 바꾼 그래프입니다. A에서 B 1, C 2, D 3을 비교해
가장 가까운 B로 갑니다.



A에서 갈 수 있는 도시

도시	거리
B	1
C	2
D	3

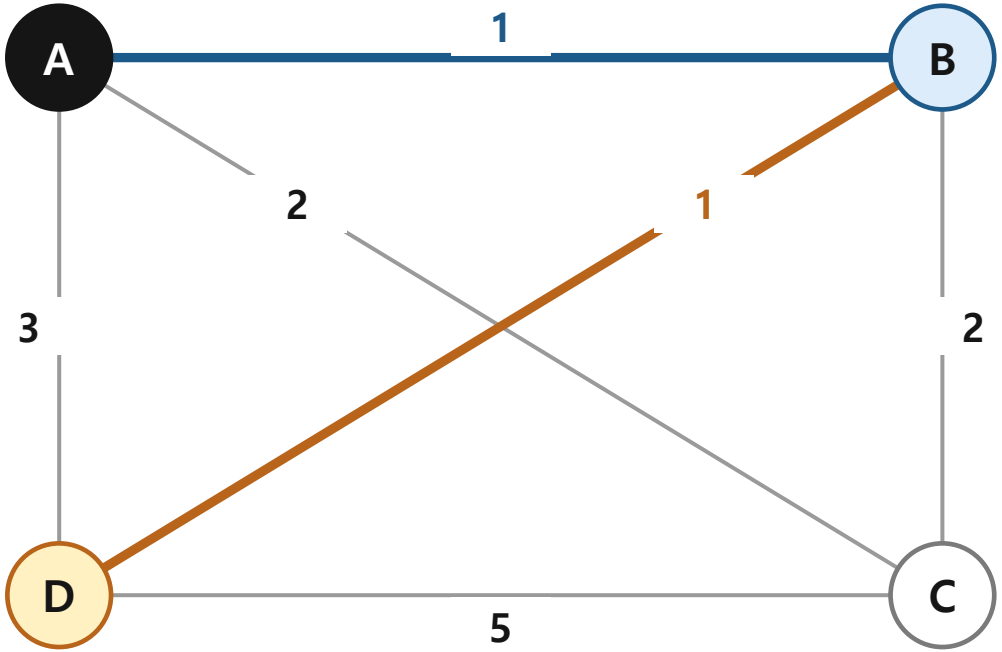
→ B로 이동

경로 A → B, 지금까지 1

검정: 출발 노랑: 방금 도착 파랑 원: 지나온 도시 파랑 굵은 선: 지나온 길 주황: 이번에 고른 길

실패 예 ② B에서 가장 가까운 D(1)

B에서 아직 가지 않은 도시는 C와 D입니다. C 2, D 1을 비교해 D로 갑니다.



B에서 갈 수 있는 도시(A는 방문함)

도시	거리
C	2
D	1

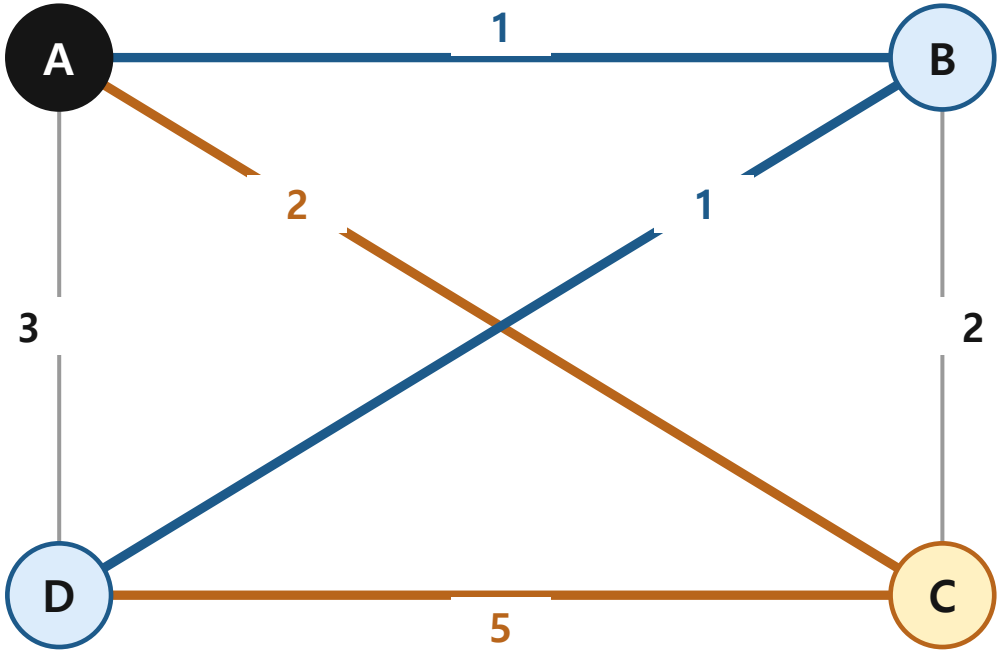
→ D로 이동

경로 A → B → D, 지금까지 1 + 1 = 2

검정: 출발 노랑: 방금 도착 파랑 원: 지나온 도시 파랑 굵은 선: 지나온 길 주황: 이번에 고른 길

실패 예 ③ 남은 C(5), 복귀 A(2) · 합 9

D에서 남은 도시는 C뿐이라 가장 비싼 5를 피할 수 없습니다. C에서 A로 돌아오면 합은 9입니다.



이동	비용
A → B	1
B → D	1
D → C	5
C → A (복귀)	2
합	**9**

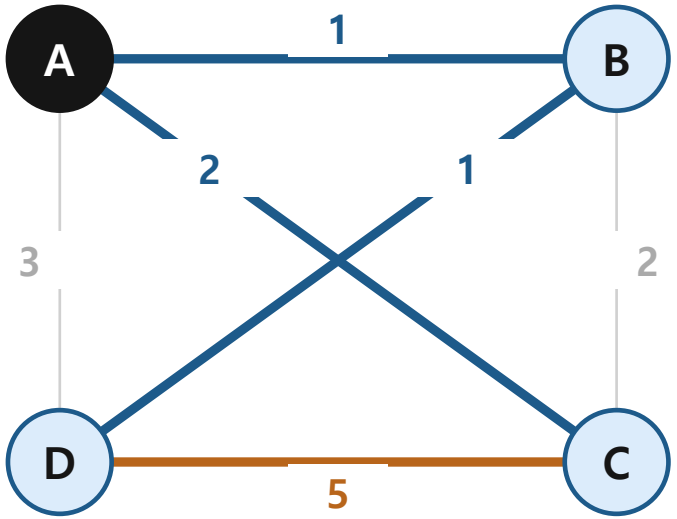
D → C는 고른 길이 아니라, 남은 도시가 C 하나뿐이라 가야만 하는 길입니다.

검정: 출발 노랑: 방금 도착 파랑 원: 지나온 도시 파랑 굵은 선: 지나온 길 주황: 이번에 간 길(D → C → A)

실패 예 ④ 그리디 9 vs 최적 8

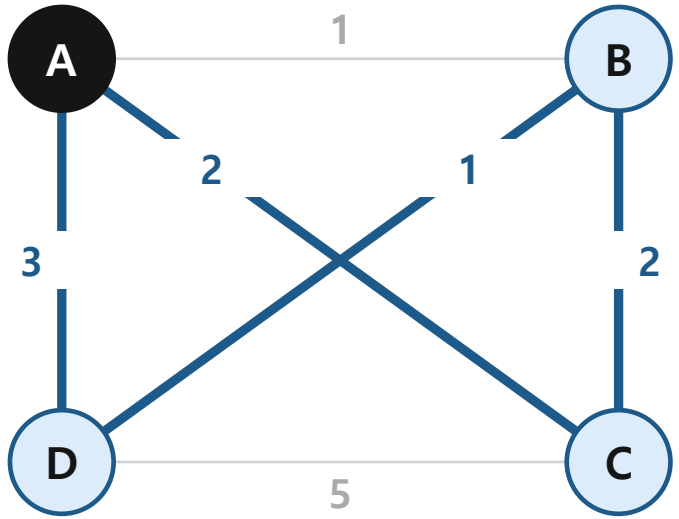
순회는 방향만 다른 것을 하나로 세면 세 가지뿐입니다(11, 9, 8). 최적은 A → C → B → D → A, 비용 8입니다.

가장 가까운 이웃: 9



1 + 1 + 5 + 2 = 9

최적: 8



2 + 2 + 1 + 3 = 8

파랑 굵은 선: 순회에 쓴 길 주황: 그리디가 피하지 못한 D → C(5) 연회색: 쓰지 않은 길

왜 실패하나: A → B를 포함하는 최적해가 없다

A에서 출발하는 순회	비용	A–B를 쓰는가
A → B → C → D → A	1 + 2 + 5 + 3 = 11	예
A → B → D → C → A (그리디)	1 + 1 + 5 + 2 = 9	예
A → C → B → D → A (최적)	**2 + 2 + 1 + 3 = 8**	**아니오**

조건 ①(탐욕적 선택 속성)이 깨졌습니다

첫 선택 A → B를 포함하는 순회는 11과 9뿐, 최적 8은 A–B를 쓰지 않습니다.

선택할 때 남을 도시를 보지 않습니다

비싼 D → C(5)는 앞의 두 선택이 남긴 결과입니다. 복귀 C → A(2)는 오히려 쌉니다.

정확한 답은 6장에서

두 도시 사이 최단 경로로 쪼갤 수는 없지만, 방문한 도시 집합까지 기억하면 동적 계획법으로 풉니다(그래도 도시 수에 대해 지수 시간).

가장 가까운 이웃은 최적해를 보장하지 못합니다

"항상 최적이다"를 반박하는 데는 이 반례 하나면 충분합니다.

