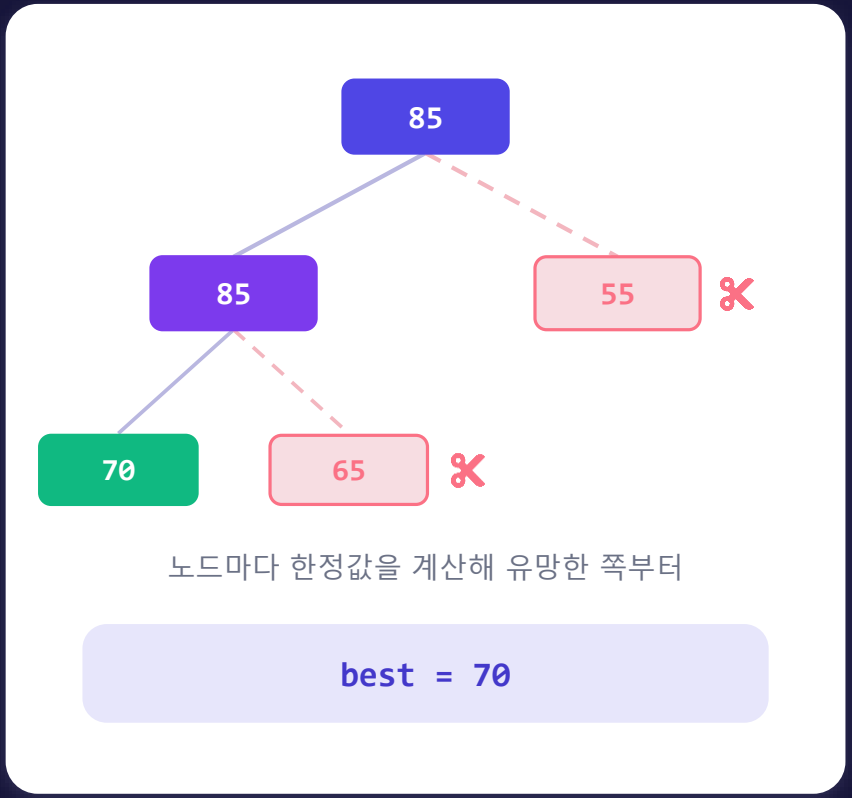
그림으로 쉽게 설명하는 파이썬 알고리즘

CHAPTER 08

분기한정 기법

Branch and Bound

가장 유망해 보이는 길부터 먼저 간다



CONTENTS

이번 장에서 배울 내용

> 7장의 백트래킹이 "갈 수 없는 길"을 잘랐다면, 이 장은 "갈 가치가 없는 길"을 자른다.

8.1

최적화를 향해

백트래킹을 넘어서 — "해가 있는가?"에서
"가장 좋은 해는 무엇인가?"로 질문이 바뀐다.

8.1

백트래킹 복습 — 강점과 한계

백트래킹의 강점

상태 공간 트리를 DFS로 탐색한다
제약 조건을 위반하면 즉시 되돌아간다
가지치기로 가망 없는 길을 조기에 차단한다
N-Queen, 미로 찾기 같은 퍼즐을 해결한다

"해가 있는가?" (Yes / No)에 특화

백트래킹의 한계

탐색 순서가 맹목적인 DFS로 고정된다
형편없는 답을 먼저 탐색할 수 있다
최적화 문제에서 심각한 비효율이 생긴다
"더 좋은 해가 있는가"를 판단할 수단이 없다

최적해를 "빨리" 찾을 방법이 없다

문제가 바뀌면 무기도 바뀌어야 한다

백트래킹의 가지치기는 "이 길을 갈 수 있는가"만 묻는다. 최적화 문제에서는 갈 수는 있지만 갈 이유가 없는 길이 훨씬 많다.

결정 문제 vs 최적화 문제

결정 문제 (decision problem)

질문: "조건을 만족하는 해가 존재하는가?"
대답: Yes 또는 No

대표 예제

N-Queen 문제 · 미로 찾기
그래프 색칠하기 · 부분 집합의 합

7장 백트래킹의 영역

최적화 문제 (optimization problem)

질문: "f(X)를 최대 또는 최소로 만드는 해 X는?"
대답: 가장 좋은 값과 그때의 해

대표 예제

배낭 채우기 (가치 최대화)
외판원 순회 (비용 최소화) · 최소 신장 트리

8장 분기한정의 영역

최적화 문제에서는 "찾았다"로 끝나지 않는다 — 더 좋은 것이 없음을 확인해야 비로소 끝난다.

분기한정의 핵심 아이디어

무작정 깊이 들어가지 말고, 가장 유망해 보이는 길부터 먼저 가보자 — 최우선 탐색(best-first search).

백트래킹 탐험가

문패만 보고 들어갈지 말지를 정한다.
곧 단순한 제약 조건 검사다.

"이 방은 못 들어가는 방인가?"

분기한정 탐험가

계산기를 두드려 보고, 가장 이득이 될 것 같은 방부터 먼저 들어간다.
곧 수학적 예측, 한정값 계산이다.

"이 방에 가봤자 최대 얼마인가?"

분기한정 = 백트래킹의 가지치기 + 수학적 예측

단순한 조건 검사를 넘어, 미래의 가능성을 숫자로 계산하면서 최적해를 찾아 나선다. 이 숫자가 바로 한정값(bound)이다.

한정값(bound)의 개념

"이쪽 길로 계속 갔을 때 내가 얻을 수 있는 보물의 최대 가치(상한선)는 얼마쯤 될까?"

