그림으로 쉽게 설명하는 파이썬 알고리즘

CHAPTER 05

그리디 알고리즘

Greedy Algorithm

지금 이 순간 가장 좋아 보이는 것을 고른다

CONTENTS

이번 장에서 배울 내용

5.1

그리디란?

눈앞의 최선을 고른다

5.2

거스름돈 문제

가장 큰 동전부터 집는다

5.3

활동 선택 문제

회의실을 가장 많이 쓰는 법

5.4

배낭 문제

쪼갤 수 있느냐가 갈림길

5.5

그리디의 두 조건

언제 최적해가 보장되는가

5.6

크루스칼 알고리즘

가장 싼 간선부터 잇는다

5.7

TSP와 그리디

가까운 도시부터 가면?

5.8

다익스트라

최단 경로의 표준 해법

5.9

코딩 테스트

동전 내주기 · ATM 줄 세우기

그리디가 언제 통하고 언제 무너지는지를 가려내는 것이 이 장의 핵심이다.

5.1

그리디 알고리즘이란?

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

5.1

매 순간 가장 탐욕스러운 선택

그리디 알고리즘은 이름 그대로 매 순간 가장 탐욕스러운 선택을 한다. 미래의 결과나 전체적인 상황을 복잡하게 따지지 않는다. 그저 지금 상황에서 가장 좋아 보이는 것을 덥석 집어 들고, 그 선택을 두 번 다시 번복하지 않은 채 끝까지 밀고 나간다.

> 되돌아보지 않기 때문에 빠르다. 그리고 되돌아보지 않기 때문에 가끔 틀린다.

선택 → 검사 → 해답

직관적 이해 — 안개 낀 산에서 정상 찾기

짙은 안개가 껴서 한 치 앞도 보이지 않는 산속에 고립되었다고 하자. 목표는 가장 높은 봉우리에 오르는 것이다. 지도가 없다면 할 수 있는 전략은 하나뿐이다.

한 걸음의 원칙

발을 디딜 때마다 지금 위치보다 더 높은 쪽으로 한 걸음씩 옮긴다. 딱 그것뿐이다.

놀랄 만큼 빠르다

전체 지형을 조사하지 않으니 계산이 가볍고 구현도 짧다.

그러나 함정이 있다

오른 곳이 산 전체의 정상이 아니라 작은 언덕일 수도 있다. 지역 최적에 갇히는 것이다.

그리디는 "지금 가장 높은 쪽"만 본다. 그 판단이 항상 정상으로 이어지는지는 별도로 증명해야 한다.

그리디 알고리즘의 세 단계

선택

selection — 남은 후보 중 지금 기준으로 가장 좋은 것 하나를 고른다.

적정성 검사

feasibility — 그 선택을 넣어도 제약 조건을 어기지 않는지 확인한다.

해답 검사

solution — 지금까지 모은 것이 완전한 해답인지 판단한다.

그리디는 한 번 고른 것을 되돌리지 않는다 — 이전 선택을 재검토하는 단계가 아예 없다.

greedy_skeleton.pseudo

```
GREEDY(C):                          // C: 후보 집합
    S <- 빈 집합
    while C가 비어있지 않고 S가 해답이 아닌 동안 do
        x <- SELECT(C)              // 선택: 가장 좋아 보이는 것
        C <- C - {x}
        if FEASIBLE(S + {x}) then  S <- S + {x}   // 적정성 검사
    if SOLUTION(S) then return S else return "해 없음"
```

5.2

거스름돈 문제

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

5.2

§5.2 거스름돈 문제

손님에게 1,260원을 거슬러 준다

손님에게 거스름돈 N원을 거슬러 줘야 한다. 호주머니에는 500원, 100원, 50원, 10원짜리 동전이 무한히 많이 들어 있다. 동전 개수를 최소로 하려면 어떤 순서로 집어야 할까?

가장 큰 단위부터

지금 남은 금액을 거슬러 줄 수 있는 가장 큰 동전을 고른다.

가능한 만큼 최대로

그 동전으로 낼 수 있는 최대 개수를 한 번에 계산한다 (몫).

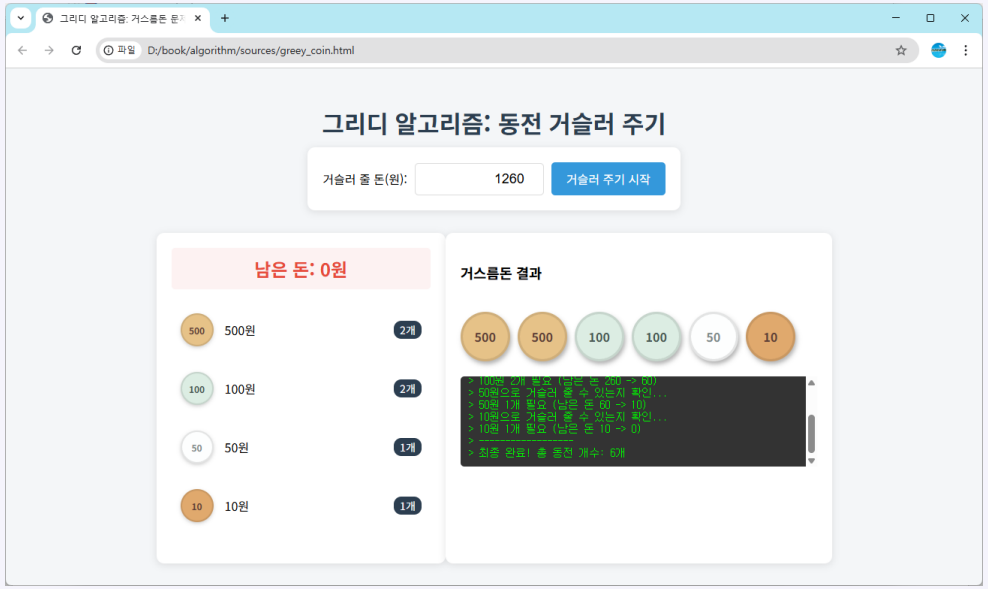
남은 돈으로 반복

나머지 금액에 대해 그다음 큰 단위로 같은 과정을 되풀이한다.

1,260원 = 500×2 + 100×2 + 50×1 + 10×1 → 동전 6개. 이보다 적게 만드는 방법은 없다.

큰 단위부터 최대한 — 동작 과정

1,260원을 500원부터 훑어 내려간다. 각 단계에서 몫만큼 한 번에 집고, 나머지를 다음 단계로 넘긴다.



500원 1260 ÷ 500 = 2개, 남은 돈 260원 — **2개**

100원 260 ÷ 100 = 2개, 남은 돈 60원 — **2개**

50원 60 ÷ 50 = 1개, 남은 돈 10원 — **1개**

10원 10 ÷ 10 = 1개, 남은 돈 0원 — **1개**

합계 6개. 큰 단위가 작은 단위의 배수인 한국 화폐 체계에서는 이 탐욕적 선택이 항상 최적이다.

파이썬 구현

change.py

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

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

for coin in coin_types:
    num_coins = n // coin    # 몫 = 이 동전을 몇 개 쓰는가
    count += num_coins
    n %= coin                # 나머지 = 남은 돈
    print(coin, "원:", num_coins, "개")

print("최소 동전 개수:", count)
```

내림차순 정렬이 전제

큰 단위부터 보지 않으면 그리디 자체가 성립하지 않는다.

몫으로 한 번에 처리

1개씩 빼지 않고 n // coin 으로 개수를 즉시 구한다.

나머지가 다음 문제

n %= coin 이 남은 금액이 되어 부분 문제로 넘어간다.

반복문 한 겹이 전부다

동전을 하나씩 빼는 대신 몫과 나머지 연산을 쓰면 금액이 1억 원이어도 반복 횟수는 동전 종류 수만큼으로 고정된다.

실행 결과