Python: 가장 가까운 이웃 ① 준비

tsp.py

```
INF = float("inf")

def solve_tsp_nearest_neighbor(adj, start=0):
    n = len(adj)
    visited = [False] * n
    visited[start] = True
    path = [start]
    current, total = start, 0
```

Python: 가장 가까운 이웃 ② 반복과 복귀

tsp.py

```
    for _ in range(n - 1):
        min_dist, next_city = INF, -1
        # 방문 안 한 도시 중 가장 가까운 곳 찾기
        for i in range(n):
            if not visited[i] and adj[current][i] < min_dist:
                min_dist, next_city = adj[current][i], i
        visited[next_city] = True
        path.append(next_city)
        total += min_dist
        current = next_city

    total += adj[current][start]   # 출발점으로 복귀
    path.append(start)
    return path, total
```

Python: 가장 가까운 이웃 ③ 실행

tsp.py

```
names = "ABCD"
matrix = [
    [0,  10, 15, 20],   # A
    [10, 0,  35, 25],   # B
    [15, 35, 0,  30],   # C
    [20, 25, 30, 0]     # D
]
for start in [0, 3]:    # A에서, D에서 출발
    path, cost = solve_tsp_nearest_neighbor(matrix, start)
    print("경로:", " -> ".join(names[c] for c in path))
    print("비용:", cost)
```

앞의 실패 예와 다른 거리 표입니다(순회 80, 95, 95).

C: 가장 가까운 이웃 ① 준비

tsp.c

```
#include <limits.h>
#include <stdio.h>
#define N 4                          /* 도시 수 */

/* 방문 순서를 path에 담고 총비용을 돌려준다 */
int solve_tsp_nearest_neighbor(int adj[][N], int start, int path[]) {
    int visited[N] = {0};
    visited[start] = 1;
    path[0] = start;
    int current = start, total = 0;
```

C: 가장 가까운 이웃 ② 반복과 복귀

tsp.c

```
    for (int k = 1; k < N; k++) {
        int min_dist = INT_MAX, next_city = -1;
        /* 방문 안 한 도시 중 가장 가까운 곳 찾기 */
        for (int i = 0; i < N; i++)
            if (!visited[i] && adj[current][i] < min_dist) {
                min_dist = adj[current][i];
                next_city = i;
            }
        visited[next_city] = 1;
        path[k] = next_city;
        total += min_dist;
        current = next_city;
    }

    total += adj[current][start];   /* 출발점으로 복귀 */
    path[N] = start;
    return total;
}
```

C: 가장 가까운 이웃 ③ 실행

tsp.c

```
int main(void) {
    const char *names = "ABCD";
    int matrix[N][N] = {
        {0,  10, 15, 20},       /* A */
        {10, 0,  35, 25},       /* B */
        {15, 35, 0,  30},       /* C */
        {20, 25, 30, 0}         /* D */
    };
    int starts[] = {0, 3};      /* A에서, D에서 출발 */
    for (int s = 0; s < 2; s++) {
        int start = starts[s];
        int path[N + 1];
        int cost = solve_tsp_nearest_neighbor(matrix, start, path);
        printf("경로: %c", names[path[0]]);
        for (int i = 1; i <= N; i++)
            printf(" -> %c", names[path[i]]);
        printf("\n비용: %d\n", cost);
    }
    return 0;
}
```

실행 결과: 출발 도시에 따라 80과 95

OUTPUT

```
경로: A -> B -> D -> C -> A
비용: 80
경로: D -> A -> B -> C -> D
비용: 95
```

출발 도시	가장 가까운 이웃	최적(세 순회 80, 95, 95 중 최소)
A	80	**80 = 최적**
D	95	**최적 80, 그리디가 15 더 깁니다**

시간 복잡도 $O(n^2)$: 바깥 반복 $n-1$번 × 안쪽에서 도시 n개 훑기

빠르게 그럴듯한 답을 내는 경험적 방법(휴리스틱)입니다. 결과가 출발 도시에 따라 달라지고, 최적과 얼마나 차이 날지도 보장하지 못합니다.

TSP 확인

실패 예 그래프의 거리입니다(행 = 출발, 열 = 도착).

	A	B	C	D
A	0	1	2	3
B	1	0	2	1
C	2	2	0	5
D	3	1	5	0

① D에서 출발하면 가장 가까운 이웃의 경로와 비용은? 최적 8과 비교하세요.

② 첫 선택 A → B가 안전하지 않다는 것을 어떻게 보이나요?

③ "가장 가까운 이웃은 항상 최적이다"를 반박하려면 무엇이 필요한가요?

④ 도시가 n개일 때 시간 복잡도는? 마지막에 따로 더하는 비용은?

5.8

다익스트라 알고리즘

"아직 확정하지 않은 곳 중 시작점에서 가장 가까운 곳까지의 거리는 더 줄어들지 않는다." — 간선 비용이 모두 0 이상일 때

한 시작점에서 모든 정점까지의 최단 거리

시작점 하나에서 **다른 모든 정점까지** 가장 짧은 경로의 길이를 구합니다.

dist[v]

시작점에서 v까지 지금까지 찾은 가장 짧은 거리.
처음엔 시작점 0, 나머지 ∞

확정

dist[v]가 더 줄어들지 않는다고 결론 내리는 것

다익스트라의 선택

아직 확정하지 않은 정점 중 dist가 가장 작은 곳부터 확정합니다

내비게이션 길 찾기, 네트워크 라우팅(예: OSPF)에 다익스트라나 그 변형이 쓰입니다.



그림: 집에서 편의점 2분, 학교 10분, 병원 15분. 아직 확정하지 않은 곳 중 가장 가까운 편의점부터 확정합니다.

방향 그래프와 세 가지 표현

간선에 방향이 있으면 **방향 그래프**입니다. A → B 8은 A에서 B로만 8에 갈 수 있다는 뜻입니다. 크루스칼과 TSP의 그래프는 방향이 없었습니다.

표현	모양	이 장에서
간선 목록	(가중치, u, v)를 모은 리스트	크루스칼: 간선을 가중치 순으로 정렬
인접 행렬	adj[u][v] = u에서 v로 가는 비용	TSP: 모든 도시 쌍의 거리
인접 리스트	정점마다 {이웃: 비용}	다익스트라 Python: 나가는 간선만 저장

dijkstra.py의 그래프(인접 리스트)

```
my_graph = {
    "A": {"B": 8, "C": 1, "D": 2},
    "B": {}, "C": {"B": 5, "D": 2},
    "D": {"E": 3, "F": 5},
    "E": {"F": 1}, "F": {"A": 5}
}
```

C 다익스트라는 같은 그래프를 인접 행렬 graph[u][v]로 쓰고, 0이면 간선이 없다는 뜻으로 둡니다.

가장 가까운 이웃과 무엇이 다른가

둘 다 "가장 가까운 곳"을 고르지만, 무엇으로 거리를 재는지가 다릅니다.

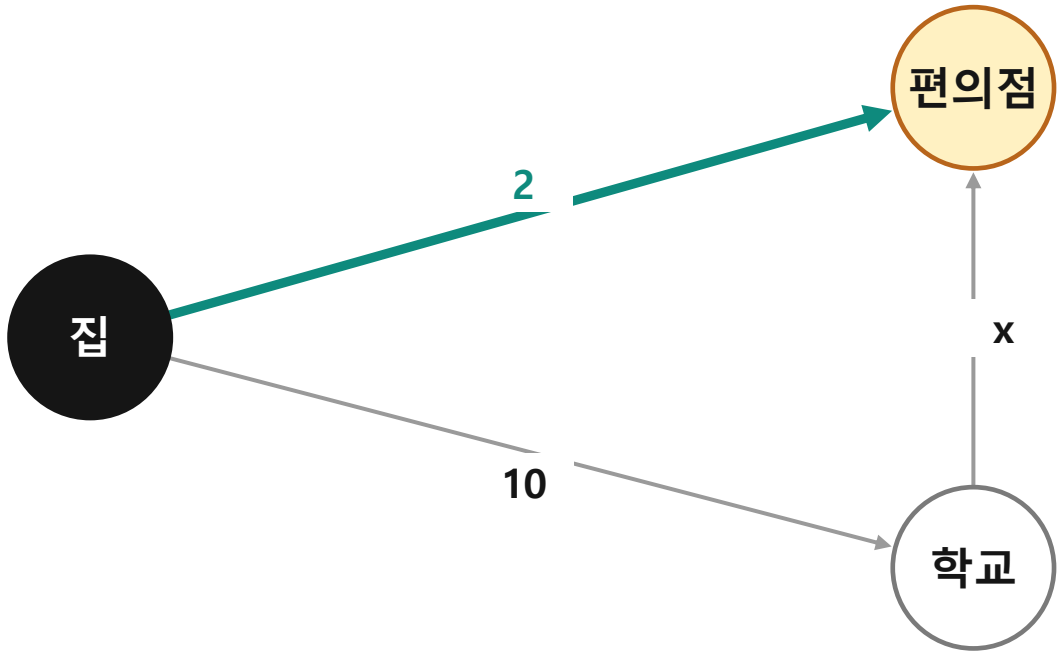
	가장 가까운 이웃 (5.7)	다익스트라 (5.8)
비교하는 거리	지금 도시에서 간선 하나	**시작점에서 누적한 거리 dist**
고르는 것	다음에 이동할 도시	최단 거리를 확정할 정점
고른 뒤	그 도시로 이동	나가는 간선으로 이웃의 dist를 고침
최적 보장	**없음**	**간선 비용이 모두 0 이상이면 있음**

예: A → B 직행은 8, A → C → B는 1 + 5 = 6

다익스트라는 정점까지의 누적 거리를 비교하므로, 직행보다 짧은 경유 경로도 찾아냅니다.

왜 가장 가까운 곳을 확정해도 안전한가

편의점 2분, 학교 10분입니다. 학교를 거쳐 편의점에 가면
10 + x분(x ≥ 0)이라 2분보다 짧아질 수 없습니다.



학교를 거쳐 편의점에 가는 길

10 + x ≥ 10 > 2

학교까지 이미 10분입니다. 간선 비용이 0 이상이면 길을 더 갈수록 거리가 줄지 않으므로 편의점 2분은 더 줄어들 수 없습니다.