1

한정값을 계산한다

이 노드 아래에서 얻을 수 있는 최선의 값을 낙관적으로 추정한다.

2

기존 최적해와 비교한다

지금까지 찾은 가장 좋은 해(best-so-far)와 그 추정치를 견준다.

3

가망 없으면 잘라 낸다

상한선이 이미 찾은 답보다 낮다면 굳이 가볼 이유가 없다.

"굳이 가볼 필요가 없겠군" — 과감하게 가지를 쳐낸다

백트래킹은 규칙 위반만 걸러 냈지만, 분기한정은 규칙을 지키더라도 이득이 없는 길을 계산으로 걸러 낸다. 잘라 내는 근거가 조건문에서 수식으로 바뀐 것이다.
다만 한정값은 반드시 낙관적이어야 한다 — 실제보다 나쁘게 추정하면 최적해까지 잘라 버린다.

8.2

분기한정의 3박자

8.2

갱도를 파고(분기), 매장량을 예측하고(한정),
기대에 못 미치면 폐쇄한다(가지치기).

분기한정의 세 요소

분기 (Branch)

갱도를 여러 갈래로 파고 들어간다.

상태 공간 트리를 확장해 나가는 과정이다.

한정 (Bound)

각 갱도 입구에서 매장량을 과학적으로 예측한다.

한정값을 계산하는 과정이다.

가지치기 (Prune)

예측량이 기대에 못 미치는 갱도는 가차 없이 폐쇄한다.

탐색을 중단하는 과정이다.

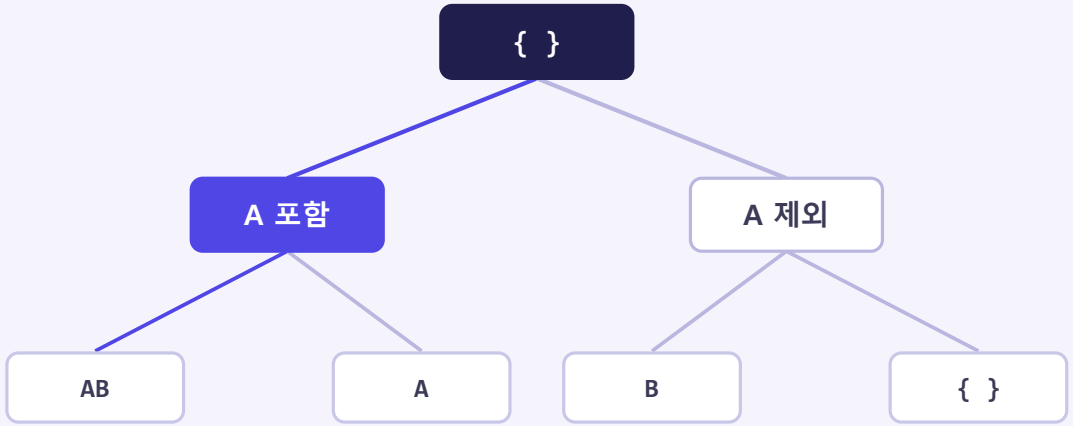
세 요소는 순환한다

분기해서 자식을 만들고, 각 자식의 한정값을 계산하고, 기준에 못 미치면 잘라 낸다. 살아남은 노드에 대해 다시 분기한다 — 이 순환이 큐가 빌 때까지 반복된다.

백트래킹에 없던 것은 가운데 "한정" 하나뿐이다. 그 하나가 탐색의 성격을 바꾼다.

분기 (branching) — 상태 공간 트리 나누기

하나의 큰 문제를 서로 배타적인 여러 개의 작은 부분 문제로 나누는 것이다. 0/1 배낭 문제(물건 A, B, C)를 예로 들어 보자.



루트 = 아무것도 결정하지 않은 초기 상태 · 레벨마다 물건 하나씩 결정

루트 노드

아무것도 결정하지 않은 초기 상태에서 출발한다.

첫 번째 분기

"물건 A를 넣는다" vs "물건 A를 넣지 않는다".

두 번째 분기

각 자식에서 물건 B에 대해 똑같이 두 갈래로 나눈다.

무턱대고 분기만 하면 노드 수가 지수적으로 폭발한다

물건이 n개면 잎 노드는 2^n개다. 분기 자체는 문제를 나눌 뿐 아무것도 줄여 주지 않는다. 줄이는 일은 다음 두 단계, 한정과 가지치기의 몫이다.

한정 (bounding) — 가능성의 상한선과 하한선

최대화 문제 → 상한 (upper bound)

이익, 가치, 점수 등을 최대화하는 문제다.

"어떤 선택을 하더라도 얻을 수 있는 가치의 최댓값은?"

전략 — 제약 조건 완화 (relaxation)

0/1 배낭이라면 물건을 쪼갤 수 있다고 가정하고 그리디하게 계산한다. 실제보다 후하게 잡히므로 안전한 상한이 된다.

최소화 문제 → 하한 (lower bound)

비용, 거리, 시간 등을 최소화하는 문제다.

"반드시 발생할 수밖에 없는 최소 비용은?"

전략 — 가장 낙관적인 가정

TSP라면 각 도시에서 가장 짧은 간선만 골라 더한다. 실제보다 싸게 잡히므로 안전한 하한이 된다.

두 경우 모두 "현실보다 유리하게" 계산하는 것이 원칙이다

한정값은 실제 결과보다 결코 나쁘면 안 된다. 낙관적으로 잡아야 최적해를 실수로 잘라 내지 않는다. 대신 지나치게 헐거우면 가지치기가 잘 되지 않는다.