```
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
```

동전	개수	누적 금액
500원	2	1,000원
100원	2	1,200원
50원	1	1,250원
10원	1	1,260원
합계	**6**	**1,260원**

남은 돈이 정확히 0이 되며 종료된다. 되돌아간 단계는 한 번도 없다.

금액을 1,260원에서 126만 원으로 늘려도 반복은 여전히 네 번이다. 금액이 아니라 동전 종류 수가 비용을 결정한다.

§5.2 거스름돈 문제

복잡도 분석

이 알고리즘의 시간 복잡도는 O(K)이다. 여기서 K는 금액이 아니라 동전의 종류 개수를 뜻한다.

반복 횟수 for 문은 화폐 종류만큼만 정확히 반복된다. 거슬러 줄 금액과는 완전히 무관하다. **K회**

내부 연산 몫(//)과 나머지(%) 계산은 금액의 크기와 상관없이 한 번의 연산으로 끝난다. **O(1)**

전형적인 값 500·100·50·10원이면 K = 4. 실무에서 K는 대개 열 개 안쪽의 작은 상수이다. **K = 4**

$$T(n) = K \times O(1) = O(K)$$

금액 n이 커져도 수행 시간은 늘지 않는다 — 사실상 상수 시간에 가깝다.

충격적인 반례 — 400원짜리 동전의 등장

편의점 사장님이 500원과 100원 사이에 400원짜리 동전을 새로 도입했다고 하자. 이제 800원을 거슬러 준다.

그리디의 선택 — 동전 4개

가장 큰 500원을 덥석 집는다. 남은 300원은 100원 3개로만 채울 수 있다.

500 + 100 + 100 + 100 = 800원 (4개)

진짜 최적해 — 동전 2개

500원을 포기하고 400원 두 개를 쓰면 끝난다.

400 + 400 = 800원 (2개)

그리디는 이 조합을 아예 후보로 올리지 못했다.

500원을 고르는 순간 남은 돈이 300원이 되어 [400, 400]이라는 더 좋은 미래가 원천 봉쇄된다. 그리디의 성패는 화폐 체계에 달려 있다.

5.3

활동 선택 문제

회의실은 하나뿐인데 요청은 N건 들어왔다.
가장 많은 팀이 회의하도록 배정하려면?

5.3

§5.3 활동 선택 문제

회의실 하나, 요청은 여섯 건

회사의 유일한 대회의실을 관리한다고 하자. 오늘 하루 회의실을 쓰겠다는 요청이 N건 들어왔고, 각 요청에는 시작 시간과 종료 시간이 정해져 있다. 회의실은 한 번에 한 팀만 쓸 수 있고 도중에 끊을 수도 없다.

목표는 사용 시간의 합이 아니라 "회의를 마칠 수 있는 팀의 수"를 최대로 만드는 것이다.

최대 개수 ≠ 최대 사용 시간

무엇을 기준으로 골라야 가장 많은 팀을 넣을 수 있을까? 그럴듯해 보이는 기준이 여럿 있다.

실패하는 직관들

① 일찍 시작하는 회의부터

"부지런한 새가 벌레를 잡는다"는 직관이다. 그러나 9시에 시작해 저녁까지 이어지는 회의 하나가 하루 전체를 잡아먹으면, 뒤에 들어올 수 있었던 여러 팀이 통째로 밀려난다.

② 회의 시간이 짧은 것부터

짧은 회의를 먼저 넣으면 많이 들어갈 것 같다. 하지만 짧은 회의 하나가 앞뒤 두 회의의 경계에 걸쳐 있으면, 한 건을 넣으려다 두 건을 잃는다.

두 기준 모두 "남은 시간을 얼마나 아꼈는가"를 보지 않는다. 그래서 반례가 쉽게 만들어진다.

§5.3 활동 선택 문제

성공하는 전략 — 빨리 끝나는 녀석이 효자다

가장 확실한 그리디 기준은 종료 시간이다. 회의가 빨리 끝날수록 뒤에 남는 여유 시간이 길어지고, 그만큼 다른 회의를 받을 여지가 커진다. "남은 자원을 최대로 남긴다"는 그리디의 철학과 정확히 맞아떨어진다.

① 정렬 모든 회의를 종료 시간이 빠른 순서로 오름차순 정렬한다.

② 첫 선택 가장 먼저 끝나는 회의를 무조건 선택한다. 이 선택은 언제나 안전하다.

③ 이어 붙이기 직전 회의가 끝난 시각 이후에 시작하는 회의 중 가장 먼저 끝나는 것을 고른다.

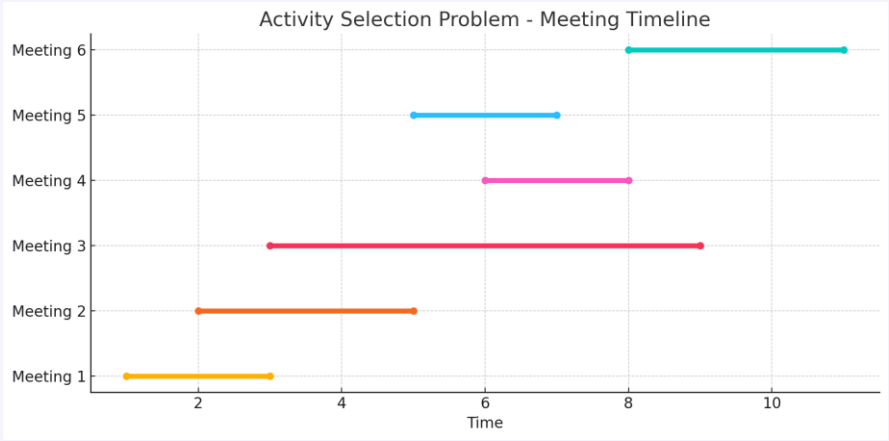
④ 반복 더 이상 넣을 회의가 없을 때까지 ③을 되풀이한다.

핵심은 "가장 빨리 비는 회의실"을 계속 유지하는 것이다.

예제 — 여섯 개의 회의 요청

N = 6, 각 회의는 (시작 시간, 종료 시간)으로 주어진다.

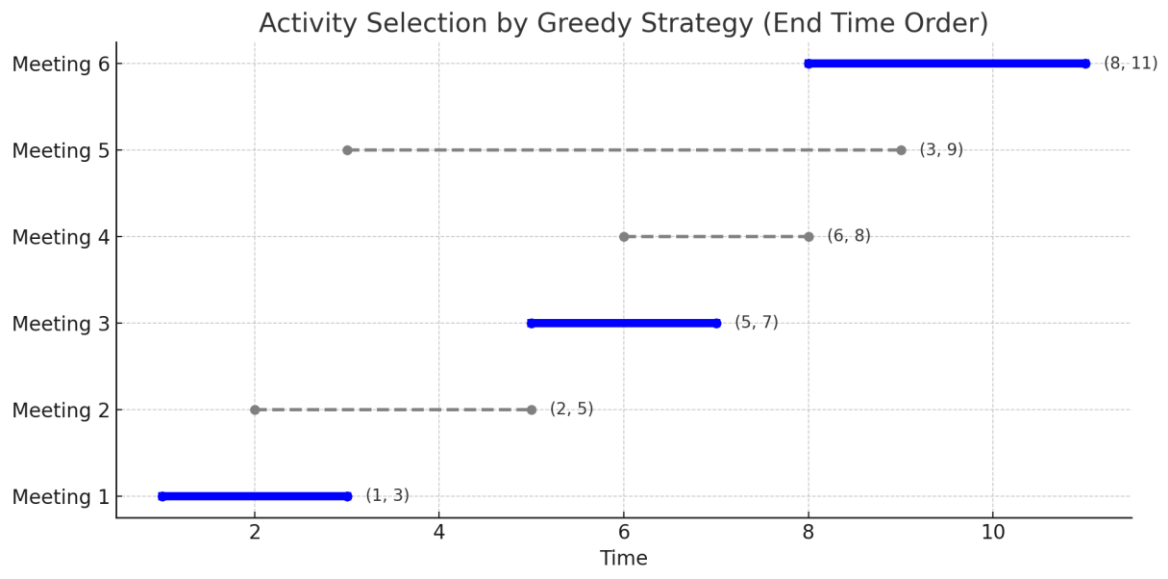
```
(1, 3)   (2, 5)   (3, 9)   (6, 8)   (5, 7)   (8, 11)
```



입력 순서 그대로는 어떤 조합이 최선인지 눈으로 가늠하기 어렵다. 먼저 줄을 세워야 한다.

종료 시간 기준으로 정렬한다

(1, 3) (2, 5) (5, 7) (6, 8) (3, 9) (8, 11)



✓	**(1, 3)**	가장 먼저 끝난다	**선택**
✗	**(2, 5)**	3시 이전에 시작	**제외**
✓	**(5, 7)**	3시 이후에 시작	**선택**
✗	**(6, 8)**	7시 이전에 시작	**제외**
✗	**(3, 9)**	7시 이전에 시작	**제외**
✓	**(8, 11)**	7시 이후에 시작	**선택**

선택된 회의는 (1,3), (5,7), (8,11) — 모두 세 건. 어떤 조합으로도 네 건은 넣을 수 없다.

활동 선택 알고리즘 (의사코드)

activity_selection.pseudo

```
// A: 활동들의 집합, 각 활동 a는 (start[a], finish[a])를 가진다

GREEDY_ACTIVITY_SELECTION(A):
    A를 finish 오름차순으로 정렬한다
    S <- 빈 집합
    lastFinish <- -무한대

    for each a in A (정렬된 순서대로):
        if start[a] >= lastFinish then
            S <- S + {a}
            lastFinish <- finish[a]

    return S
```

정렬이 전략의 전부

이 한 줄이 그리디 기준을 결정한다. 정렬 기준이 바뀌면 답이 무너진다.

lastFinish 하나로 충분

이전에 선택한 회의 전체를 기억할 필요 없이 마지막 종료 시각만 들고 다니면 된다.

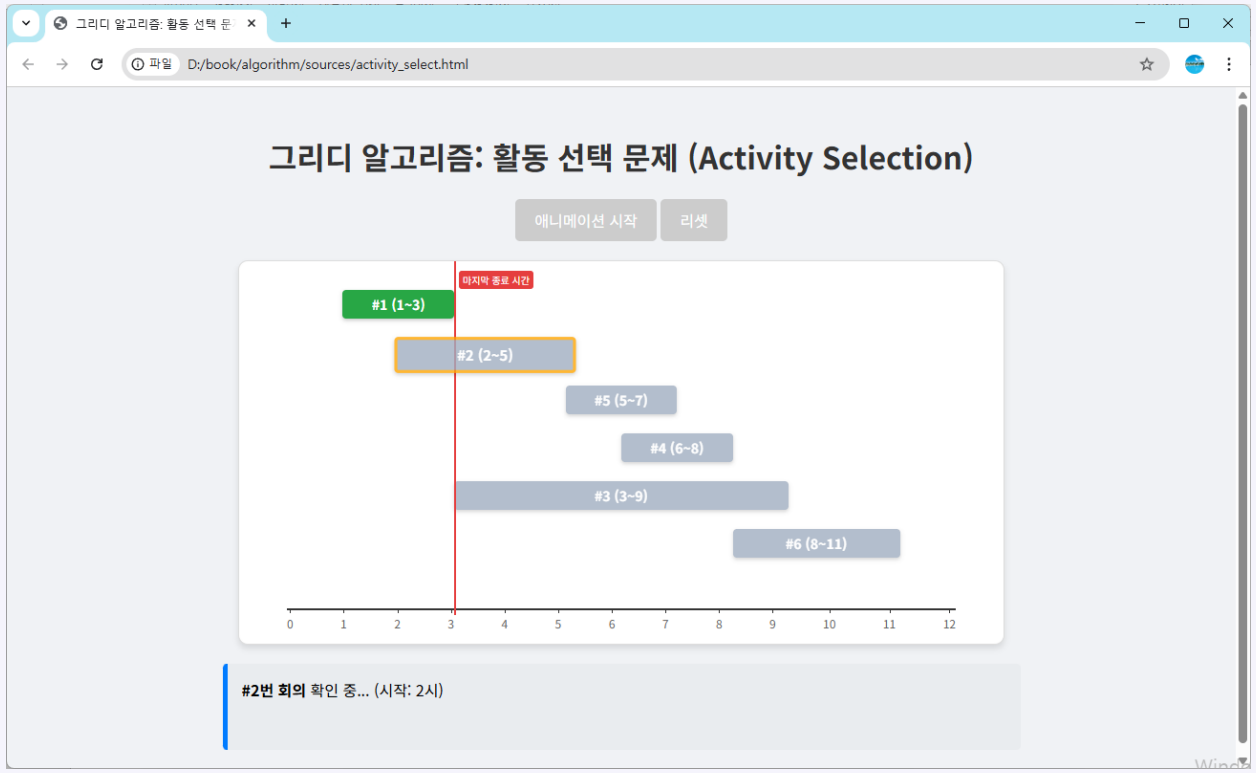
조건은 부등호 하나

start ≥ lastFinish 이면 겹치지 않는다. 같은 시각에 끝나고 시작하는 것은 허용된다.

정렬을 마친 뒤에는 배열을 한 번만 훑는다

되돌아가서 다시 검토하는 구간이 전혀 없다. 그래서 구현이 열 줄 남짓으로 끝난다.

동작 과정 한눈에 보기



① 3시에 첫 회의가 끝난다

(1,3)을 선택하고 lastFinish를 3으로 둔다.

② 겹치는 회의는 건너뛴다

(2,5)는 2시에 시작하므로 조건에 걸린다.

③ 5시 시작 회의를 채택

(5,7)이 조건을 만족한다. lastFinish는 7이 된다
.

④ 마지막으로 (8,11)

8시에 시작하므로 안전하게 들어간다. 총 세 건.

파이썬 구현

activity.py