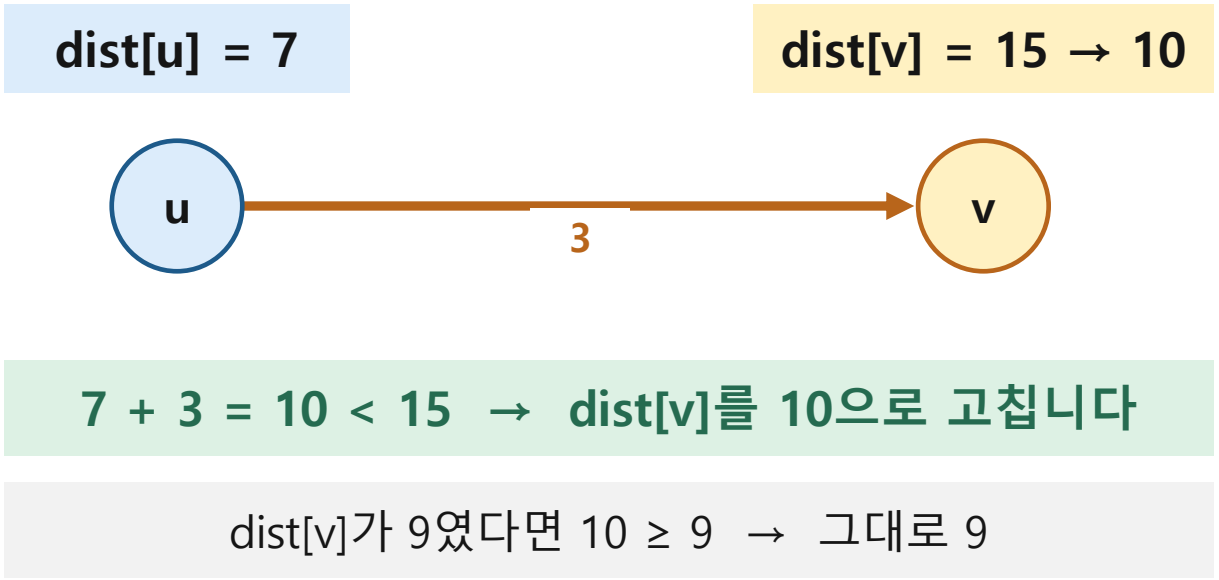
x = −9라면?

10 − 9 = 1 < 2. 음수 간선이 있으면 이 논리가 깨집니다.

검정: 시작점 노랑: 지금 가장 가까운 미확정 정점 x: 학교 → 편의점 비용

완화 · dist[v] 15 → 10

u를 거쳐 v로 가는 길이 더 짧으면 dist[v]를 고칩니다.
연습문제 예: dist[u] = 7, 가중치 3, dist[v] = 15



완화(relaxation)

dist[v] = min(dist[v], dist[u] + w)

relax.pseudo

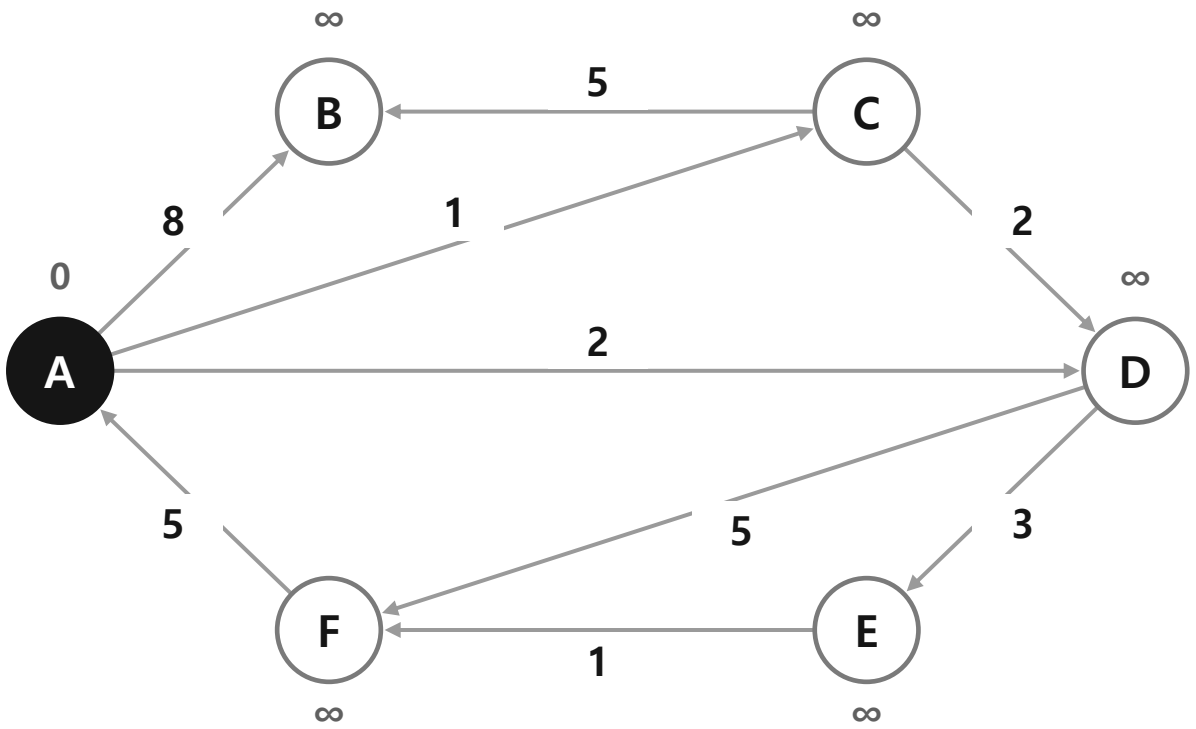
```
if dist[u] + w < dist[v] then
    dist[v] <- dist[u] + w
```

w: u → v 간선의 비용. 더 짧을 때만 고치고, 같거나 길면 그대로 둡니다.

파랑: dist를 아는 정점 u 노랑: 고치려는 정점 v 주황: 완화하는 간선 u → v

예제: A에서 출발 · 초기화

방향 그래프 A~F, 시작점은 A입니다. dist[A] = 0, 나머지는 아직 길을 모르니 ∞로 둡니다.



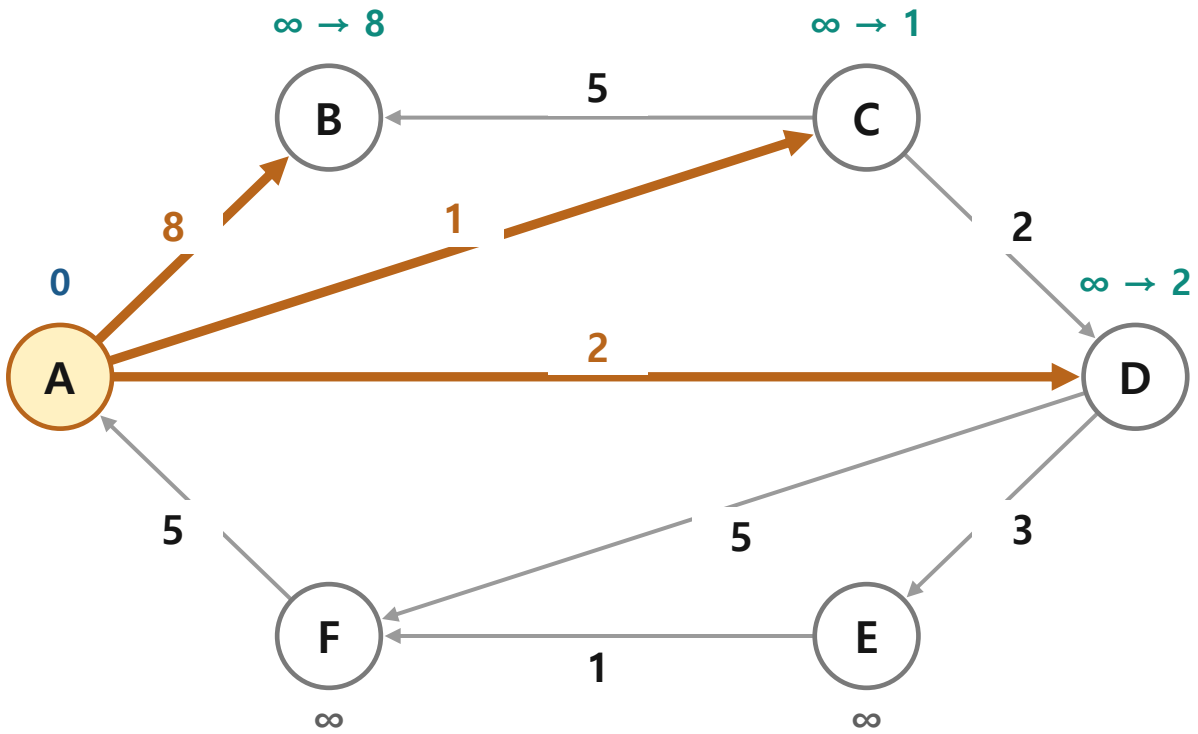
정점	dist	상태
A	0	미확정
B	∞	미확정
C	∞	미확정
D	∞	미확정
E	∞	미확정
F	∞	미확정

다음: 미확정 중 dist가 가장 작은 A(0)를 확정

검정: 시작점 정점 옆 숫자 = 지금의 dist ∞: 아직 길을 모름

① A 확정 · B 8, C 1, D 2

미확정 중 가장 작은 A(0)를 확정합니다. A에서 나가는 간선으로 B, C, D를 고칩니다.