가지치기 (pruning) — 계산에 의한 포기

"가망 없는 길은 쳐다보지도 마라."

최대화 문제를 기준으로 가지치기 논리를 정리하면 다음과 같다.

① 갱신 해를 발견할 때마다 지금까지의 최적해(best-so-far)를 갱신한다. `best 갱신`

② 계산 어떤 노드에 도착하면 그 노드의 상한값을 계산한다. `UB 계산`

상한값 > best-so-far

아직 더 좋은 해가 나올 수 있다 → 유망함, 탐색을 계속한다

상한값 ≤ best-so-far

최고로 잘 나와야 이미 아는 답 이하다 → 즉시 가지치기

최소화 문제라면 부등호만 뒤집으면 된다 — 하한값 ≥ best-so-far 이면 잘라 낸다.

백트래킹 vs 분기한정 — 가지치기의 차이

구분	백트래킹	분기한정
판단 기준	타당성 (feasibility)	**최적성 (optimality)**
질문	"제약 조건을 위반했는가?"	"더 좋은 해를 찾을 가능성이 있는가?"
차단 대상	갈 수 없는 길	**갈 가치가 없는 길**
탐색 전략	`DFS` 고정	`Best-First / DFS / BFS` 선택
문제 유형	결정 문제 (Yes/No)	최적화 문제 (Max/Min)

"갈 수 없는 길"과 "갈 가치가 없는 길"

백트래킹은 규칙을 어긴 길만 막는다. 분기한정은 규칙을 지키더라도 이득이 없는 길까지 막는다. 최적화 문제에서는 후자가 압도적으로 많기 때문에 효과가 크다.

분기한정은 백트래킹을 대체하는 것이 아니라 포함한다 — 타당성 검사는 여전히 함께 쓰인다.

8.3

탐색의 전략

같은 트리라도 어떤 순서로 꺼내느냐에 따라
방문하는 노드 수가 완전히 달라진다.

8.3

세 가지 탐색 전략 비교

전략	자료구조	탐색 순서	장점	단점
FIFO (BFS)	큐 (queue)	얕은 곳부터 레벨 순서	레벨별 체계적 탐색	메모리 폭발 — 거의 미사용
LIFO (DFS)	스택 (stack)	한 방향 끝까지 깊이 우선	메모리 효율적	맹목적 탐색으로 시간 낭비
Best-First	**우선순위 큐**	한정값이 가장 좋은 것부터	최적해 조기 발견, 가지치기 극대화	큐 크기 관리 필요

결론 — 특별한 제약이 없다면 Best-First Search가 표준이다

세 전략은 모두 같은 상태 공간 트리를 탐색한다. 차이는 오직 "다음에 어느 노드를 꺼낼 것인가"뿐인데, 이 선택이 방문 노드 수를 자릿수 단위로 바꾼다.

다만 메모리가 빠듯하면 LIFO(DFS) 기반 분기한정이 현실적인 절충안이 된다.

(1) FIFO 분기한정 — BFS 기반

큐(queue)를 써서 트리를 레벨 순서대로, 얕은 곳부터 넓게 탐색한다.

① 루트 노드부터 시작한다.

② 현재 레벨의 모든 노드를 방문하고 각각의 한정값을 계산한다.

③ 살아남은 자식 노드들을 큐에 순서대로 삽입한다.

④ 큐의 맨 앞, 즉 가장 먼저 들어온 노드를 꺼내서 탐색한다.

치명적 단점 — 메모리 오버플로우

특정 레벨에 존재하는 노드 수가 기하급수적으로 폭발한다. 그래서 현실적인 문제에서는 거의 쓰이지 않는다.

(2) LIFO 분기한정 — DFS 기반

스택(또는 재귀)을 써서 한쪽 경로를 끝까지 파고든다. 탐색 순서 자체는 7장의 백트래킹과 같다.

① 한쪽 경로를 선택해서 끝까지 내려간다.

② 잎 노드에 도달하거나 가지치기되면 직전 부모로 되돌아온다.

③ 아직 가보지 않은 다른 자식 노드로 다시 깊이 내려간다.

백트래킹과의 결정적 차이

탐색 순서는 같지만, 노드에 도착할 때마다 한정값을 계산해 미래의 가능성을 예측한다는 점이 다르다. 백트래킹은 단순히 제약 조건만 검사한다.

장점 — FIFO에 비해 메모리 사용량이 압도적으로 적다

단점 — 최적해가 다른 쪽에 있으면 시간을 낭비한다

(3) 최우선 탐색 (best-first search)

분기한정 기법의 꽃이자 가장 강력한 전략 — 핵심 도구는 우선순위 큐(priority queue)다.

맹목적인 순서(FIFO, LIFO) 대신 가장 유망한 노드부터 먼저 탐색한다.

① 새로운 자식 노드를 생성하고 각각의 한정값을 계산한다.

② 우선순위 큐에 삽입한다. 한정값 순서로 자동 정렬된다.

③ 우선순위가 가장 높은, 즉 한정값이 최선인 노드를 꺼내 탐색한다.

최대화 → 상한값이 가장 큰 노드부터

최소화 → 하한값이 가장 작은 노드부터

최우선 탐색의 장점

1

최적해를 빨리 발견한다

가능성이 가장 높은 경로를 먼저 탐색하므로, 좋은 해에 일찍 도달할 확률이 높다.