```python
def solve_activity_selection(meetings):
    # meetings: (회의ID, 시작시간, 종료시간) 튜플의 리스트

    # 1단계: 종료 시간 기준 오름차순 정렬 – 그리디의 핵심
    meetings.sort(key=lambda x: x[2])

    # 가장 빨리 끝나는 회의는 무조건 선택한다
    selected = [meetings[0]]
    last_end = meetings[0][2]

    # 2단계: 두 번째 회의부터 순서대로 확인
    for i in range(1, len(meetings)):
        mid, start, end = meetings[i]

        # 3단계: 직전 회의가 끝난 뒤에 시작하면 채택
        if start >= last_end:
            selected.append(meetings[i])
            last_end = end

    return selected
```

key=lambda x: x[2]

튜플의 세 번째 값, 즉 종료 시간을 정렬 기준으로 삼는다.

첫 회의는 무조건

정렬 후 첫 원소는 항상 답에 포함된다. 증명이 뒷받침한다.

last_end 갱신

채택할 때마다 기준선을 옮긴다. 이 변수 하나가 상태의 전부다.

한 번의 순회

리스트를 앞에서 뒤로 한 번만 지나간다. 되돌아가지 않는다.

실행 결과

run.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("총 배정된 회의 수:", len(result), "개")
for m in result:
    print(m[0], "번 회의", m[1], "시 ~", m[2], "시")
```

OUTPUT

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

회의	시간	판정
1번	1시 ~ 3시	선택
2번	2시 ~ 5시	제외
5번	5시 ~ 7시	선택
4번	6시 ~ 8시	제외
3번	3시 ~ 9시	제외
6번	8시 ~ 11시	선택

여섯 건 중 세 건이 배정되었다. 종료 시간 정렬 덕분에 되돌아간 판단이 하나도 없다.

복잡도 분석

정렬	n개의 회의를 종료 시간 기준으로 정렬한다. 파이썬 sort()는 평균 O(n log n)이다.	`O(n log n)`
순회	정렬된 리스트를 앞에서 뒤로 한 번 훑으며 조건 하나를 검사한다.	`O(n)`
공간	선택된 회의 목록 외에 추가 자료구조가 필요 없다.	`O(n)`

$O(n \log n) + O(n) \Rightarrow O(n \log n)$

전체 비용은 정렬 단계가 지배한다 — 그리디 자체는 사실상 공짜다.

5.4

배낭 문제

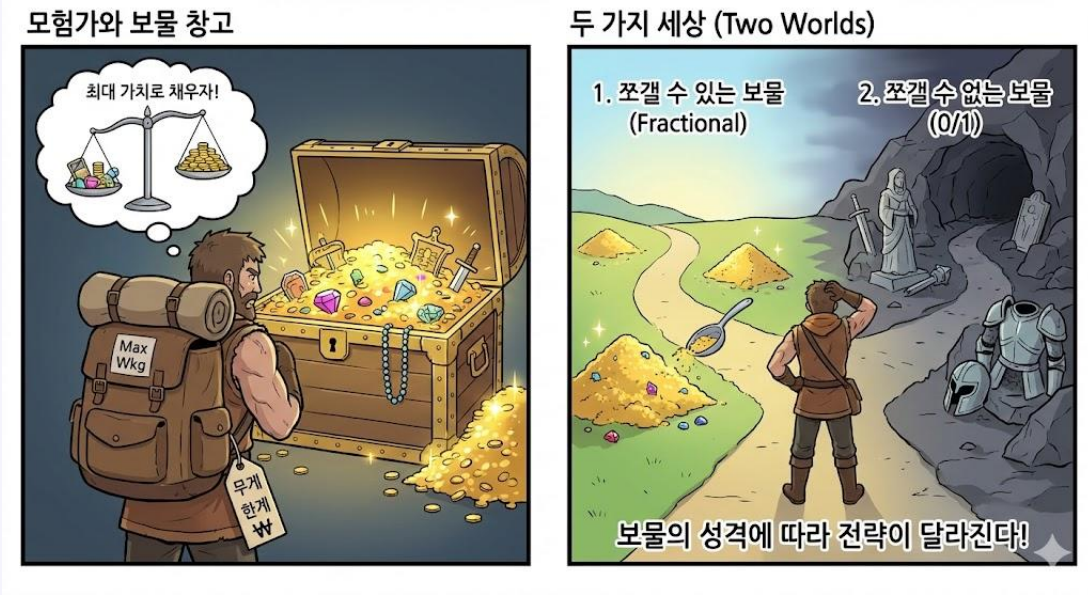
보물을 쪼갤 수 있느냐 없느냐,
단 하나의 차이가 그리디의 운명을 가른다.

5.4

§5.4 배낭 문제

배낭에 무엇을 담을 것인가

용량이 정해진 배낭 하나를 메고 보물 창고에 들어섰다. 보물마다 무게와 가치가 다르다. 배낭이 감당할 수 있는 무게 안에서 가치의 합을 최대로 만드는 것이 배낭 문제(knapsack problem)이다.



이 문제는 보물을 쪼갤 수 있는가에 따라 완전히 다른 두 문제로 갈라진다.

용량 W = 30kg

쪼갤 수 있으면 그리디가 최적해를 보장하고, 쪼갤 수 없으면 보장하지 못한다.

① 쪼갤 수 있는 배낭 문제 (fractional knapsack)

첫 번째 시나리오의 보물은 사금, 마법 가루, 희귀 향신료처럼 원하는 만큼 덜어낼 수 있는 형태다. 배낭 용량은 30kg이다.

보물 이름	총 무게	총 가치	무게당 가치(가성비)
A (황금 가루)	20kg	14,000 골드	700 골드/kg (1등!)
B (은 가루)	15kg	9,000 골드	600 골드/kg
C (보석 가루)	15kg	9,000 골드	600 골드/kg

보물	무게	가치	무게당 가치
A (황금 가루)	20kg	14,000G	**700G/kg**
B (은 가루)	15kg	9,000G	600G/kg
C (보석 가루)	15kg	9,000G	600G/kg

고를 기준은 총 가치가 아니라 무게당 가치, 곧 가성비다.

§5.4 배낭 문제

가성비가 높은 순서대로 쓸어 담는다

1kg당 가치가 가장 비싼 것부터 꾹꾹 눌러 담고, 공간이 부족해지면 남은 공간만큼만 덜어서 담는다.

A 전체 — 가성비 1등인 A(700G/kg)를 먼저 본다. 20kg 전부가 배낭에 들어간다. 남은 용량 10kg. → **14,000G**

B 일부 — B는 총 15kg이지만 남은 공간이 10kg뿐이다. 딱 10kg만 덜어 담는다 (600G × 10kg). → **6,000G**

C 제외 — 배낭이 가득 찼으므로 남은 보물은 볼 필요조차 없다. 여기서 반복이 끝난다. → **0G**

20kg + 10kg = 30kg · 14,000 + 6,000 = 20,000 G

빈 공간 없이 가장 비싼 것들로만 채웠다 — 쪼갤 수 있다면 그리디는 언제나 최적해다.

② 0/1 배낭 문제 (0/1 knapsack)

이번에 발견한 보물은 황금 조각상, 고대 마법서, 미스릴 갑옷처럼 단단한 고체다. 덜어낼 수 없으니 배낭에 넣거나(1) 포기하거나(0), 둘 중 하나만 가능하다.

"조금만 담기"라는 선택지가 사라지는 순간, 가성비 순서는 더 이상 안전한 기준이 아니다.

그리디의 답 — 14,000G

가성비 1등 A(20kg)를 담으면 남은 공간은 10kg. B도 C도 15kg이라 더 넣을 수 없다. 10kg를 텅 빈 채로 들고 나온다.

§5.4 배낭 문제

만약 A를 포기했다면?

가성비 1등 A를 일부러 버리고, 차순위인 B(15kg)와 C(15kg)를 함께 담아 본다.

그리디의 선택

A 하나만 담는다 (20kg)

14,000 G

배낭에 10kg의 빈 공간이 남는다

진짜 최적해

B + C를 담는다 (15kg + 15kg)

18,000 G

30kg를 빈틈없이 가득 채운다

4,000골드의 차이 — 그리디는 이 조합을 후보로도 올리지 못한다

보물을 쪼갤 수 없는 0/1 배낭 문제에서 그리디는 최적해를 보장하지 못한다. "지금 가장 비싼 것"이 "전체에서 가장 좋은 조합"과 어긋나기 때문이다.