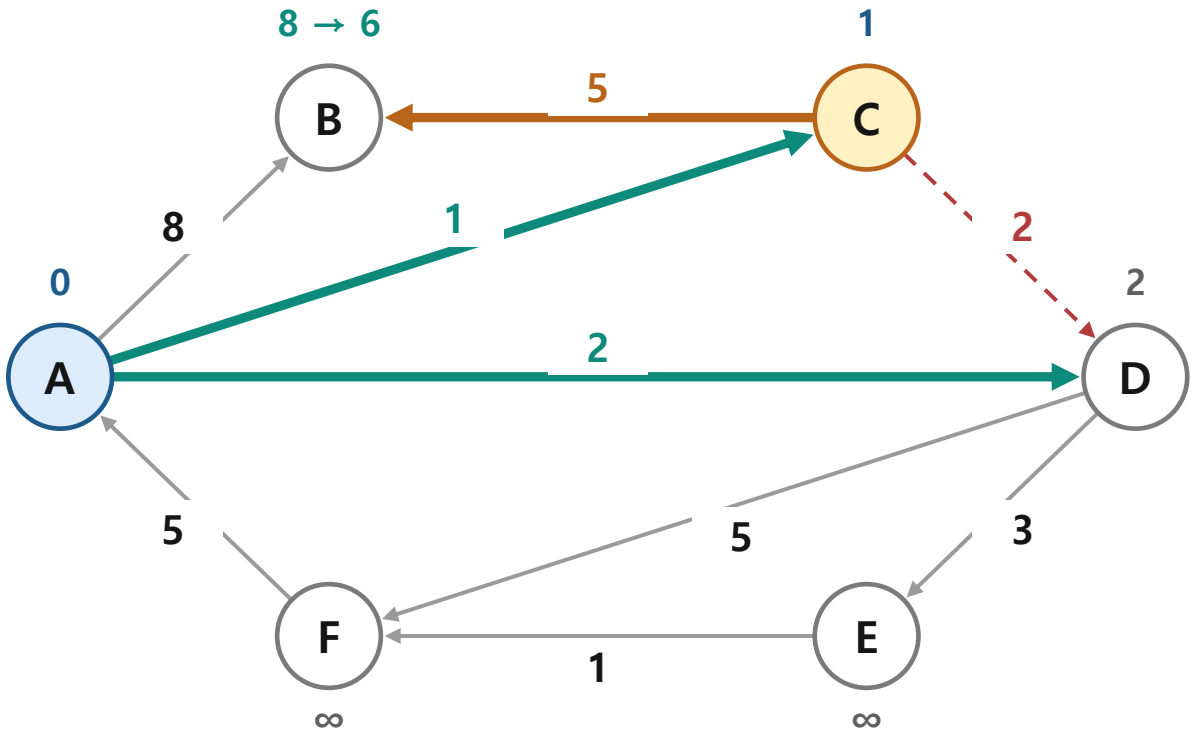
정점	dist	상태
A	0	이번 확정
B	∞ → 8	미확정
C	∞ → 1	미확정
D	∞ → 2	미확정
E	∞	미확정
F	∞	미확정

0 + 8 = 8, 0 + 1 = 1, 0 + 2 = 2. 다음: 가장 작은 C(1)

노랑: 이번에 확정 파랑: 확정 주황: 이번에 완화한 간선 청록: 지금 dist를 만든 간선 빨강 점선: 줄지 않음

② C 확정 · B 8 → 6

가장 작은 C(1)를 확정합니다. C → B로 1 + 5 = 6 < 8이라 B를 6으로 고칩니다.



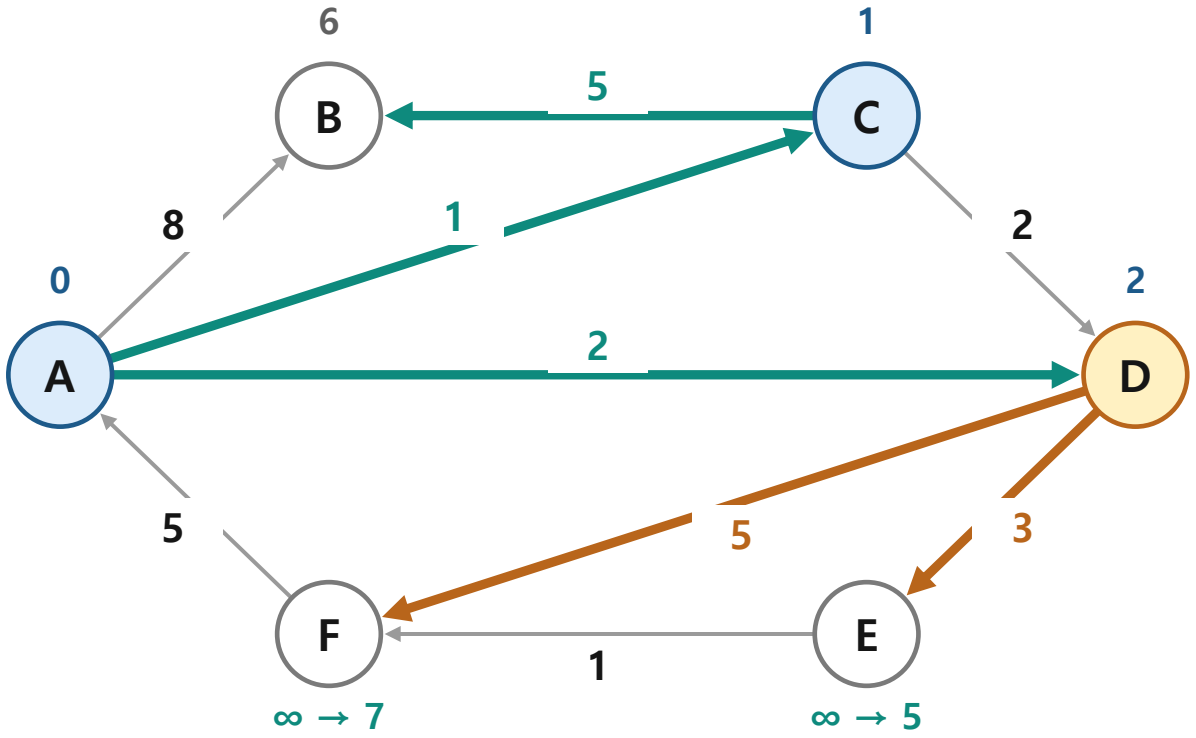
정점	dist	상태
A	0	확정
B	8 → 6	미확정
C	1	이번 확정
D	2	미확정
E	∞	미확정
F	∞	미확정

C → D는 1 + 2 = 3 ≥ 2라 그대로입니다. 다음: D(2)

노랑: 이번에 확정 파랑: 확정 주황: 이번에 완화한 간선 청록: 지금 dist를 만든 간선 빨강 점선: 줄지 않음

③ D 확정 · E 5, F 7

가장 작은 D(2)를 확정합니다. D → E로 2 + 3 = 5, D → F로 2 + 5 = 7이 처음 생깁니다.



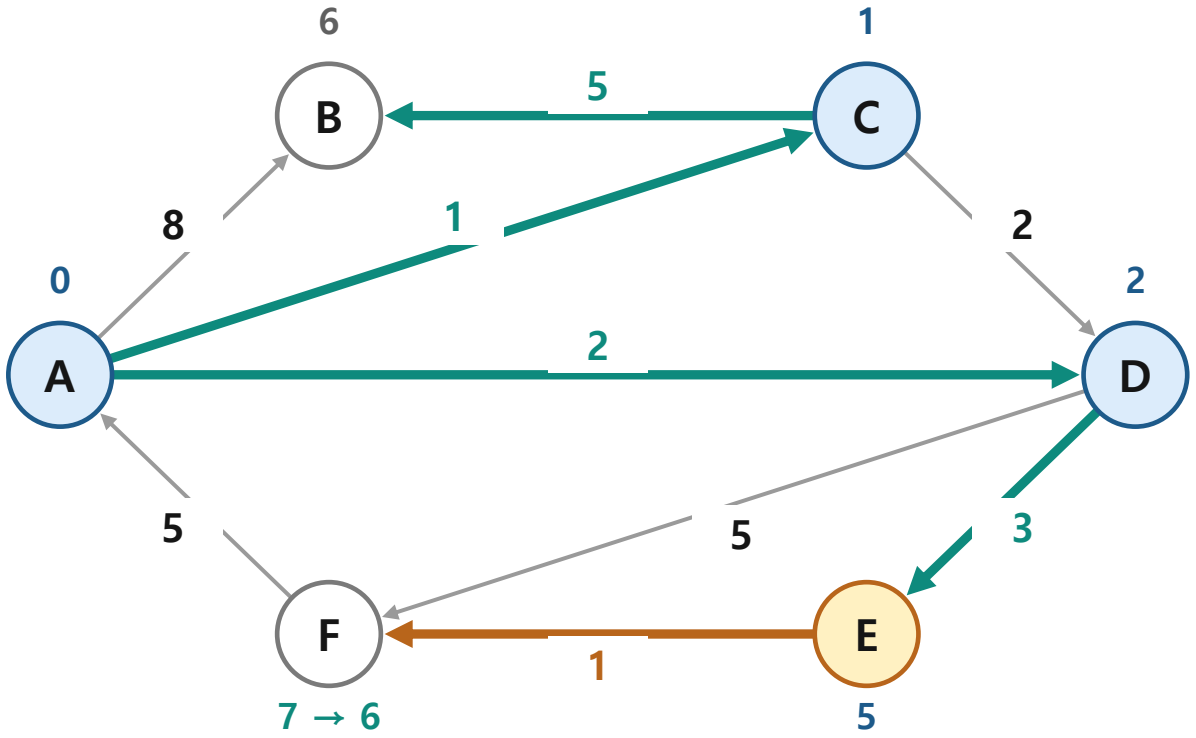
정점	dist	상태
A	0	확정
B	6	미확정
C	1	확정
D	2	이번 확정
E	∞ → 5	미확정
F	∞ → 7	미확정

확정 순서 A, C, D = 거리 0, 1, 2. 다음: E(5)

노랑: 이번에 확정 파랑: 확정 주황: 이번에 완화한 간선 청록: 지금 dist를 만든 간선 빨강 점선: 줄지 않음

④ E 확정 · F 7 → 6

가장 작은 E(5)를 확정합니다. E → F로 5 + 1 = 6 < 7이라 F를 6으로 고칩니다.