2

기준이 빨리 높아진다

좋은 해를 빨리 찾으면 best-so-far가 그만큼 일찍 올라간다.

3

가지치기가 극대화된다

기준이 높아질수록 그보다 못한 노드들이 더 쉽고 빠르게 제거된다.

세 가지 장점은 서로를 강화하는 선순환을 만든다

유망한 순서로 탐색 → 좋은 해를 일찍 확보 → 기준이 올라감 → 더 많은 가지가 잘림 → 남은 탐색이 더 빨라짐. 이것이 분기한정에서 우선순위 큐를 표준으로 삼는 이유다.

대가는 메모리다

우선순위 큐에는 아직 확장하지 않은 노드가 모두 살아 있다. 문제가 커지면 큐 자체가 부담이 되므로, 실무에서는 큐 크기를 제한하거나 DFS와 섞어 쓰는 하이브리드 전략을 쓴다.

8.4

작업 할당 문제

세 사람에게 세 가지 일을 나눠 주되,
비용의 합이 최소가 되도록 배정하라.

8.4

§8.4 작업 할당 문제

작업 할당 문제 — 문제 정의

직원 A, B, C가 있고 작업 1, 2, 3이 있다. 각 사람은 정확히 하나의 작업을 맡는다. 비용의 합이 최소가 되도록 배정하라.

직원 ₩ 작업	작업 1	작업 2	작업 3
A	2	9	5
B	6	8	3
C	7	1	4

최적해 : A→작업1, B→작업3, C→작업2

총 비용 = 2 + 3 + 1 = 6

한정값 공식

"아직 일을 안 맡은 사람들은 각자의 최소 비용을 낸다고 치자."

왜 이것이 하한인가

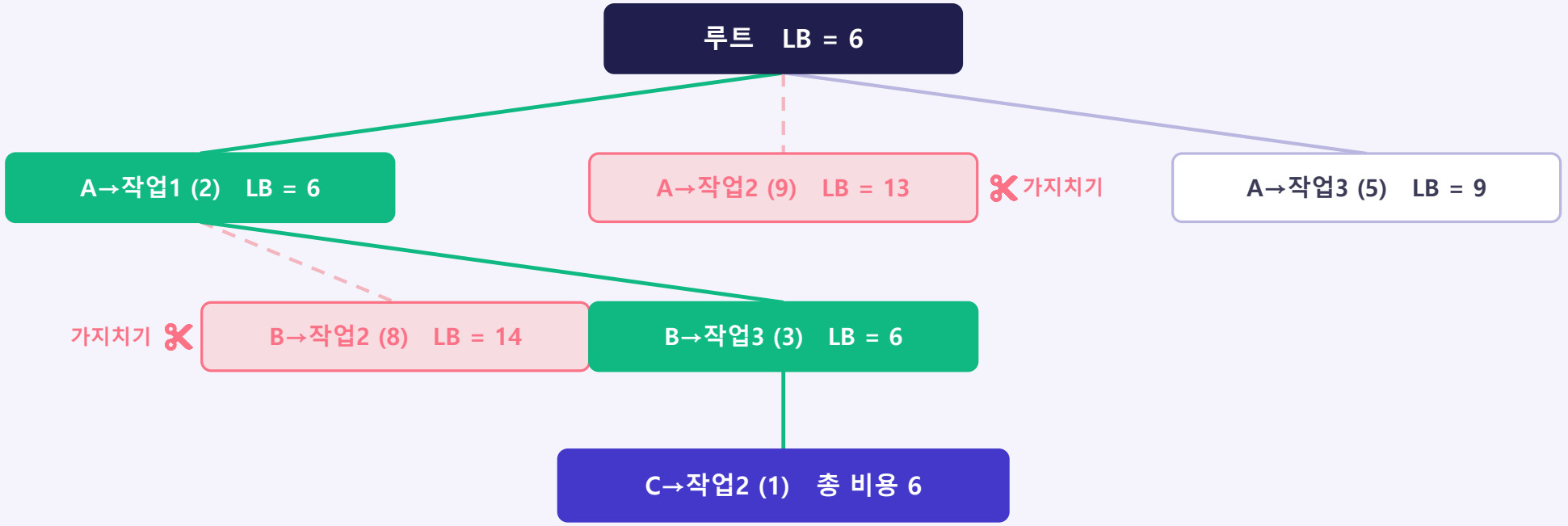
실제로는 작업이 겹치면 안 되므로 각자 최소 비용을 동시에 낼 수 없다. 따라서 이 값은 실제 비용 이하 — 안전한 하한이다.

완전 탐색이라면 3! = 6가지, N명이면 N!가지

N이 10만 되어도 360만 가지가 넘는다. 분기한정은 한정값으로 가망 없는 배정을 미리 접어 버린다.

작업 할당 — 분기한정 풀이 과정

루트의 하한값은 각 사람의 최소 비용을 모두 더한 값이다. LB = min(A) + min(B) + min(C) = 2 + 3 + 1 = 6



LB가 가장 작은 노드부터 파고든다 · 굵은 초록 경로가 실제 탐색 순서

작업 할당 문제 — 파이썬 코드 (1/2)

job_assignment.py

```python
import heapq

costs = [
    [2, 9, 5],   # A
    [6, 8, 3],   # B
    [7, 1, 4],   # C
]
N = 3

class Node:
    def __init__(self, person, assigned_jobs, current_cost):
        self.person = person
        self.assigned_jobs = assigned_jobs
        self.current_cost = current_cost
        self.lb = self.calculate_lb()

    def calculate_lb(self):
        lb = self.current_cost
        for i in range(self.person + 1, N):   # 아직 배정 안 된 사람
            min_val = float("inf")
            for j in range(N):
                if not self.assigned_jobs[j]:  # 남은 작업 중
                    min_val = min(min_val, costs[i][j])
            lb += min_val
        return lb

    def __lt__(self, other):
        return self.lb < other.lb    # 우선순위 큐 정렬 기준
```

person

지금까지 몇 번째 사람까지 배정했는지를 나타낸다. 곧 트리의 깊이다.

assigned_jobs

어떤 작업이 이미 배정되었는지 표시하는 불리언 리스트다.

calculate_lb

남은 사람마다 남은 작업 중 최소 비용을 더한다 — 이것이 하한값이다.