쪼갤 수 있는 배낭 알고리즘 (의사코드)

fractional_knapsack.pseudo

```
// items: 각 원소가 (weight, value)를 가진 리스트
// W: 배낭의 최대 용량
FRACTIONAL_KNAPSACK(items, W):

    for each item in items do
        item.ratio <- item.value / item.weight    // 무게당 가치

    sort items by ratio in descending order

    totalValue <- 0
    remaining  <- W

    for each item in items do
        if remaining = 0 then break
        if item.weight <= remaining then
            take <- item.weight          // 통째로 담는다
            frac <- 1
        else
            take <- remaining            // 남은 만큼만 쪼갠다
            frac <- remaining / item.weight

        totalValue <- totalValue + item.value * frac
        remaining  <- remaining - take

    return totalValue
```

ratio가 유일한 기준

총 가치가 아니라 무게당 가치로 줄을 세운다. 이 정렬이 그리디의 전부다.

frac로 쪼갠다

통째로 못 넣으면 남은 공간 비율만큼만 가치를 더한다. 이 한 줄이 0/1과의 차이다.

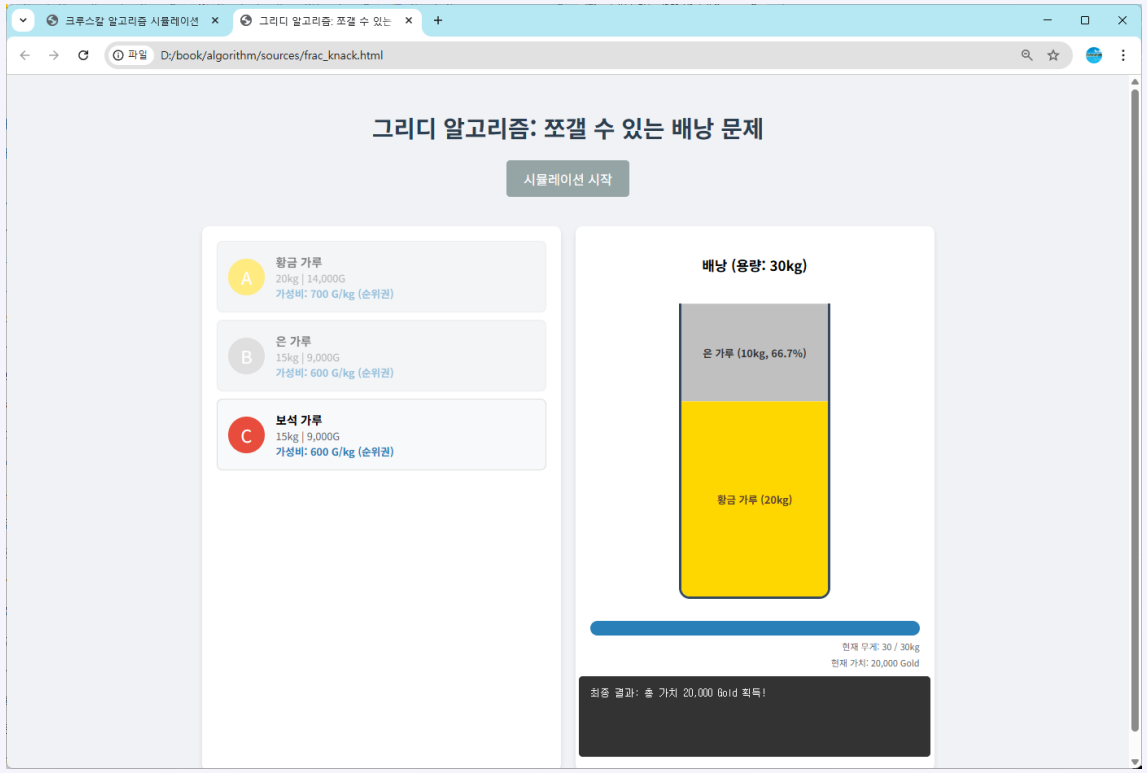
remaining = 0이면 종료

배낭이 가득 차는 순간 남은 보물은 살펴볼 필요가 없다.

되돌아가지 않는다

한 번 담은 것을 다시 꺼내는 분기가 코드에 존재하지 않는다.

동작 과정 한눈에 보기



① 가성비로 줄 세우기

A(700) → B(600) → C(600) 순서가 정해진다.

② A를 통째로

20kg ≤ 30kg 이므로 전부 담는다. 남은 용량 10kg.

③ B를 10kg만

15kg는 들어가지 않으니 2/3만 덜어 담는다.

④ 배낭이 가득 찬다

총 30kg, 20,000골드로 반복이 종료된다.

파이썬 구현

knapsack.py

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

def solve_fractional_knapsack(capacity, items):
    # 1단계: 가성비 내림차순 정렬 — 그리디의 핵심
    items.sort(key=lambda x: x.ratio, reverse=True)

    total_value = 0.0
    current_weight = 0.0

    # 2단계: 정렬된 순서대로 배낭 채우기
    for item in items:
        if current_weight + item.weight <= capacity:
            current_weight += item.weight   # 통째로
            total_value += item.value
        else:
            room = capacity - current_weight # 남은 공간
            total_value += room * item.ratio
            current_weight += room
            break

    return total_value
```

ratio를 생성자에서

객체를 만들 때 가성비를 미리 계산해 두면 정렬 키가 단순해진다.

reverse=True

내림차순이어야 비싼 것부터 본다. 이 인자를 빼면 답이 뒤집힌다.

room × ratio

쪼개 담은 부분의 가치는 남은 공간에 가성비를 곱해 구한다.

break로 종료

한 번 쪼개는 순간 배낭은 가득 찬다. 더 볼 이유가 없다.

§5.4 배낭 문제

실행 결과와 시간 복잡도