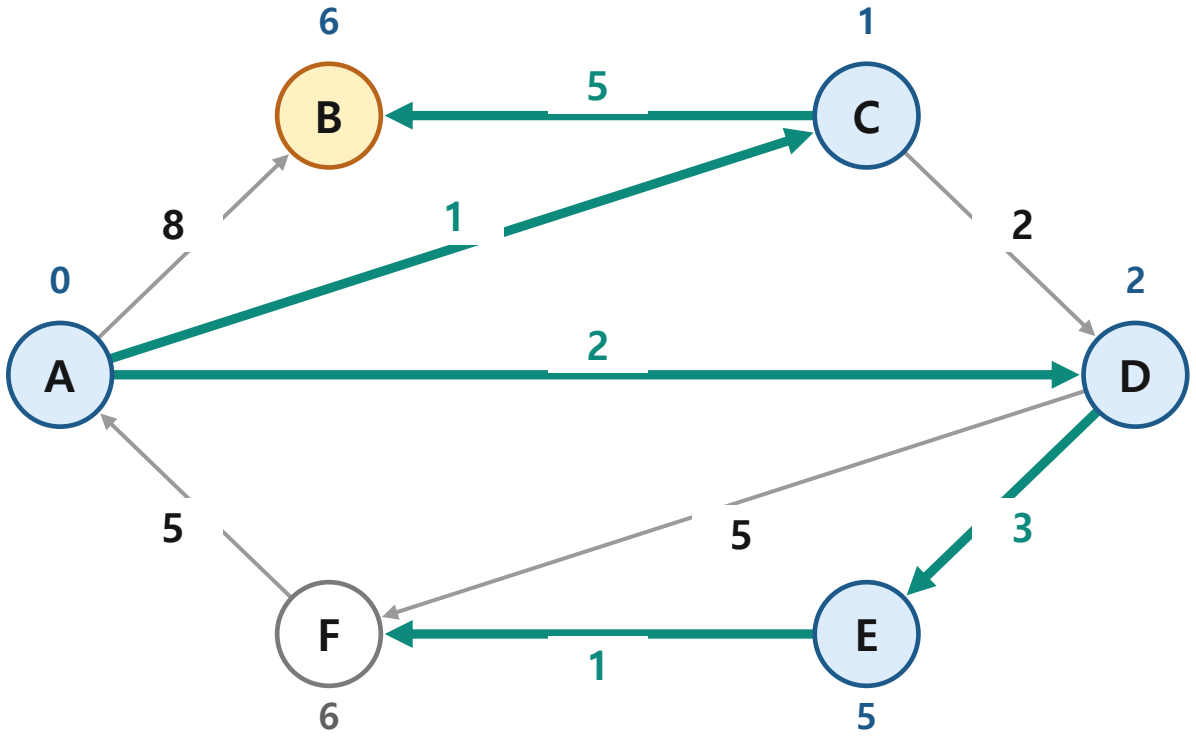
정점	dist	상태
A	0	확정
B	6	미확정
C	1	확정
D	2	확정
E	5	이번 확정
F	7 → 6	미확정

다음: B 6 = F 6. 같으면 어느 쪽을 먼저 해도 됩니다(두 코드 모두 B 먼저).

노랑: 이번에 확정 파랑: 확정 주황: 이번에 완화한 간선 청록: 지금 dist를 만든 간선 빨강 점선: 줄지 않음

⑤ B 확정 · 나가는 간선 없음

B(6)를 확정합니다. B에서 나가는 간선이 없으므로 고칠 값이 없습니다.



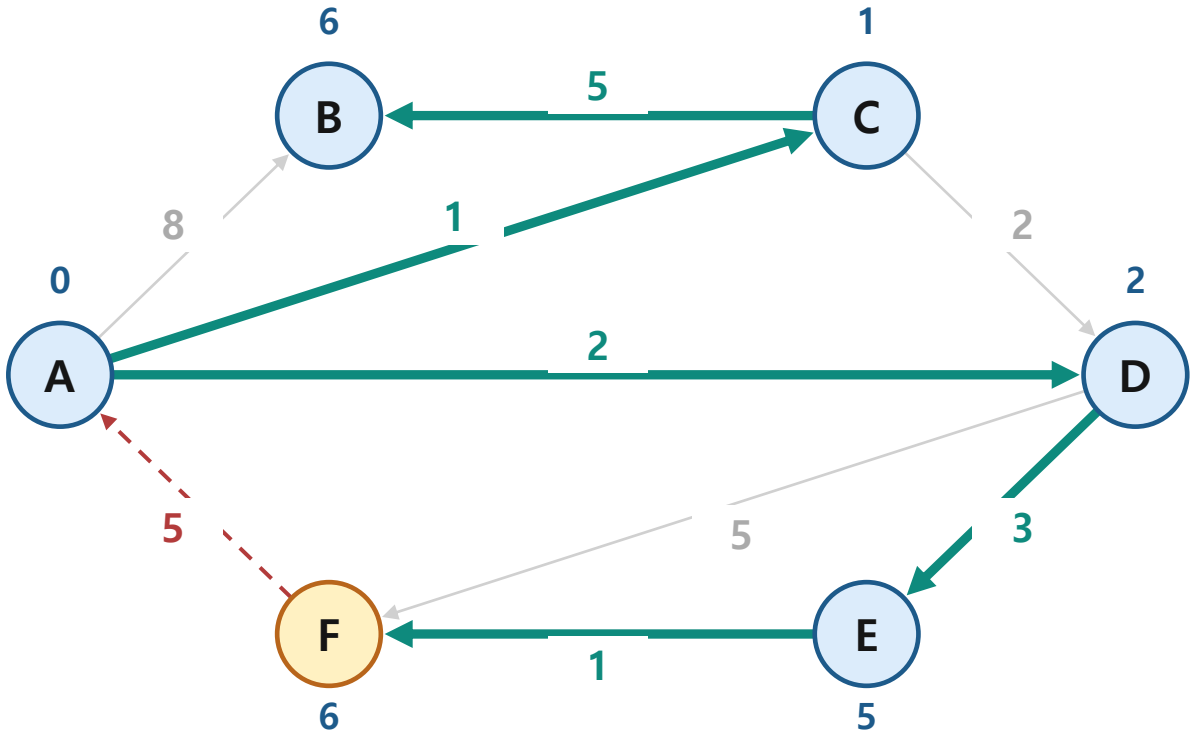
정점	dist	상태
A	0	확정
B	6	이번 확정
C	1	확정
D	2	확정
E	5	확정
F	6	미확정

B = 6은 직행 8이 아니라 A → C → B입니다. 다음: F(6)

노랑: 이번에 확정 파랑: 확정 주황: 이번에 완화한 간선 청록: 지금 dist를 만든 간선 빨강 점선: 줄지 않음

⑥ F 확정 · 최단 거리 완성

F(6)를 확정합니다. F → A는 A가 이미 확정되어 고치지 않습니다(6 + 5 = 11 ≥ 0이기도 합니다).



정점	dist	상태
A	0	확정
B	6	확정
C	1	확정
D	2	확정
E	5	확정
F	6	이번 확정

모든 정점 확정. 청록 간선 5개 = 최단 경로 트리. A → D → E → F = 2 + 3 + 1 = 6

청록: 최단 경로 트리 빨강 점선: 줄지 않은 간선 연회색: 쓰지 않은 간선

다익스트라 의사코드 (배열판)

dijkstra.pseudo

```
DIJKSTRA(G, start):
    dist[모든 v] <- ∞;  visited[모든 v] <- False;  dist[start] <- 0
    V번 반복:
        u <- 미확정 중 dist가 가장 작은 정점 (모두 ∞면 break);  visited[u] <- True   // 확정
        for u에서 나가는 간선 (u, v, w):
            if not visited[v] and dist[u] + w < dist[v] then
                dist[v] <- dist[u] + w          // 완화
    return dist
```

선택: 확정하지 않은 정점 중 dist가 가장 작은 정점

배열을 처음부터 끝까지 훑으면 V번 비교, 이것을 V번 반복 → $O(V^2)$

확정: visited[u] <- True

확정한 정점은 다시 고치지 않습니다

완화: u에서 나가는 간선마다

인접 행렬이면 한 행 V칸을 훑으므로 반복마다 O(V)

배열판 전체 $O(V^2)$: C 코드가 이 방식입니다

우선순위 큐와 heapq · (8, B)와 (6, B)

우선순위 큐: 가장 작은 항목을 먼저 꺼내는 자료구조.
Python은 힙(heap)으로 만든 heapq를 씁니다.



(1, C)를 꺼내 C 확정 → B가 8에서 6으로 → (6, B)를 새로 넣음



힙: 가장 작은 값이 늘 맨 앞(뿌리)에 오게 유지하는 이진 트리. 넣고 꺼낼 때 높이 log n만큼만 자리를 바꿉니다.

함수	하는 일
heapq.heappush(q, x)	x를 넣음
heapq.heappop(q)	최솟값을 꺼냄

둘 다 O(log n). 튜플은 첫 값(거리)부터 비교합니다.

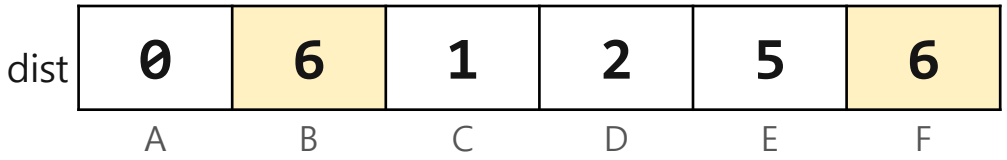
heapq에는 넣은 항목의 거리를 줄이는 함수가 없습니다

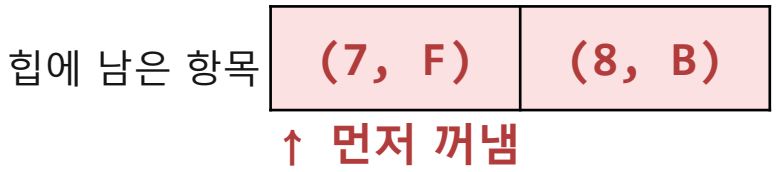
그래서 (6, B)를 하나 더 넣고 (8, B)는 그대로 남겨 둡니다.