__lt__

heapq가 lb 기준으로 정렬하도록 비교 연산자를 정의한다.

작업 할당 문제 — 파이썬 코드 (2/2)

job_assignment.py

```python
def solve_job_assignment():
    pq = []
    root = Node(-1, [False] * N, 0)
    heapq.heappush(pq, root)

    while pq:
        node = heapq.heappop(pq)     # LB가 가장 작은 노드 먼저

        if node.person == N - 1:     # 모든 사람 배정 완료
            return node.current_cost

        next_person = node.person + 1

        for job in range(N):
            if not node.assigned_jobs[job]:
                new_assigned = node.assigned_jobs[:]
                new_assigned[job] = True
                new_cost = node.current_cost + costs[next_person][job]
                child = Node(next_person, new_assigned, new_cost)
                heapq.heappush(pq, child)

print(solve_job_assignment())
```

OUTPUT

```
6
```

heappop이 전략 그 자체

LB가 가장 작은 노드를 꺼내는 이 한 줄이 곧 최우선 탐색이다.

꺼낸 즉시 종료해도 된다

완성된 노드를 꺼냈다면 그보다 LB가 작은 노드가 큐에 없다는 뜻이므로 최적해가 확정된다.

가지치기는 어디에?

LB가 큰 노드는 큐 뒤로 밀려 영영 꺼내지지 않는다 — 사실상 잘린 것과 같다.

최소 비용 6 — 전수조사 없이 우선순위 큐가 최적 배정을 먼저 꺼내 준다.

8.5

0/1 배낭 문제

쪼갤 수 없는 보석을 담는 최대화 문제.
상한값을 계산해 가망 없는 가지를 자른다.

8.5

§8.5 0/1 배낭 문제

0/1 배낭 문제 — 문제 재정의

배낭의 무게 제한을 넘지 않으면서 가치의 합이 최대가 되도록 담는다. 보석은 쪼갤 수 없으므로 통째로 가져가거나(1) 두고 가야(0) 한다.

보석	무게	가치	가성비 (v/w)
보석 1	2kg	40원	**20**
보석 2	5kg	30원	6
보석 3	10kg	50원	5
보석 4	5kg	10원	2

배낭 용량 W = 10kg

전형적인 최대화 문제

가치를 최대로 만들어야 하므로, 상한값(UB)을 계산해 가지치기한다.

6장 DP와 무엇이 다른가

DP는 용량 W만큼의 표를 채운다. 분기한정은 표를 만들지 않고 유망한 조합만 골라 탐색한다. W가 아주 크거나 실수일 때 유리하다.

가성비 내림차순 정렬이 모든 것의 출발점

보석 1(20) → 보석 2(6) → 보석 3(5) → 보석 4(2) 순서로 정렬해 두면, 상한값을 그리디하게 한 번의 순회로 계산할 수 있다.

§8.5 0/1 배낭 문제

한정값 계산 전략 — "만약 쪼갤 수 있다면?"

핵심은 제약 조건 완화(relaxation) — 0/1 제약을 풀어 주고 분할 가능 배낭 문제로 계산한다.

① 모든 보석을 가성비(v/w)가 높은 순서대로 정렬한다.

② UB를 현재까지 확보한 가치(current_value)로 초기화한다.

③ 남은 보석을 가성비 순서로 보며, 통째로 담을 수 있으면 담고 UB를 늘린다.

④ 못 담으면 남은 용량만큼 비율로 쪼개서 담는다고 가정하고 그만큼만 더한다.

실제로는 쪼갤 수 없으므로 언제나 실제 최적해 ≤ UB — 완벽하게 안전한 상한선이다.

§8.5 0/1 배낭 문제

시뮬레이션 ① 루트 노드 (W = 10kg)

루트는 아무것도 담지 않은 상태다. 현재 무게 0, 현재 가치 0에서 상한값을 계산해 보자.

보석 1	2kg, 40원 — 통째로 담는다. 남은 용량 8kg.	+40
보석 2	5kg, 30원 — 통째로 담는다. 남은 용량 3kg.	+30
보석 3	10kg, 50원 — 3kg만 쪼개서 담는다고 가정: (3/10) × 50 = 15원.	+15

UB = 40 + 30 + 15 = 85

쪼갤 수 있다고 가정했을 때의 최대 가치 — 실제 최적해는 이보다 클 수 없다

```
PQ = [ (UB=85, level=0, w=0, v=0) ]
```

시뮬레이션 ② 첫 번째 분기 (보석 1)

왼쪽 — 보석 1 포함

무게 2kg, 가치 40원. 남은 8kg에 보석 2를 통째로, 보석 3을 3kg만 넣는다고 가정하면 UB는 그대로 85다.