```
OUTPUT
--- 배낭 채우기 시작 (용량: 30kg) ---
-> A(황금 가루) 전체 추가 (현재 무게: 20.0kg)
-> B(은 가루) 10.0kg만 쪼개서 추가 (가득 참!)
----------------------------------
배낭의 최대 가치 합계: 20000.0 골드
```

정렬	가성비 기준 정렬	O(n log n)
순회	리스트를 한 번 훑으며 채우기	O(n)
연산	각 단계는 사칙연산뿐	O(1)

A 100% + B 66.7% = 30kg, 20,000골드. 이론값과 정확히 일치한다.

O(n log n)

여기서 n은 보물 아이템의 개수다. 아이템이 백만 개여도 정렬 한 번이면 최적해가 나온다 — 단, 쪼갤 수 있을 때만.

§5.4 배낭 문제

그리디의 실패, 그리고 새로운 희망

우리는 0/1 배낭 문제를 통해 그리디의 한계를 분명히 목격했다. 미래의 조합을 내다보지 않고 현재의 가성비만 따지는 방식으로는, 쪼갤 수 없는 보물들의 조합 문제를 풀 수 없다.

그리디가 못 하는 것

"넣었을 때"와 "넣지 않았을 때"를 동시에 비교하지 못한다. 갈래를 나누는 사고 자체가 없다.

필요한 것은 경우의 수

일직선으로 전진하는 대신 여러 갈래를 모두 계산해 두었다가 최선을 고르는 방식이 필요하다.

6장 — 동적 계획법

문제를 작은 조각으로 나눠 표를 채워 나가는 방법이다. 0/1 배낭의 진짜 정답은 여기서 나온다.

그리디가 포기한 문제가 풀 수 없는 문제인 것은 아니다

한계를 정확히 아는 것이 다음 도구로 넘어가는 출발점이다. 0/1 배낭 문제는 6장 동적 계획법에서 다시 만난다.

5.5

그리디가 통하는 조건

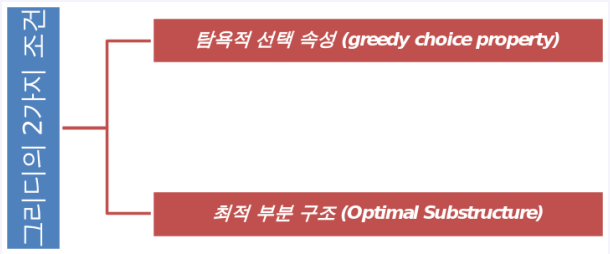
5.5

그리디가 최적해를 보장하려면
두 가지 조건이 반드시 성립해야 한다.

⚖ §5.5 그리디의 두 조건

그리디가 통하기 위한 두 가지 조건

알고리즘 이론은 그리디 전략이 수학적으로 최적해를 보장하기 위해 갖춰야 할 조건 두 가지를 제시한다.



① 탐욕적 선택 속성

greedy choice property

지금 가장 좋아 보이는 선택을 해도 전체 최적해를 놓치지 않는다는 보장.

② 최적 부분 구조

optimal substructure

전체 문제의 최적해가 부분 문제들의 최적해 조합으로 이루어진다는 성질.

둘 중 하나라도 성립하지 않으면 그리디는 최적해를 보장하지 못한다 — 400원 동전과 0/1 배낭이 그 증거다.

① 탐욕적 선택 속성 (greedy choice property)

문제를 풀 때 지금 당장 가장 좋아 보이는 선택을 해도, 그 선택 때문에 전체 최적해를 놓치지는 않는다는 보장이다. 다시 말해 "처음에 하는 가장 좋아 보이는 선택이 정답으로 가는 길에서도 언제나 안전하다"는 성질이다.

핵심 단어는 "안전"이다. 더 좋다는 뜻이 아니라, 손해 볼 것이 없다는 뜻이다.

증명은 대개 교환 논법으로

최적해가 그리디의 선택을 포함하지 않는다고 가정한 뒤, 바꿔 넣어도 손해가 없음을 보인다.

실제 예 — 안전한 선택과 위험한 선택

성공 — 활동 선택 문제

가장 빨리 끝나는 회의를 고르면 남은 시간이 최대가 된다. 다른 어떤 선택보다 손해 볼 것이 없으므로 언제나 안전하다.

실패 — 비표준 동전

[500, 400, 100]으로 800원을 만들 때 500원을 고르는 순간 남은 돈이 300원이 되어 [400, 400]이라는 더 좋은 미래가 사라진다.

판별 기준 "이 선택을 포함하는 최적해가 반드시 존재하는가?" 이 질문에 예라고 답할 수 있으면 탐욕적 선택 속성이 성립한다.

같은 그리디 전략도 문제의 구조에 따라 안전해지기도 하고 위험해지기도 한다

전략이 문제를 고르는 것이 아니라, 문제의 구조가 전략의 정당성을 결정한다. 그래서 그리디는 항상 "왜 이 선택이 안전한가"를 함께 물어야 한다.

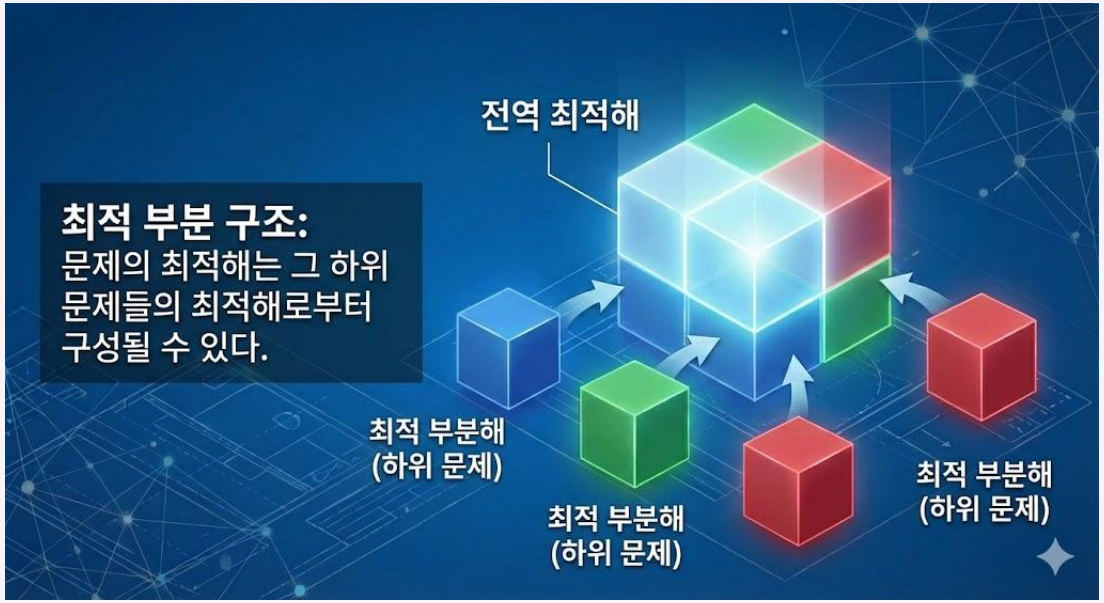
② 최적 부분 구조 (optimal substructure)

전체 문제의 최적해가 부분 문제들의 최적해 조합으로 이루어진다는 성질이다. 그리디는 한 번 선택하면 그것을 고정하고, 남은 문제를 더 작은 같은 종류의 문제로 바꿔서 계속 푼다. 이때 그 작은 문제의 최적해가 전체 최적해의 일부여야만 이 방식이 성립한다.

선택 고정 → 남은 문제도 같은 문제

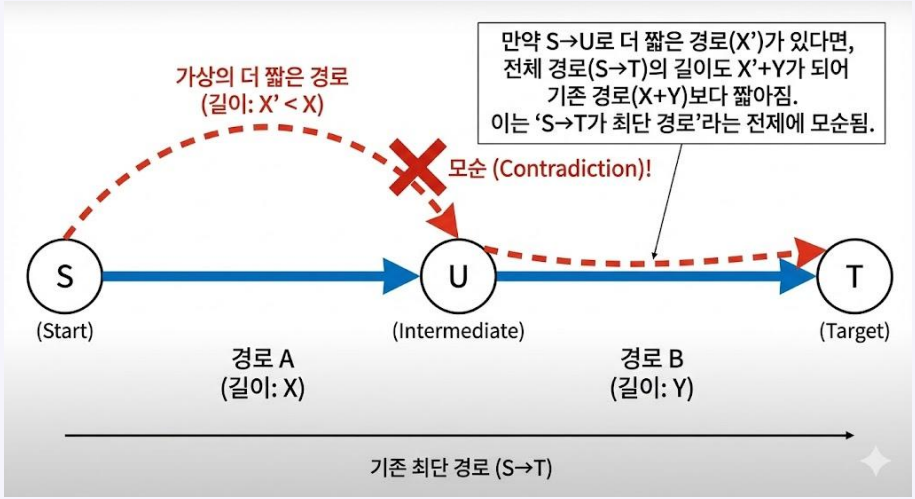
분할 정복·DP와 공유하는 성질

최적 부분 구조는 그리디만의 것이 아니다. 4장의 분할 정복도, 6장의 동적 계획법도 같은 성질 위에 서 있다.



성공 사례 — 최단 경로 문제

최단 경로에서는 전체 최단 경로의 어떤 부분 경로를 떼어 내도, 그것이 해당 구간의 최단 경로가 된다. 전체 최적해가 부분 최적해로 구성되므로 최적 부분 구조가 성립한다.



부분 경로도 최단이다

A→D 최단 경로가 A→B→C→D라면, 그 안의 A→B→C 역시 A에서 C까지의 최단 경로다.

귀류법으로 바로 보인다

더 짧은 A→C 경로가 있다면 그것으로 갈아 끼워 전체를 더 줄일 수 있다. 이는 A→D가 최단이라는 전제와 모순이다.

그래서 다익스트라 알고리즘(§5.8)이 그리디 방식으로 최단 경로를 정확히 구할 수 있다.

실패 사례 — 외판원 순회 문제(TSP)

최적 부분 구조가 무너지는 대표적인 예가 외판원 순회 문제다. 최적 순회 경로가 A → B → C → D → A 라고 하자.

A → B → C → D → A

1

부분만 떼어 보면?

중간 구간 A → B → C 가 A에서 C까지의 최단 경로일까? 그렇지 않을 수 있다.

2

더 짧은 길이 있어도

A → C 로 곧장 가는 더 짧은 길이 존재할 수 있다. 하지만 그 길을 택하면 D를 방문할 수 없게 된다.

3

전체가 망가진다

남은 도시를 모두 들르고 돌아오는 조건 때문에, 부분에서 최적이 아닌 경로가 전체에서는 최적일 수 있다.

⚠ 부분 문제의 최적해를 모아도 전체 최적해가 되지 않는다 — 최적 부분 구조의 붕괴다.

5.6

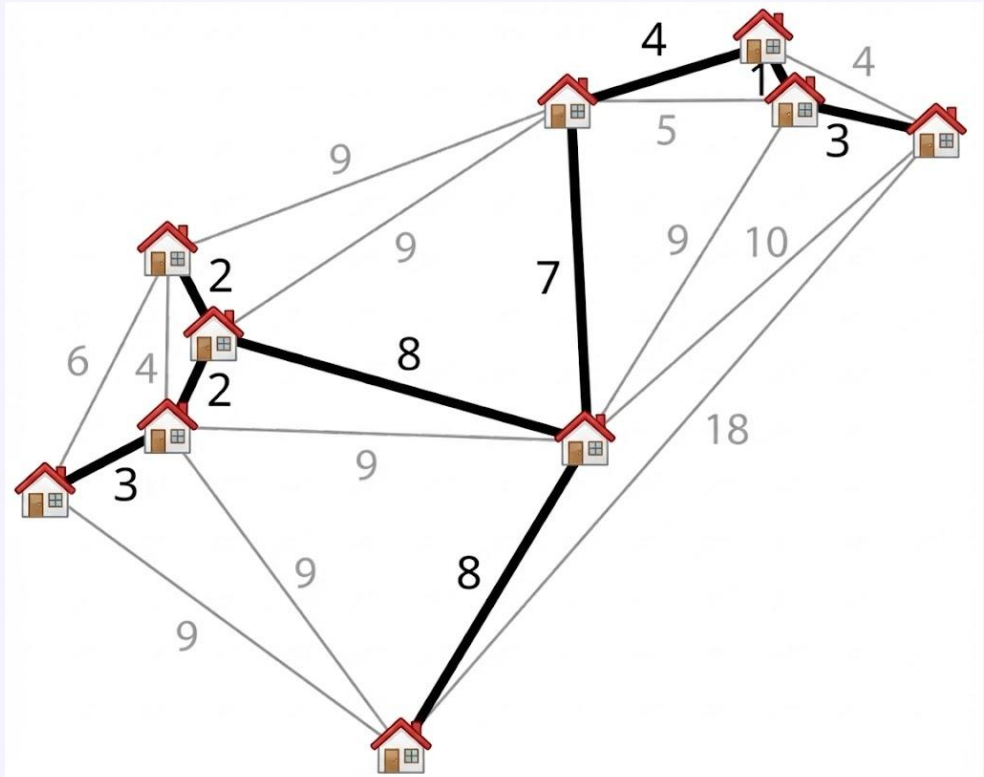
크루스칼 알고리즘

모든 도시를 잇되 도로 비용은 최소로 —
가장 싼 간선부터 하나씩 골라 나간다.

5.6

최소 신장 트리 (Minimum Spanning Tree)

여러 도시를 도로로 연결하려 한다. 모든 도시가 서로 오갈 수 있어야 하되, 전체 공사비는 최소여야 한다. 이렇게 모든 정점을 연결하면서 간선 가중치의 합이 최소가 되는 간선 집합을 최소 신장 트리라고 한다.



간선은 정확히 V-1개

정점이 V개면 트리를 이루는 간선은 항상 V-1개다. 하나라도 더 넣으면 사이클이 생긴다.

사이클은 금지

사이클이 있다는 것은 없어도 되는 도로가 있다는 뜻이다. 비용만 늘어난다.

모두 연결되어야 한다

어느 도시도 고립되면 안 된다. 끊긴 그래프는 신장 트리가 아니다.

크루스칼 알고리즘 — 가장 저렴한 간선부터

전체 그래프에서 가장 싼 간선부터 하나씩 집어 들면서, 사이클을 만들지 않는 것만 골라 트리에 추가한다.

① 간선을 가중치 오름차순 정렬

전체 간선을 싼 것부터 줄 세운다. 이 정렬이 그리디의 기준이다.

② 사이클을 만들지 않으면 채택

두 정점이 이미 같은 덩어리에 속해 있으면 그 간선은 버린다.

③ 간선 V-1개가 될 때까지

트리가 완성되는 순간 남은 간선은 볼 필요가 없다.

핵심 판정은 "사이클이 생기는가" 하나뿐이다

이 질문에 빠르게 답하기 위해 서로소 집합(union-find) 자료구조를 쓴다. 두 정점의 대표가 같으면 이미 연결되어 있다는 뜻이다.

크루스칼 알고리즘 (의사코드)

kruskal.pseudo