항목은 작은 순으로 적었습니다 초록: 새로 넣은 항목 빨강: 오래된 항목

오래된 항목 건너뛰기 · (7, F)와 (8, B)

F까지 확정해도 힙에 (7, F), (8, B)가 남습니다. 꺼낸 거리가 dist보다 크면 오래된 항목이라 건너뜁니다.





(7, F): dist[F] = 6 < 7 → 건너뜀

(8, B): dist[B] = 6 < 8 → 건너뜀

꺼낸 항목	처리
(0, A)	A 확정
(1, C)	C 확정
(2, D)	D 확정
(5, E)	E 확정
(6, B)	B 확정
(6, F)	F 확정
(7, F)	**건너뜀**
(8, B)	**건너뜀**

힙이 비면 반복이 끝납니다.

노랑: 이미 6으로 확정된 B, F 빨강: 오래된 항목

Python: 다익스트라 ① 초기화

dijkstra.py

```
import heapq          # 우선순위 큐

INF = float("inf")

def dijkstra(graph, start):
    # 초기화: 모든 노드까지의 거리를 무한대로
    distances = {node: INF for node in graph}
    distances[start] = 0
    queue = []
    heapq.heappush(queue, (0, start))   # (거리, 노드)
```

Python: 다익스트라 ② 꺼내고 완화하기

dijkstra.py

```
while queue:
    # 가장 가까운 노드 선택 – 그리디 선택
    cur_dist, cur_node = heapq.heappop(queue)
    # 이미 더 짧은 경로를 찾았다면 무시
    if distances[cur_node] < cur_dist:
        continue
    # 이웃 노드 갱신 (relaxation)
    for neighbor in graph[cur_node]:
        new_dist = cur_dist + graph[cur_node][neighbor]
        if new_dist < distances[neighbor]:
            distances[neighbor] = new_dist
            heapq.heappush(queue, (new_dist, neighbor))
return distances
```

Python: 다익스트라 ③ 실행

dijkstra.py

```
my_graph = {
    "A": {"B": 8, "C": 1, "D": 2},
    "B": {}, "C": {"B": 5, "D": 2},
    "D": {"E": 3, "F": 5},
    "E": {"F": 1}, "F": {"A": 5}
}
shortest = dijkstra(my_graph, "A")
print("시작점 [A]로부터의 최단 거리:")
for node, dist in shortest.items():
    print(f"A -> {node} : {dist}")
```

C: 다익스트라 ① 초기화

dijkstra.c

```
/* C에는 힙이 없어 미확정 노드 중 최소를 배열에서 찾는다: O(V²) */
#include <limits.h>
#include <stdio.h>
#define V 6                     /* 노드 수 */
#define INF INT_MAX

void dijkstra(int graph[][V], int start, int dist[]) {
    int visited[V] = {0};       /* 최단 거리가 확정되었는가 */
    /* 초기화: 모든 노드까지의 거리를 무한대로 */
    for (int i = 0; i < V; i++) dist[i] = INF;
    dist[start] = 0;
```

C: 다익스트라 ② 확정과 완화

dijkstra.c

```
    for (int k = 0; k < V; k++) {
        /* 가장 가까운 노드 선택 – 그리디 선택 */
        int u = -1, min_dist = INF;
        for (int i = 0; i < V; i++)
            if (!visited[i] && dist[i] < min_dist) {
                min_dist = dist[i];
                u = i;
            }
        if (u == -1) break;     /* 남은 노드는 갈 수 없다 */
        visited[u] = 1;         /* 최단 거리 확정 */
        /* 이웃 노드 갱신 (relaxation) */
        for (int v = 0; v < V; v++) {
            if (graph[u][v] == 0 || visited[v]) continue;
            int new_dist = dist[u] + graph[u][v];
            if (new_dist < dist[v]) dist[v] = new_dist;
        }
    }
}
```

C: 다익스트라 ③ 실행

dijkstra.c

```
int main(void) {
    const char *names = "ABCDEF";
    /* graph[u][v]: u에서 v로 가는 가중치, 0이면 간선 없음 */
    int graph[V][V] = {
        {0, 8, 1, 2, 0, 0},     /* A */
        {0, 0, 0, 0, 0, 0},     /* B */
        {0, 5, 0, 2, 0, 0},     /* C */
        {0, 0, 0, 0, 3, 5},     /* D */
        {0, 0, 0, 0, 0, 1},     /* E */
        {5, 0, 0, 0, 0, 0}      /* F */
    };
    int dist[V];
    dijkstra(graph, 0, dist);
    printf("시작점 [A]로부터의 최단 거리:\n");
    for (int v = 0; v < V; v++)
        printf("A -> %c : %d\n", names[v], dist[v]);
    return 0;
}
```

실행 결과: A에서 각 정점까지

OUTPUT

```
시작점 [A]로부터의 최단 거리:
A -> A : 0
A -> B : 6
A -> C : 1
A -> D : 2
A -> E : 5
A -> F : 6
```

정점	A	B	C	D	E	F
최단 거리	0	6	1	2	5	6
최단 경로	A	A→C→B	A→C	A→D	A→D→E	A→D→E→F

Python(힙)과 C(배열)의 출력이 같습니다. B는 직행 8이 아니라 C를 거친 6입니다.

이 코드는 거리만 출력합니다. 경로까지 출력하려면 dist를 고칠 때 직전 정점도 함께 저장합니다.

배열 $O(V^2)$ vs 힙 $O(E \log V)$

V: 정점 수, E: 간선 수. 예제는 V = 6, E = 9입니다.

	배열판 (C)	힙판 (Python)
최소 찾기	V칸 훑기 O(V) × V번	heappop O(log E) × 최대 E + 1번
완화	한 행 V칸 훑기 × V번	간선마다 한 번, heappush O(log E)
전체	**$O(V^2)$**	**$O(E \log V)$**
유리한 그래프	간선이 많을 때 ($E \approx V^2$)	간선이 적을 때 ($E \ll V^2$)

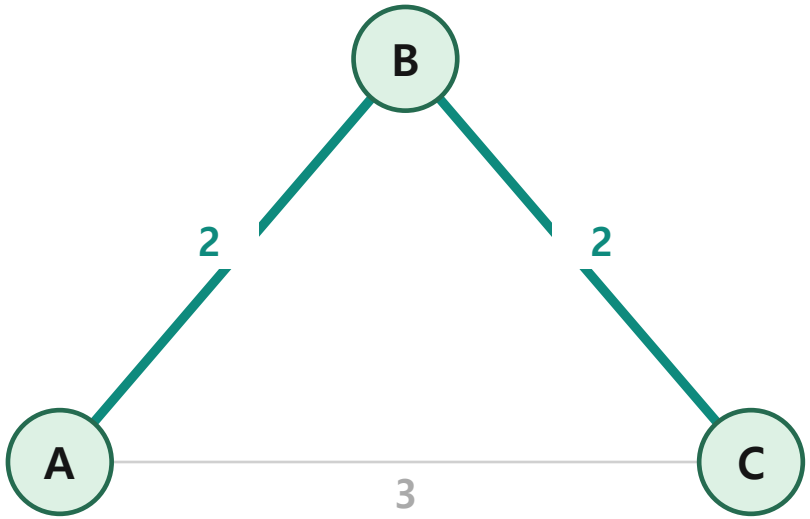
$E \le V^2$이므로 $\log E \le 2 \log V$, 즉 힙 연산 한 번은 $O(\log V)$입니다.

초기화 $O(V)$ + 힙 연산 $O(E \log V)$. 모든 정점에 갈 수 있으면 $E \ge V - 1$이라 $O(E \log V)$로 씁니다.

최소 신장 트리와 최단 경로 트리는 다를 수 있다

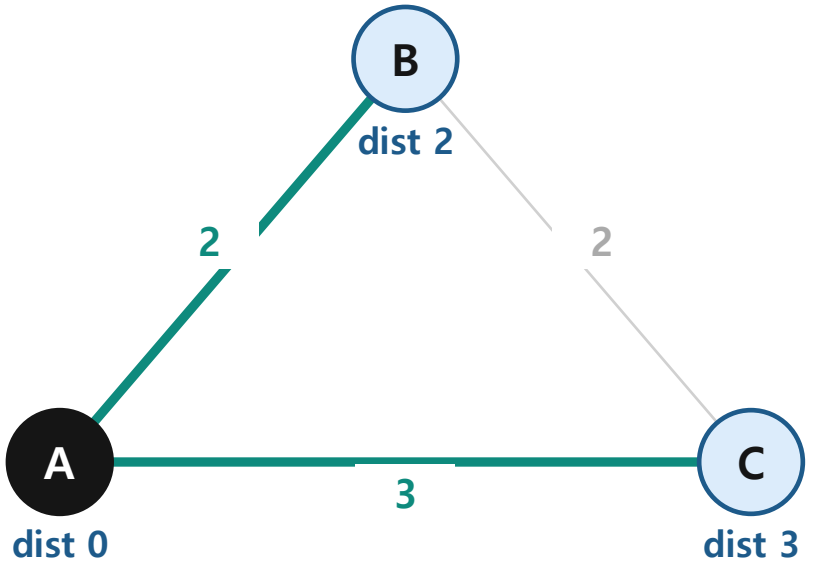
MST는 간선 합을, 최단 경로 트리는 시작점에서 각 정점까지의 거리를 최소로 합니다. 같은 그래프에서도 둘은 다를 수 있습니다.