`UB = 85` · `max_value = 40` 으로 갱신

유망하다 → 우선순위 큐에 삽입

오른쪽 — 보석 1 미포함

무게 0, 가치 0. 남은 10kg에 보석 2(30원)를 통째로, 보석 3을 절반(25원) 넣는다고 가정하면 UB는 55로 떨어진다.

`UB = 55`

아직은 유망 → 큐에 함께 삽입

PQ = [(UB=85, 보석1 포함), (UB=55, 보석1 미포함)]

UB가 85인 노드가 먼저 꺼내진다

우선순위 큐가 상한값이 큰 쪽을 앞에 세운다. 가장 유망한 가지를 먼저 파고들면 좋은 해를 일찍 만나게 되고, 그만큼 기준이 빨리 올라간다.

§8.5 0/1 배낭 문제

시뮬레이션 ③ 가지치기의 위력

보석 1과 2를 넣고 나머지는 넣지 않는 단말 노드에 도달하면 가치는 70이 된다. max_value가 70으로 확정된다.

max_value = 70

보석 1(40) + 보석 2(30) — 지금까지 찾은 최선

이제 PQ에서 UB = 55 노드(보석 1 미포함)가 나올 차례다

"이 길로 가봤자 최고로 잘 나와야 55원인데, 나는 이미 70원짜리 답을 알고 있다. 가볼 필요가 없다."

`UB(55) ≤ max_value(70) → 즉시 가지치기, 자식 노드를 아예 생성하지 않는다`

결과 — 최적해는 가치 70 (보석 1 + 보석 2)

그 아래 매달려 있던 모든 조합을 단 한 번의 부등호 비교로 통째로 지웠다. 이것이 한정값이 하는 일이다.

좋은 해를 일찍 찾을수록 기준이 높아지고, 기준이 높을수록 더 많이 잘린다.

0/1 배낭 문제 — 파이썬 코드 (1/2)

knapsack_bnb.py

```python
import heapq

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

class Node:
    def __init__(self, level, profit, weight, bound):
        self.level = level    # 고려 중인 아이템 인덱스
        self.profit = profit  # 현재까지의 가치 합
        self.weight = weight  # 현재까지의 무게 합
        self.bound = bound    # 잠재적 최대 가치 (상한값)

    # heapq는 최소 힙 -> 부등호를 뒤집어 최대 힙처럼
    def __lt__(self, other):
        return self.bound > other.bound
```

bound.py

```python
def calculate_bound(node, n, capacity, items):
    if node.weight >= capacity:
        return 0

    profit_bound = node.profit
    j = node.level + 1
    tot_weight = node.weight

    # 1. 통째로 넣을 수 있는 만큼 넣는다
    while j < n and tot_weight + items[j].weight <= capacity:
        tot_weight += items[j].weight
        profit_bound += items[j].value
        j += 1

    # 2. 남은 공간은 쪼개서 채운다고 가정 (fractional)
    if j < n:
        profit_bound += (capacity - tot_weight) * items[j].ratio

    return profit_bound
```

ratio를 미리 계산한다

생성자에서 가성비를 구해 두면 정렬과 상한값 계산이 모두 단순해진다.

__lt__ 부등호를 뒤집는 이유

파이썬 heapq는 최소 힙이다. bound가 큰 쪽을 먼저 꺼내려면 비교를 반대로 정의해야 한다.

쪼개서 더하는 마지막 한 줄

남은 용량 × 가성비를 더하는 이 줄이 0/1 제약을 완화해 상한을 만든다.

0/1 배낭 문제 — 파이썬 코드 (2/2)

knapsack_bnb.py

```python
def solve_knapsack_bnb(capacity, weights, values):
    n = len(values)
    items = [Item(weights[i], values[i]) for i in range(n)]
    items.sort(key=lambda x: x.ratio, reverse=True)  # [1] 가성비 정렬
    pq = []
    u = Node(level=-1, profit=0, weight=0, bound=0)
    u.bound = calculate_bound(u, n, capacity, items)
    heapq.heappush(pq, u)
    max_profit = 0
    while pq:
        u = heapq.heappop(pq)              # [2] Best-First
        if u.bound <= max_profit:          # [3] 가지치기
            continue
        level = u.level + 1

        # 분기 1: 아이템을 넣는 경우 (왼쪽 자식)
        nw = u.weight + items[level].weight
        np_ = u.profit + items[level].value
        if nw <= capacity:
            if np_ > max_profit: max_profit = np_
            v_inc = Node(level, np_, nw, 0)
            v_inc.bound = calculate_bound(v_inc, n, capacity, items)
            if v_inc.bound > max_profit: heapq.heappush(pq, v_inc)

        # 분기 2: 아이템을 넣지 않는 경우 (오른쪽 자식)
        v_exc = Node(level, u.profit, u.weight, 0)
        v_exc.bound = calculate_bound(v_exc, n, capacity, items)
        if v_exc.bound > max_profit: heapq.heappush(pq, v_exc)

    return max_profit
```

run.py

```python
W = 10
val = [40, 30, 50, 10]
wt  = [2, 5, 10, 5]
print(solve_knapsack_bnb(W, wt, val))
```

OUTPUT

```
70
```

[1] 가성비 정렬

이 전처리가 없으면 상한값이 헐거워져 가지치기

[2] Best-First

bound가 가장 큰 노드를 먼저 꺼낸다.

[3] 가지치기

bound가 이미 찾은 최대값 이하면 확장하지 않고 건너뛴다.

8.6

외판원 순회 문제 (TSP)

8.6

모든 도시를 한 번씩 들르고 돌아오는
최소 비용 경로를 하한값으로 찾아낸다.