```
// 입력: 그래프 G = (V, E)
// 출력: 최소 신장 트리를 구성하는 간선들의 집합
KRUSKAL(G):

    MST <- 공집합
    for each vertex v in V:
        Make-Set(v)        // 모든 정점을 독립된 집합으로

    sort E in increasing order by weight   // 그리디의 핵심

    for each edge (u, v) in E (정렬된 순서대로):
        if Find-Set(u) != Find-Set(v) then
            MST <- MST + {(u, v)}   // 사이클이 없으면 채택
            Union(u, v)             // 두 집합을 합친다

    return MST
```

Make-Set

처음에는 모든 정점이 각자 혼자다. 부모를 자기 자신으로 둔다.

Find-Set

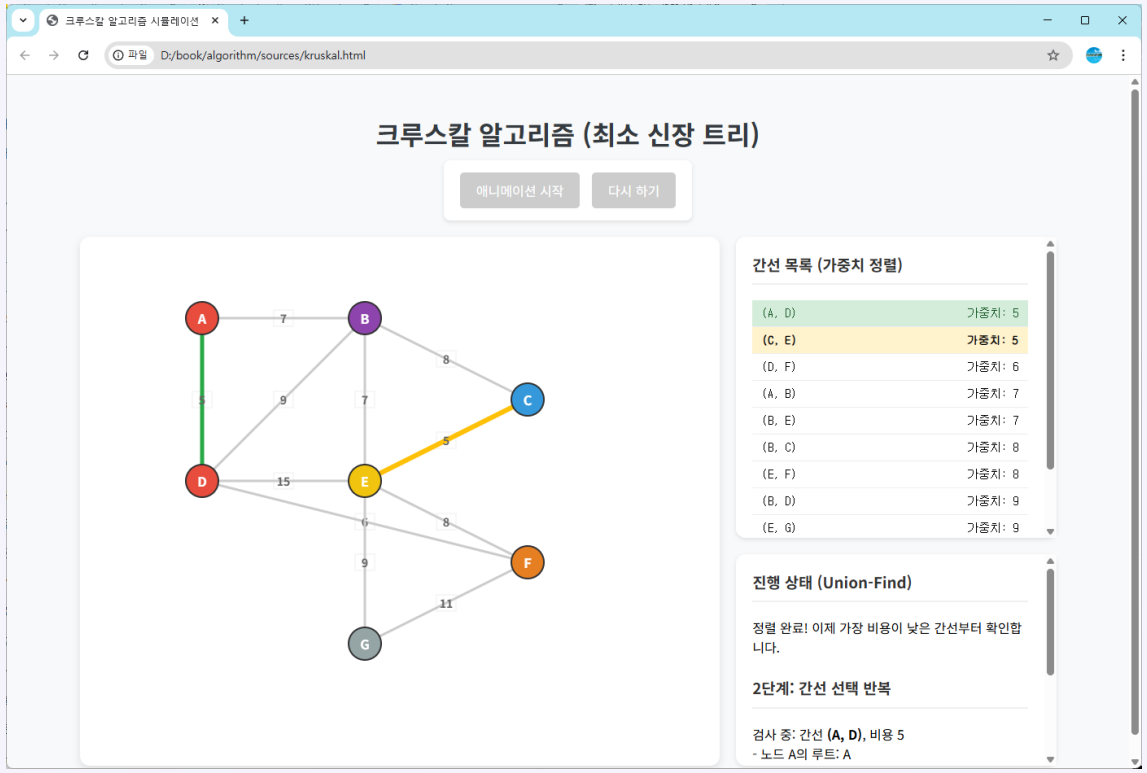
어느 덩어리에 속하는지 대표를 찾는다. 두 대표가 같으면 사이클이다.

Union

서로 다른 두 덩어리를 하나로 합친다. 연결 관계가 갱신된다.

정렬 한 번 + 사이클 검사 반복 — 크루스칼의 전부다.

동작 과정 한눈에 보기



✓	10	a — f 채택	✓
✓	12	c — d 채택	✓
✓	15	b — a 채택	✓
✓	16	b — c 채택	✓
✗	18	d — a 사이클	✗
✓	22	d — e 채택	✓

간선 다섯 개(정점 6개 − 1)를 채택한 순간 트리가 완성된다. 총 비용 10+12+15+16+22 = 75.

서로소 집합 (Disjoint Set, Union-Find)

disjoint_set.py

```python
class DisjointSet:
    def __init__(self, n):
        # 처음에는 모든 정점이 각자 독립된 집합
        self.parent = list(range(n))

    def find(self, x):
        # x가 속한 집합의 대표(루트)를 찾는다
        if self.parent[x] != x:
            # 경로 압축: 찾은 루트를 곧바로 연결해 둔다
            self.parent[x] = self.find(self.parent[x])
        return self.parent[x]

    def union(self, x, y):
        px, py = self.find(x), self.find(y)
        if px == py:
            return False    # 이미 같은 집합 -> 사이클
        self.parent[py] = px
        return True
```

대표 하나로 판정

두 정점의 루트가 같으면 이미 이어져 있다는 뜻이다. 사이클 검사가 비교 한 번으로 끝난다.

경로 압축

탐색 중 만난 노드를 루트에 직접 붙여 두면 다음 탐색이 거의 상수 시간이 된다.

union의 반환값

합쳐졌으면 True, 이미 같은 집합이면 False. 이 값이 그대로 채택 여부가 된다.

경로 압축과 랭크 최적화를 함께 쓰면 연산 하나당 평균 $O(\alpha(V))$ — 사실상 상수 시간이다.

파이썬 구현

kruskal.py

```python
def kruskal(n, edges):
    # (1) 간선을 가중치 오름차순 정렬
    #     튜플이 (가중치, u, v) 순이므로 그냥 sort()면 된다
    edges.sort()

    # (2) 서로소 집합 초기화
    ds = DisjointSet(n)
    mst = []

    # (3) 싼 간선부터 하나씩 확인
    for weight, u, v in edges:
        # 사이클이 생기지 않을 때만 채택
        if ds.union(u, v):
            mst.append((u, v, weight))

        # MST는 간선 (정점 수 - 1)개면 완성
        if len(mst) == n - 1:
            break

    return mst
```

run.py

```python
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, 0, 1)
]

mst = kruskal(6, edges)
for u, v, w in mst:
    print(u, "--", v, "(가중치", w, ")")
```

튜플 정렬의 요령

가중치를 튜플 맨 앞에 두면 별도 key 없이 sort()만으로 원하는 순서가 나온다.

조기 종료

간선 V-1개를 모으면 남은 간선은 확인할 필요가 없다.

실행 결과와 시간 복잡도

```
OUTPUT
최소 신장 트리 구성:
0 -- 5 (가중치 10)
2 -- 3 (가중치 12)
1 -- 0 (가중치 15)
1 -- 2 (가중치 16)
3 -- 4 (가중치 22)
```

간선 5개, 총 비용 75. 정점 6개를 모두 잇는 가장 싼 도로망이다.

정렬	간선 E개를 가중치 순으로	O(E log E)
탐색	find / union 연산	O(α(V))
구성	간선 V-1개를 선택	V-1회

O(E log E)

크루스칼은 탐욕적 선택 속성과 최적 부분 구조를 모두 만족한다. 그래서 그리디이면서도 언제나 최적해를 준다.

5.7

TSP와 그리디

"일단 지금 있는 곳에서 가장 가까운 도시로!"
가장 단순한 그리디 접근이 어디까지 통할까.

5.7

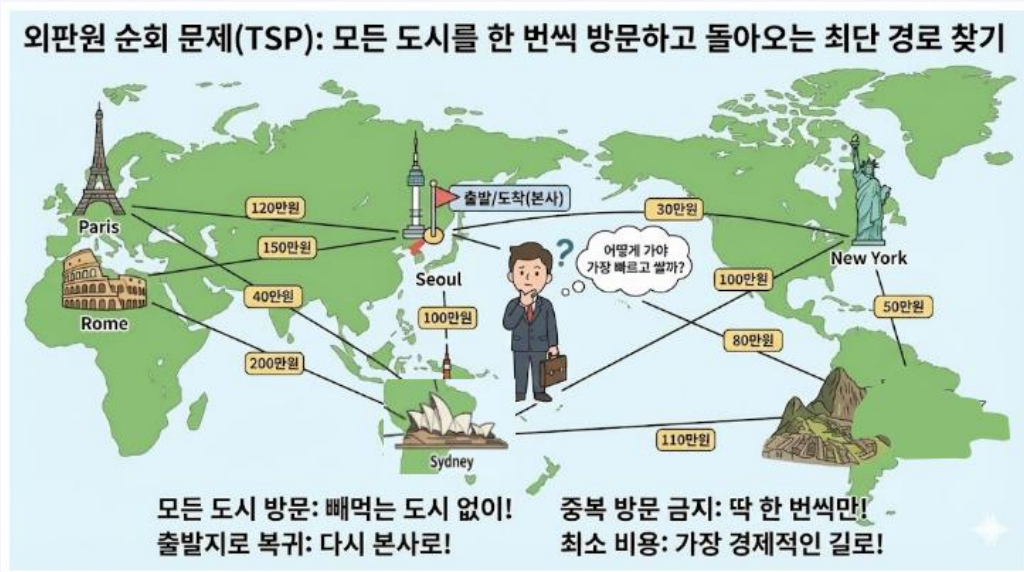
외판원 순회 문제와 가장 가까운 이웃

외판원 순회 문제(TSP)는 모든 도시를 한 번씩 방문하고 출발지로 돌아오는 최소 비용 경로를 찾는 문제다. 가장 단순한 그리디 접근은 "지금 있는 곳에서 가장 가까운 도시로 이동하자"이며, 이를 가장 가까운 이웃(nearest neighbor) 기법이라 부른다.

구현은 짧고 실행은 빠르다. 문제는 정확성이다.

TSP는 NP-난해 문제다

도시가 늘어나면 모든 경로를 다 따져 보는 방식은 현실적으로 불가능하다. 그래서 근사해를 빠르게 얻는 그리디가 실무에서 자주 쓰인다.



가장 가까운 이웃 알고리즘 (의사코드)

nearest_neighbor.pseudo