최소 신장 트리(크루스칼)



간선 합 2 + 2 = 4 (최소) · A → C 거리 4

최단 경로 트리(다익스트라, 시작 A)

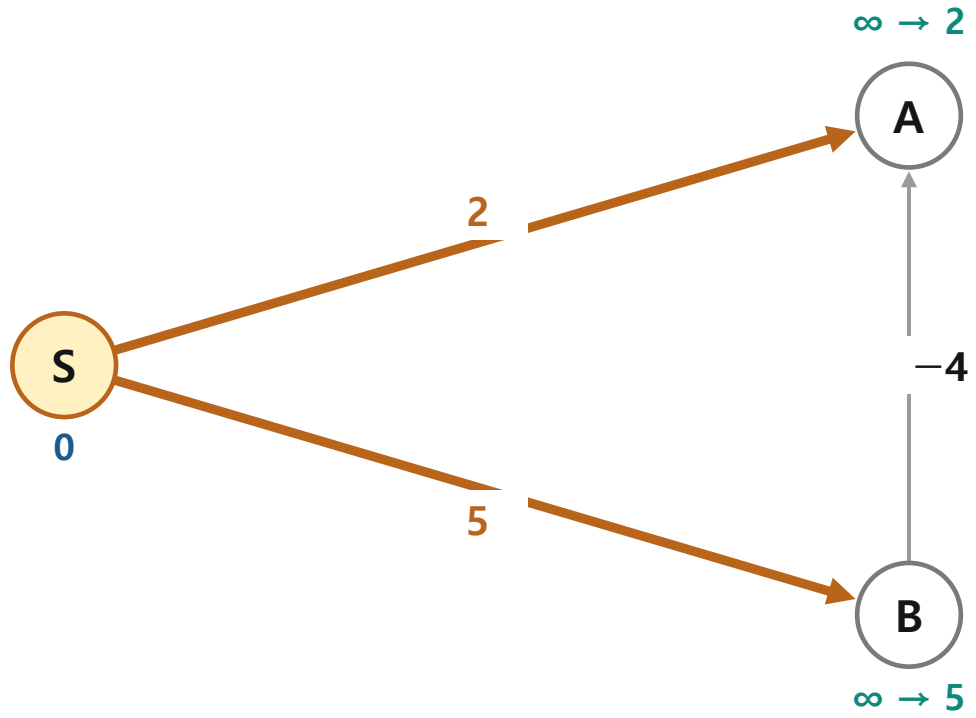


간선 합 2 + 3 = 5 · A → C 거리 3 (최단)

청록: 트리 간선 연회색: 트리에 없는 간선 방향 없는 간선은 양방향 간선 2개로 보고 다익스트라를 돌립니다

음수 간선 ① S 확정 · A 2, B 5

연습문제 예: S → A 2, S → B 5, B → A −4. 시작점 S를 확정하고 A, B를 고칩니다.



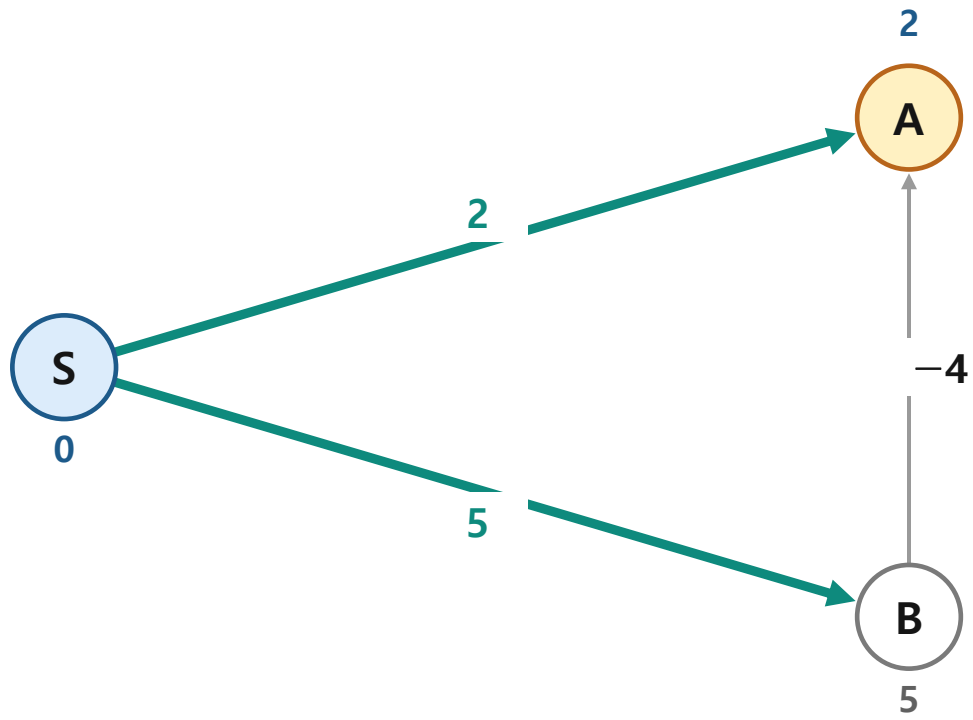
정점	dist	상태
S	0	이번 확정
A	∞ → 2	미확정
B	∞ → 5	미확정

0 + 2 = 2, 0 + 5 = 5. 다음: 미확정 A 2, B 5 중 더 작은 A

노랑: 이번에 확정 주황: 이번에 완화한 간선 B → A −4: 음수 간선

음수 간선 ② A를 2로 확정

미확정 중 가장 작은 A(2)를 확정합니다. 이제 dist[A] = 2는 다시 고치지 않습니다.



노랑: 이번에 확정 파랑: 확정 청록: 지금 dist를 만든 간선

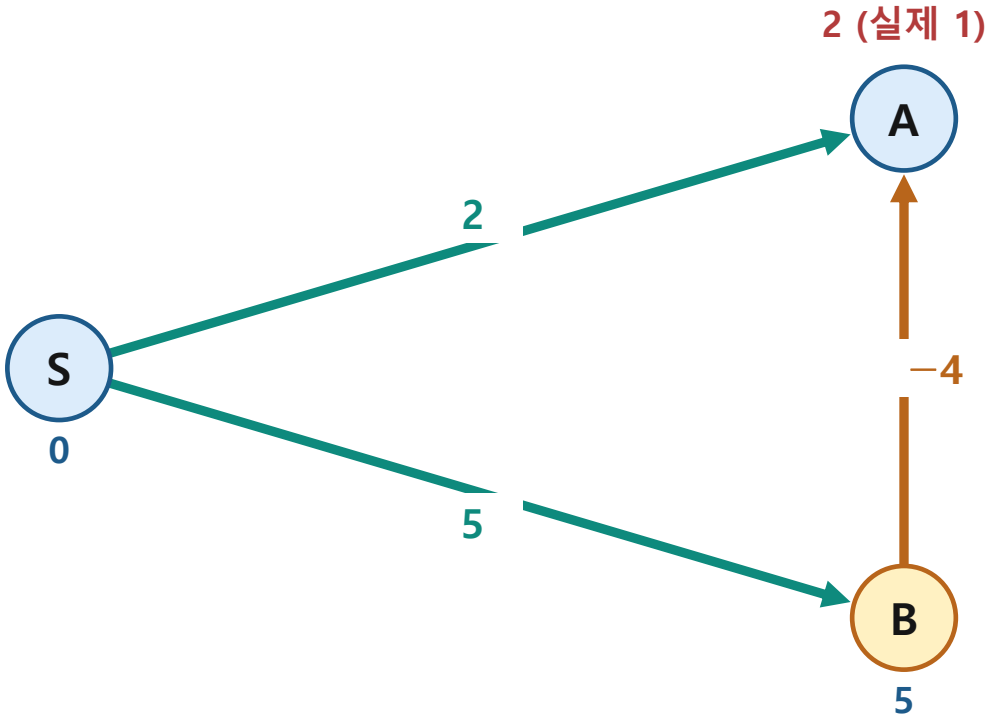
정점	dist	상태
S	0	확정
A	2	이번 확정
B	5	미확정

dist[A] = 2 확정

비용이 0 이상이라면 옳은 판단입니다. 하지만 B를 거치는 길에 −4가 남아 있습니다.

음수 간선 ③ B 확정 · 실제 최단 1을 놓친다

B(5)를 확정하고 B → A를 보면 5 + (−4) = 1 < 2입니다.
하지만 A는 이미 확정되어 고치지 않습니다.



정점	dist	상태
S	0	확정
A	2	확정
B	5	이번 확정

다익스트라(의사코드·C)의 답: A = 2

실제 최단은 S → B → A = 1

A를 2로 확정한 뒤 다시 처리하지 않으므로, A에서 나가는 간선이 있었다면 그 뒤 정점도 틀립니다.

주황: 1을 만드는 간선 B → A 음수 간선이 있으면 벨만-포드 같은 다른 알고리즘을 씁니다

다익스트라 확인

① dist[u] = 4, w(u, v) = 5, dist[v] = 12이면 완화 뒤 dist[v]는? dist[v]가 8이었다면?

② 예제 그래프(A~F)에서 A → F의 최단 경로와 거리는?

③ 음수 간선 예(S → A 2, S → B 5, B → A −4)에서 다익스트라의 답과 실제 최단 거리는? 틀리는 이유는?

④ 이 장의 Python 힙 코드에 ③의 그래프를 넣으면 A는 얼마가 나올까요? 의사코드와 다른 이유는?

⑤ 최소 신장 트리와 최단 경로 트리는 항상 같은가요?

⑥ 정점 V개, 간선 E개일 때 배열판과 힙판의 시간 복잡도는?

5.9

코딩 테스트 예제

CT(Coding Test) 문제 두 개를 그리디로 풉니다.
동전 내주기 · ATM 줄 세우기

CT 5.1 동전 내주기 · 문제

동전 N종류(각각 무한히 많음)로 K원을 만들 때 필요한 **동전의 최소 개수**를 구합니다.

입력

첫 줄에 N과 K, 다음 N줄에 동전 가치가 오름차순으로

조건

첫 동전은 1원, 큰 단위는 작은 단위의 배수

출력

최소 개수 하나

기호 주의: 여기서 N은 동전 종류 수, K는 금액입니다(5.2절의 K는 종류 수).

배수 조건 → 그리디가 안전

100원 5개 = 500원 1개처럼 작은 동전 여러 개를 큰 동전으로 바꿀 수 있어, 큰 동전을 덜 쓰면 개수만 늘어납니다.

배수 조건은 충분조건입니다. 깨지면 보장이 사라지고(400원 예), {1, 5, 10, 25}처럼 배수가 아니어도 맞는 경우가 있습니다.