§8.6 외판원 순회(TSP)

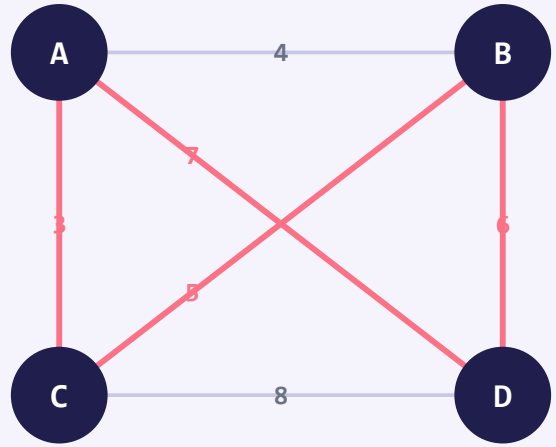
외판원 순회 문제 — 복습

모든 도시를 정확히 한 번씩만 방문하고 출발 도시로 되돌아오는 경로 중 총 이동 비용이 가장 적은 것을 찾는다. 곧 최소 비용의 해밀턴 순환(Hamiltonian cycle)을 찾는 문제다.

비용을 최소화한다 → 하한값(LB)
최소화 문제이므로 하한값을 계산해 가지치기한다.

완전 탐색은 (n-1)!
도시가 20개만 넘어도 사실상 계산이 불가능하다.

분기한정은 전수조사보다 훨씬 똑똑하게 최적해를 찾는다



최적 경로 A → C → B → D → A (비용 21)

TSP 하한값 — "최단 2간선 합계" 방식

모든 도시에는 들어오는 문과 나가는 문이 하나씩 있다 → 각 도시에서 가장 짧은 간선 2개를 더한다.

$$LB = \lceil S / 2 \rceil = \lceil \Sigma\,(d_{i,1} + d_{i,2}) / 2 \rceil$$

① 각 도시 i에 대해 가장 짧은 간선 2개의 길이를 더한다.

② 모든 도시에 대해 그 값을 합쳐 S를 구한다.

③ 간선 하나는 두 도시를 잇기 때문에 두 번씩 더해졌다 → S를 2로 나눈다.

④ 비용이 정수라면 소수점은 올림 처리한다.

TSP 예제 — 도시 4개

도시	A	B	C	D
A	∞	4	3	7
B	4	∞	5	6
C	3	5	∞	8
D	7	6	8	∞

루트 하한값 계산

A 3 + 4 = 7

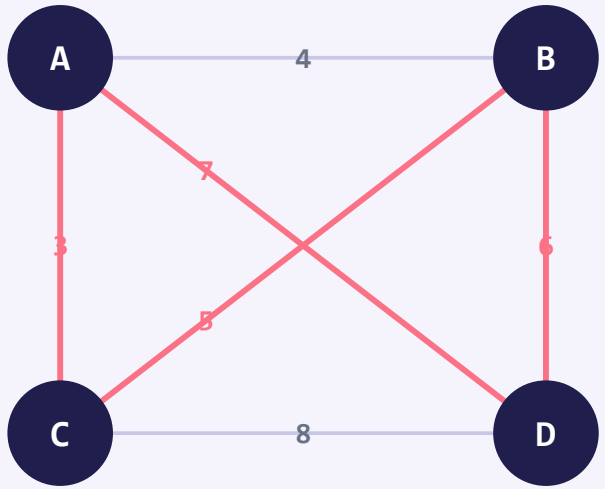
B 4 + 5 = 9

C 3 + 5 = 8

D 6 + 7 = 13

S = 37 → LB = ⌈37 / 2⌉ = 19

실제 최적 경로 비용은 21 — LB 19는 그보다 작으므로 안전한 하한이다



최적 경로 A → C → B → D → A · 비용 21

TSP 분기 과정 — A에서 출발

어떤 간선을 확정하면 그 도시의 "최단 2간선"이 바뀔 수 있다. 확정된 간선은 반드시 포함해야 하므로 하한값이 올라가기도 한다.

자식 #1 — 간선 (A,C) 선택, 거리 3

A: 필수 (A,C)[3] + 최단 (A,B)[4] = 7
C: 필수 (C,A)[3] + 최단 (C,B)[5] = 8
합계 7 + 9 + 8 + 13 = 37

LB = ⌈37/2⌉ = 19

유망!

자식 #2 — 간선 (A,B) 선택, 거리 4

A: (A,B)[4] + (A,C)[3] = 7
B: (B,A)[4] + (B,C)[5] = 9
합계 7 + 9 + 8 + 13 = 37

LB = 19

유망!

자식 #3 — 간선 (A,D) 선택, 거리 7

A: (A,D)[7] + (A,C)[3] = 10 ← 증가!
D: (D,A)[7] + (D,B)[6] = 13
합계 10 + 9 + 8 + 13 = 40

LB = ⌈40/2⌉ = 20

우선순위 밀림

LB가 19인 두 자식이 먼저, LB가 20인 자식은 뒤로 밀린다.