```
// start: 출발 도시, C: 전체 도시 집합
GREEDY_NEAREST_NEIGHBOR(start, C):

    current <- start
    V <- {start}                  // 방문한 도시 집합

    while |V| < |C| do
        // 아직 방문하지 않은 도시 중 가장 가까운 곳
        next <- dist(current, c)가 최소인 c (c not in V)

        V <- V + {next}           // 방문 처리
        current <- next           // 현재 위치 갱신

    return 방문 순서 + start      // 출발지로 복귀
```

기준은 거리 하나

남은 도시 중 가장 가까운 곳만 본다. 그 뒤에 무엇이 남는지는 따지지 않는다.

출발지에 따라 답이 달라진다

같은 그래프라도 어디서 출발하느냐에 따라 전혀 다른 경로가 나온다.

빠른 대신 근사

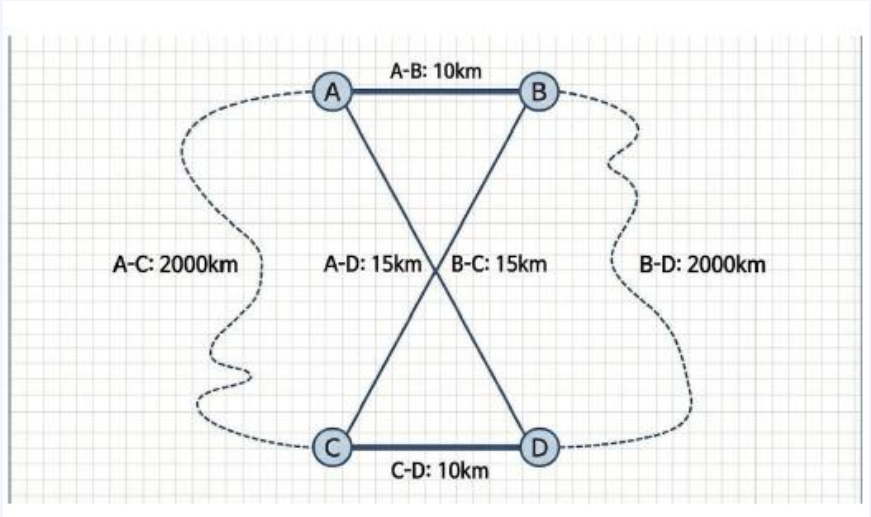
도시 수 n에 대해 $O(n^2)$이면 끝난다. 대신 최적해라는 보장은 없다.

마지막 한 걸음이 문제다

남은 도시를 모두 돌고 나면 출발지로 반드시 돌아와야 한다. 그런데 그 복귀 비용은 선택 과정에서 한 번도 고려되지 않았다.

성공 사례 — 운이 좋을 때

A에서 출발한다고 하자. 가까운 곳을 차례로 따라가면 자연스럽게 모든 도시를 훑고 돌아오게 된다.



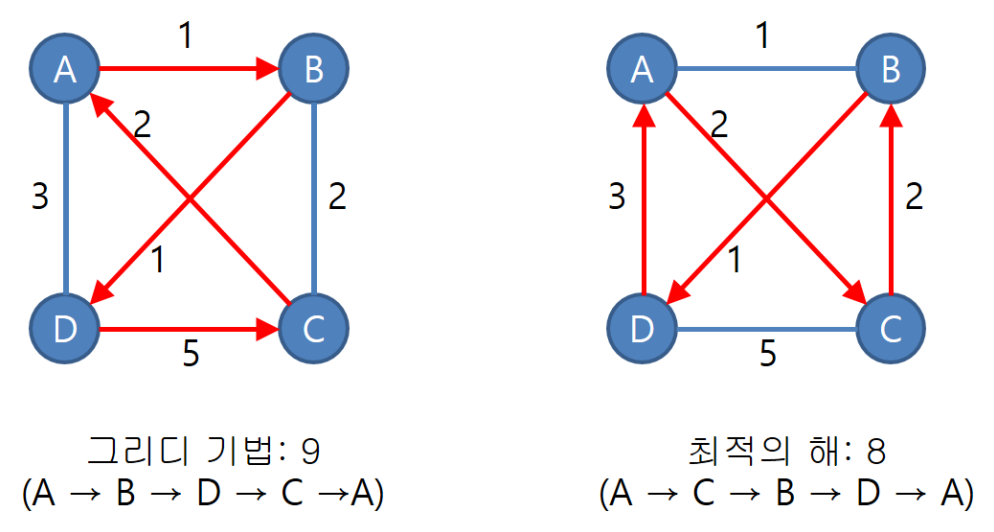
A → B	A에서 가장 가까운 곳은 B	10
B → C	남은 C, D 중 C가 가깝다	15
C → D	남은 도시는 D 하나뿐	10
D → A	출발지로 복귀한다	15

$$\mathbf{A \to B \to C \to D \to A \;=\; 50}$$

이 그래프에서는 그리디가 실제 최적 경로와 일치했다 — 하지만 그것은 보장이 아니라 우연이다.

실패 사례 — 마지막 한 걸음의 함정

가까운 도시만 좇다 보면, 마지막에 출발지로 돌아오는 길이 터무니없이 멀어지는 상황이 생긴다.



그리디 기법: 9
(A → B → D → C →A)

최적의 해: 8
(A → C → B → D → A)

앞부분은 계속 저렴하다

매 단계 가장 싼 간선을 고르므로 초반 비용은 확실히 낮다. 문제는 그 선택들이 마지막에 어떤 상황을 남기는지 아무도 보지 않는다는 점이다.

복귀 비용이 폭발한다

남겨진 마지막 도시가 출발지에서 아주 멀면, 앞에서 아낀 것을 한 번에 다 토해 낸다. 최적 부분 구조가 성립하지 않는다는 §5.5의 결론이 여기서 그대로 드러난다.

TSP에서 가장 가까운 이웃은 근사 알고리즘이다

빠르게 그럴듯한 답을 얻는 용도로는 쓸 만하지만, 최적해를 보장하지는 않는다. 정확한 해가 필요하면 동적 계획법이나 분기 한정법 같은 다른 도구를 써야 한다.

파이썬 구현과 실행 결과

tsp.py

```python
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

    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] != 0:
                if 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
```

run.py

```python
matrix = [
    [0,  10, 15, 20],   # A
    [10, 0,  35, 25],   # B
    [15, 35, 0,  30],   # C
    [20, 25, 30, 0]     # D
]
path, cost = solve_tsp_nearest_neighbor(matrix, 0)
```

OUTPUT

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

이중 반복문 = $O(n^2)$

매 단계마다 방문하지 않은 도시를 전부 훑어 최솟값을 찾는다.

5.8

다익스트라 알고리즘

"아직 방문하지 않은 곳 중 가장 가까운 곳은
이미 확정된 최단 거리다."

§5.8 다익스트라

한 지점에서 모든 지점까지의 최단 경로

한 지점에서 다른 모든 지점까지의 최단 경로를 찾는 다익스트라(Dijkstra) 알고리즘은 가장 널리 쓰이는 그리디 알고리즘이다. 시작점에서 출발해 연결된 이웃까지의 거리를 계산하고, 그중 가장 가까운 곳을 하나씩 확정해 나간다.

내비게이션의 경로 탐색, 네트워크 라우팅이 모두 이 알고리즘 위에 서 있다.

가장 가까운 곳부터 확정한다

§5.8 다익스트라

핵심 아이디어 — 왜 그 선택이 안전한가

집에서 편의점까지 2분, 학교까지 10분이라고 하자. 혹시 학교를 거쳐 편의점으로 돌아오는 길이 더 빠를 수는 없을까?

1

상식적으로 불가능하다

학교로 가는 순간 이미 10분을 썼다. 거기서 아무리 빨라도 2분보다 짧아질 수 없다.

2

음수 가중치가 없다면

간선 비용이 모두 0 이상이면 돌아가는 경로가 직행보다 짧아지는 일은 생기지 않는다.

3

곧 탐욕적 선택 속성

지금 가장 가까운 곳을 고르는 판단은 미래의 어떤 정보로도 뒤집히지 않는다. §5.5의 조건 ①이다.

전제 조건 가중치가 음수인 간선이 하나라도 있으면 이 논리가 무너진다. 그때는 벨만-포드 같은 다른 알고리즘을 써야 한다.

한 번 확정한 거리는 다시 계산하지 않는다 — 되돌아보지 않는 그리디의 성격이 그대로 나타난다.

다익스트라 알고리즘 (의사코드)

dijkstra.pseudo

```
// 입력: 그래프 G = (V, E), 시작 정점 start
// 출력: start에서 모든 정점까지의 최단 거리 dist
DIJKSTRA(G, start):

    for each vertex v in V:
        dist[v] <- 무한대
        visited[v] <- False
    dist[start] <- 0

    repeat |V| times:
        // [그리디 선택] 미방문 정점 중 dist가 가장 작은 것
        u <- 최소 dist를 갖는 미방문 정점
        if u is null or dist[u] = 무한대 then break
        visited[u] <- True        // 최단 거리 확정