1원이 꼭 필요한 이유

{5, 10}으로 7원이면 5원 1개 후 2원이 남는데도 1을 출력합니다.

Python: 동전 내주기 ① 입력

coin_change.py

```
import sys

# N: 동전의 종류 수, K: 목표 금액
N, K = map(int, sys.stdin.readline().split())
# 조건: 오름차순, 첫 동전은 1원, 큰 단위는 작은 단위의 배수
coins = [int(sys.stdin.readline()) for _ in range(N)]
```

Python: 동전 내주기 ② 큰 단위부터

coin_change.py

```
# 핵심 전략: 큰 단위부터 확인하도록 뒤집는다
coins.reverse()

count = 0
for coin in coins:
    if K >= coin:
        count += K // coin   # 한 번에 몇 개 쓸 수 있는가
        K %= coin            # 남은 금액 갱신
    if K == 0:               # 다 거슬러 줬으면 종료
        break

print(count)
```

coins.reverse()는 오름차순 입력을 뒤집기만 하므로 O(N)입니다(정렬이 아닙니다).

C: 동전 내주기 ① 입력

coin_change.c

```c
#include <stdio.h>

int main(void) {
    int N, K, coins[100];
    /* N: 동전의 종류 수, K: 목표 금액 */
    if (scanf("%d %d", &N, &K) != 2 || N < 0 || N > 100) return 1;
    /* 조건: 오름차순, 첫 동전은 1원, 큰 단위는 작은 단위의 배수 */
    for (int i = 0; i < N; i++)
        if (scanf("%d", &coins[i]) != 1) return 1;
```

C: 동전 내주기 ② 큰 단위부터

coin_change.c

```
    /* 핵심 전략: 큰 단위부터 확인하도록 뒤에서부터 본다 */
    int count = 0;
    for (int i = N - 1; i >= 0; i--) {
        if (K >= coins[i]) {
            count += K / coins[i];  /* 한 번에 몇 개 쓸 수 있는가 */
            K %= coins[i];          /* 남은 금액 갱신 */
        }
        if (K == 0)                 /* 다 거슬러 줬으면 종료 */
            break;
    }

    printf("%d\n", count);
    return 0;
}
```

CT 5.1 실행 결과: 4,200원 → 6개

입력 coin_change.in

```
10 4200
1
5
10
50
100
500
1000
5000
10000
50000
```

OUTPUT

```
6
```

동전(큰 것부터)	몫 = 개수	남은 금액
50,000 · 10,000 · 5,000	0	4,200
1,000	**4**	200
500	0	200
100	**2**	0 → 반복 끝

1,000원 4개 + 100원 2개 = 6개

시간 복잡도 O(N): 뒤집기 O(N) + 동전마다 한 번씩. 금액 K가 커져도 반복은 늘지 않습니다.

CT 5.2 ATM 줄 세우기 · 문제

ATM 한 대 앞에 N명이 줄을 섭니다. i번 사람이 돈을 뽑는 데 P[i]분이 걸립니다. 줄 서는 순서를 정해 **각 사람이 기다린 시간 + 인출 시간의 합**을 최소로 만드세요.

입력

첫 줄에 N, 둘째 줄에 P[1] … P[N]

출력

모든 사람의 (기다린 시간 + 인출 시간)을 더한 값의 최솟값

한 사람의 기다린 시간 + 인출 시간 = 그 사람이 일을 마치는 시각

예: 5명, P = [3, 1, 4, 3, 2] → 답 32

정렬하면 39 → 32

입력 순서 그대로면 마치는 시각의 합이 39, 인출 시간 오름차순으로 세우면 32입니다.

입력 순서 그대로

인출 시간	3	1	4	3	2
마치는 시각	3	4	8	11	13

3 + 4 + 8 + 11 + 13 = 39

인출 시간 오름차순으로 정렬

인출 시간	1	2	3	3	4
마치는 시각	1	3	6	9	13

1 + 3 + 6 + 9 + 13 = 32

마치는 시각 = 앞사람의 마치는 시각 + 자기 인출 시간 (누적합)

왜 오름차순인가: 이웃한 두 사람 바꾸기

긴 사람 a가 짧은 사람 b 바로 앞에 있으면, 둘을 바꿔 합을 a − b만큼 줄일 수 있습니다.

예: 입력의 앞 두 사람 a = 3, b = 1 (두 사람 앞에서 마친 시각 T, 여기서는 T = 0)

바꾸기 전	a = 3	b = 1	
마치는 시각	T + 3	T + 4	합 2T + 7
바꾼 뒤	b = 1	a = 3	
마치는 시각	T + 1	T + 4	합 2T + 5

빨강: 바꾸기 전 앞사람의 마치는 시각 초록: 바꾼 뒤 줄어든 시각

일반적으로 a > b이면

바로 이웃한 a, b를 바꾸면

```
바꾸기 전: (T+a) + (T+a+b)
바꾼 뒤:   (T+b) + (T+b+a)
차이 = a - b > 0
```

두 사람의 앞과 뒤에 선 사람들이 마치는 시각은 그대로입니다.

긴 사람이 짧은 사람 바로 앞에 선 곳이 없는 순서 = 오름차순

Python: ATM 줄 세우기

atm.py

```
import sys

N = int(sys.stdin.readline())
times = list(map(int, sys.stdin.readline().split()))

# 핵심 전략: 오름차순 정렬 (그리디)
# 짧게 걸리는 사람이 앞에 서야 뒷사람 대기가 줄어든다
times.sort()

total_sum = 0     # 모든 사람의 소요 시간 합 (정답)
current_sum = 0   # 현재 사람까지의 누적 시간

for t in times:
    current_sum += t      # 대기 시간 + 내 인출 시간
    total_sum += current_sum

print(total_sum)
```

C: ATM 줄 세우기 ① 비교 함수와 입력

atm.c

```
#include <stdio.h>
#include <stdlib.h>

int ascending(const void *a, const void *b) {
    return *(const int *)a - *(const int *)b;
}

int main(void) {
    int N, times[1000];
    if (scanf("%d", &N) != 1 || N < 0 || N > 1000) return 1;
    for (int i = 0; i < N; i++)
        if (scanf("%d", &times[i]) != 1) return 1;
```

C: ATM 줄 세우기 ② 정렬과 누적합

atm.c

```
    /* 핵심 전략: 오름차순 정렬 (그리디) */
    /* 짧게 걸리는 사람이 앞에 서야 뒷사람 대기가 줄어든다 */
    qsort(times, N, sizeof(int), ascending);

    int total_sum = 0;      /* 모든 사람의 소요 시간 합 (정답) */
    int current_sum = 0;    /* 현재 사람까지의 누적 시간 */

    for (int i = 0; i < N; i++) {
        current_sum += times[i];    /* 대기 시간 + 내 인출 시간 */
        total_sum += current_sum;
    }

    printf("%d\n", total_sum);
    return 0;
}
```

CT 5.2 실행 결과와 복잡도

입력 atm.in

```
5
3 1 4 3 2
```

OUTPUT

```
32
```

정렬 [1, 2, 3, 3, 4] → 마치는 시각 1, 3, 6, 9, 13 → 합 32

시간 복잡도 O(N log N): 정렬 O(N log N) + 누적 O(N)

두 문제 모두 고를 기준(큰 동전부터, 짧은 사람부터)을 정하고, 그 기준이 왜 안전한지(배수 조건, 교환 논증)를 확인했습니다.

5장 정리

01 그리디는 되돌아보지 않는다

매 단계 지금 기준의 최선을 확정하고 번복하지 않습니다. 선택 → 적정성 검사 → 해답 검사를 해답이 될 때까지 반복합니다.

02 최적을 보장하는 두 조건

탐욕적 선택 속성(이 선택을 포함하는 최적해가 있다)과 최적 부분 구조. "항상 최적"은 반례 하나로 반박합니다.

03 정렬 기준이 곧 전략

활동 선택은 종료가 빠른 순, 쪼갤 수 있는 배낭은 가성비가 큰 순, 크루스칼은 가중치가 작은 순, ATM은 인출 시간이 짧은 순.

04 최적을 보장하는 경우

배수 조건의 거스름돈, 활동 선택, 쪼갤 수 있는 배낭(가성비 순), 크루스칼 MST, 다익스트라(간선 비용이 모두 0 이상일 때).

05 보장하지 못하는 경우

400원이나 {1, 3, 4} 동전, 0/1 배낭, TSP의 가장 가까운 이웃, 음수 간선이 있는 다익스트라.

06 다음은 동적 계획법

0/1 배낭, 400원 동전, TSP는 남은 상태를 기억하는 6장 동적 계획법으로 정확한 답을 구합니다(TSP는 그래도 지수 시간).

Python · C 대응표

같은 알고리즘을 두 언어로 쓸 때 자주 바뀌는 부분입니다.

하는 일	Python	C
정렬	list.sort(key=..., reverse=True)	qsort(배열, 개수, 크기, 비교 함수)
큰 것부터 보기	coins.reverse()	for (i = N - 1; i >= 0; i--)
무한대	float("inf"), float("-inf")	INT_MAX, INT_MIN (limits.h)
최솟값 꺼내기	heapq.heappush / heappop	배열을 훑어 최소 찾기 O(V)
묶음 자료	class Item, 튜플 (w, u, v)	typedef struct { ... } Item;
그래프(다익스트라)	딕셔너리 {정점: {이웃: 비용}}	2차원 배열 graph[u][v], 0 = 간선 없음
입력	sys.stdin.readline().split()	scanf("%d %d", &N, &K)
몫 · 나머지	K // coin, K % coin	K / coins[i], K % coins[i]

비교 함수는 a가 b보다 앞에 와야 하면 음수, 같으면 0, 뒤에 와야 하면 양수를 돌려줍니다. 오름차순이면 앞이 작을 때 음수입니다(배낭의 by_ratio는 반대).

INT_MAX는 진짜 무한대가 아니어서 더하면 넘칩니다.

Q & A

지금 가장 좋아 보이는 답을 골라 봅시다.
틀렸다면, 왜 안전하지 않았는지 함께 따져 봅시다.