        // [갱신] u를 거쳐 가는 길이 더 짧으면 고친다
        for each neighbor v of u:
            if not visited[v] then
                new_cost <- dist[u] + weight(u, v)
                if new_cost < dist[v] then
                    dist[v] <- new_cost

    return dist
```

그리디 선택

미방문 정점 중 가장 가까운 것을 고른다. 이 한 줄이 알고리즘의 심장이다.

확정(visited)

한 번 방문 처리된 정점의 거리는 최종값이다. 이후 다시 건드리지 않는다.

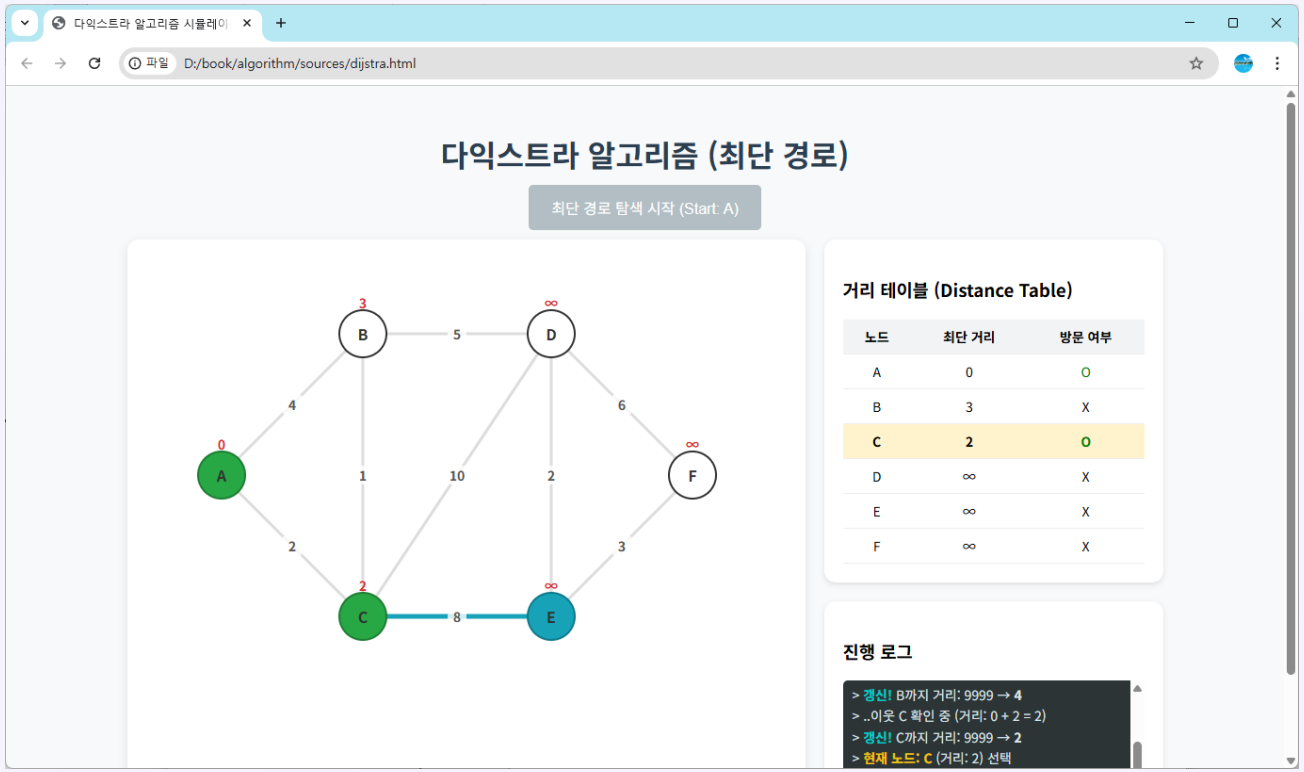
완화(relaxation)

u를 경유하는 새 경로가 더 짧으면 dist를 고친다. 갱신은 이웃에만 전파된다.

무한대 초기화

아직 도달 방법을 모르는 정점은 무한대로 두어 비교에서 자연히 밀려나게 한다.

동작 과정 한눈에 보기



① 시작점만 0

나머지는 모두 무한대로 두고 출발한다.

② 가장 가까운 곳 확정

큐에서 거리가 최소인 정점을 꺼내 방문 처리한다
.

③ 이웃 거리 갱신

그 정점을 경유하면 더 짧아지는 이웃의 값을 고친다.

④ 큐가 빌 때까지

②와 ③을 반복하면 모든 최단 거리가 확정된다.

파이썬 구현 — 우선순위 큐를 쓴다

dijkstra.py

```python
import heapq          # 우선순위 큐

INF = float("inf")

def dijkstra(graph, start):
    # 1. 초기화: 모든 노드까지의 거리를 무한대로
    distances = {node: INF for node in graph}
    distances[start] = 0

    # 2. (거리, 노드) 형태로 큐에 넣는다
    queue = []
    heapq.heappush(queue, (0, start))

    while queue:
        # 3. 가장 가까운 노드 선택 – 그리디 선택
        cur_dist, cur_node = heapq.heappop(queue)

        # 4. 이미 더 짧은 경로를 찾았다면 무시
        if distances[cur_node] < cur_dist:
            continue

        # 5. 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
```

힙이 최솟값을 대신 찾아 준다

매번 전체를 훑지 않아도 heappop 한 번이면 가장 가까운 노드가 나온다.

오래된 항목 건너뛰기

큐에는 갱신 이전 값이 남아 있을 수 있다. 4번 조건이 그것을 걸러 낸다.

갱신될 때만 push

거리가 실제로 줄어든 경우에만 큐에 넣어 불필요한 작업을 막는다.

실행 결과와 시간 복잡도

run.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")
```

OUTPUT

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

간선	모든 간선을 한 번씩 확인	E
힙	삽입·삭제마다 로그 시간	log V

O(E log V)

음수 가중치는 허용되지 않는다

지나갈 때마다 시간이 줄어드는 간선이 있다면 "지금 가장 가까운 곳이 최단"이라는 대전제가 무너진다. 그때는 벨만-포드를 쓴다.

A→B는 직행 8이 아니라 C를 경유한 6이 최단이다. 완화 단계가 정확히 작동했다.

5.9

코딩 테스트

배운 전략을 직접 코드로 옮겨 본다.

5.9

CT 5.1 동전 내주기

동전 내주기

편의점에서 일하는 준규는 손님에게 K원을 거슬러 주어야 한다. 동전은 N종류이며 각각 무한히 많다. 흥미로운 점은 동전의 가치가 오름차순으로 주어지되 "큰 단위는 반드시 작은 단위의 배수"라는 규칙을 따른다는 것이다. K원을 만드는 데 필요한 동전의 최소 개수를 구하라.

input.txt

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

OUTPUT

```
6
```

4200원

1000원 4개
100원 2개
합계 6개

배수 관계라는 조건이 곧 "그리디를 써도 안전하다"는 보증서다.

왜 배수 조건이 중요한가

§5.2의 400원 반례를 떠올려 보자. 큰 단위가 작은 단위의 배수가 아니면 큰 것부터 집는 전략이 무너진다. 이 문제는 그 위험을 조건으로 미리 제거해 두었다.

큰 단위부터 몫과 나머지로 처리하면 반복 횟수는 동전 종류 수만큼이다.

CT 5.1 동전 내주기

풀이 — 큰 단위부터 몫으로 한 번에

coin_change.py

```python
import sys

# N: 동전의 종류 수, K: 목표 금액
N, K = map(int, sys.stdin.readline().split())

coins = [int(sys.stdin.readline()) for _ in range(N)]

# 핵심 전략: 큰 단위부터 확인하도록 뒤집는다
coins.sort(reverse=True)

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)
```

reverse=True

입력이 오름차순이므로 뒤집어야 큰 단위부터 본다.

몫으로 일괄 처리

1개씩 빼면 K가 클 때 시간 초과가 난다. 나눗셈 한 번으로 끝낸다.

K == 0이면 break

남은 금액이 0이 되는 순간 더 볼 동전이 없다.

시간 복잡도 O(N)

동전 종류 수만큼만 반복한다. 금액 K와는 무관하다.

CT 5.2 ATM 줄 세우기

ATM 줄 세우기

어떤 은행에는 ATM이 딱 한 대뿐이다. 지금 그 앞에 N명이 줄을 서 있고, i번 사람이 돈을 인출하는 데 걸리는 시간은 P[i]분이다. N명을 적절한 순서로 줄 세워서 모든 사람이 돈을 인출하는 데 필요한 시간의 합을 최소로 만들어라.

input.txt

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

OUTPUT

```
32
```

내 인출 시간은 나뿐 아니라 내 뒤에 선 모든 사람의 대기 시간에 더해진다.

그리디 기준: 짧은 사람부터

앞에 선 사람의 시간은 뒷사람 전원에게 반복해서 더해진다. 그러니 여러 번 더해질 값일수록 작아야 한다. 오름차순 정렬이 곧 정답이다.

순서	시간	누적
1번째	1	1
2번째	2	3
3번째	3	6
4번째	3	9
5번째	4	13

1 + 3 + 6 + 9 + 13 = 32 — 이보다 작은 합을 만드는 순서는 존재하지 않는다.

CT 5.2 ATM 줄 세우기

풀이 — 정렬 후 누적합을 두 번 더한다

atm.py

```python
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)
```

정렬이 곧 증명

교환 논법으로 보면, 앞뒤를 바꿔 더 작아지는 경우가 없다는 것이 오름차순이다.

누적합의 누적합

current_sum은 그 사람이 끝나는 시각, total_sum은 그 시각들의 합이다.

O(n log n)

정렬이 전부이고, 이후 순회는 O(n)이다.

두 문제 모두 "무엇을 기준으로 정렬할 것인가"를 정하는 순간 풀이가 끝났다.

SUMMARY

5장 정리

그리디는 되돌아보지 않는다

매 순간 가장 좋아 보이는 것을 고르고 그 선택을 번복하지 않는다. 선택 → 적정성 검사 → 해답 검사의 세 단계로 돌아간다.

통하려면 두 조건이 필요하다

탐욕적 선택 속성과 최적 부분 구조. 둘 중 하나라도 깨지면 최적해를 보장하지 못한다.

정렬 기준이 전략이다

활동 선택은 종료 시간, 배낭은 가성비, 크루스칼은 간선 가중치. 무엇으로 줄을 세우느냐가 답을 결정한다.

성공하는 대표 사례

활동 선택, 쪼갤 수 있는 배낭, 크루스칼 MST, 다익스트라 최단 경로는 그리디로 정확한 최적해를 얻는다.

실패하는 대표 사례

400원 동전이 낀 거스름돈, 0/1 배낭, TSP의 가장 가까운 이웃은 그리디가 최적해를 놓친다.

다음은 동적 계획법

그리디가 포기한 0/1 배낭 문제는 6장에서 표를 채우는 방식으로 다시 푼다.

CHAPTER 05

Q & A

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