TSP — 파이썬 코드 (1/2)

tsp_bnb.py

```python
import heapq
INF = float("inf")

adj_matrix = [
    [INF, 4,   3,   7  ],   # A
    [4,   INF, 5,   6  ],   # B
    [3,   5,   INF, 8  ],   # C
    [7,   6,   8,   INF],   # D
]
N = len(adj_matrix)

def solve_tsp_simple(matrix):
    # [1] 각 도시별 가장 짧은 간선 비용을 미리 계산
    min_edges = []
    for row in matrix:
        min_val = min([x for x in row if x != INF])
        min_edges.append(min_val)

    pq = []
    initial_bound = sum(min_edges)
    heapq.heappush(pq, (initial_bound, 0, 0, [0]))
    min_cost = INF
    best_path = []
```

min_edges 전처리

도시마다 가장 싼 간선 비용을 미리 구해 두면 하한값을 매번 빠르게 만들 수 있다.

큐에 넣는 튜플 네 개

(하한값, 현재까지 비용, 현재 도시, 지금까지의 경로) 순서다.

튜플 첫 원소가 우선순위

heapq는 튜플을 앞에서부터 비교하므로 하한값이 작은 노드가 먼저 나온다.

출발점은 0번 도시

순환 경로이므로 어디서 출발해도 같다. 하나로 고정해 중복 탐색을 없앤다.

TSP — 파이썬 코드 (2/2)

tsp_bnb.py

```
    while pq:
        bound, current_cost, curr_node, path = heapq.heappop(pq)

        if bound >= min_cost:            # [가지치기]
            continue

        if len(path) == N:               # [경로 완성]
            return_cost = matrix[curr_node][path[0]]
            total_cost = current_cost + return_cost
            if total_cost < min_cost:
                min_cost = total_cost
                best_path = path + [path[0]]
            continue

        for next_node in range(N):       # [다음 도시 탐색]
            if next_node not in path:
                new_cost = current_cost + matrix[curr_node][next_node]
                future_bound = new_cost
                for i in range(N):
                    if i not in path and i != next_node:
                        future_bound += min_edges[i]
                if future_bound < min_cost:
                    heapq.heappush(pq,
                        (future_bound, new_cost, next_node, path + [next_node]))

    return best_path, min_cost
```

run.py

```
path, cost = solve_tsp_simple(adj_matrix)
names = {0:"A", 1:"B", 2:"C", 3:"D"}
print(" -> ".join(names[n] for n in path))
print(cost)
```

OUTPUT

```
A -> C -> B -> D -> A
21
```

하한 19, 실제 최적 21

하한값이 실제 값보다 작으므로 최적해를 잘라낼 위험이 없다. 그러면서도 20 이상인 가지는 일찍 밀어낸다.

시간 복잡도 비교 — DP vs 분기한정

구분	동적 계획법 (DP)	분기한정법 (B&B)
시간 복잡도	$O(nW)$	최악 $O(2^n)$ — 가지치기로 대폭 감소
특징	W가 커지면 메모리·시간이 비례해 증가	W와 무관. 물건 개수 n이 중요
결론	**W가 작을 때 유리**	**W가 매우 크거나 실수일 때 유리**

잘 설계된 한정값을 쓰면 실제로는 $O(2^n)$보다 훨씬 빠르게 동작한다

최악의 복잡도는 완전 탐색과 같지만, 그 최악은 한정값이 전혀 쓸모없을 때의 이야기다. 좋은 한정값 하나가 알고리즘의 실효 성능을 결정한다.

두 기법은 경쟁 관계가 아니다

용량이 작은 정수라면 6장의 DP가, 용량이 크거나 실수이거나 물건 수가 적당하다면 분기한정이 낫다. 문제의 모양을 보고 고르면 된다.

8.7

코딩 테스트

"분기한정"이라는 이름으로는 잘 나오지 않는다.
대신 "시간 초과"라는 형태로 나타난다.

8.7

코딩 테스트에서의 분기한정

용어가 명시적으로 등장하는 경우는 드물다

문제 지문에 "분기한정법으로 푸시오"라고 쓰여 있는 경우는 거의 없다. 그러나 난이도 상의 백트래킹 문제나 삼성 역량테스트(A형·Pro)에서는 이 기법을 모르면 시간 초과를 받게 되는 경우가 꽤 있다.

> "현재 상태에서 아무리 잘해봤자 기존 정답보다 못하다면 가차 없이 자른다"

논리적 가지치기 형태

별도의 우선순위 큐 없이, DFS 안에 부등호 한 줄을 넣는 형태로 자주 나온다.

정렬이 힌트다

큰 것부터 또는 가성비 순으로 정렬하라는 조건이 붙으면 가지치기를 쓰라는 신호다.

시간 초과가 신호다

정답은 맞는데 시간 초과라면, 가지치기 조건을 하나 더 찾아야 한다는 뜻이다.

CT 8.1 택배 배송 최적 경로

택배 배송 최적 경로 (TSP 변형)

물류 센터(0번)에서 출발해 N개 배송지를 모두 방문하고 다시 물류 센터로 돌아온다. 이동 비용 행렬 dist[i][j]가 주어질 때 최소 이동 비용을 구하라.

입력 규모	N이 10~15 정도로 작게 주어진다. 이 크기가 곧 분기한정을 쓰라는 신호다.	`N ≤ 15`
단순 DFS	모든 순열을 만들면 O(N!)이다. N이 12만 넘어가도 위험하다.	`O(N!)`
한정값	아직 방문하지 않은 도시들의 "가장 싼 간선 비용"을 모두 더해 현재 비용에 얹는다.	`LB`
자료구조	우선순위 큐로 유망한 경로부터 꺼내고, LB가 현재 최적해 이상이면 버린다.	`PQ`

CT 8.2 로드 밸런싱

클라우드 서버 로드 밸런싱

N개의 작업을 K대 서버에 분배한다. 작업은 분할할 수 없다. 최대 부하의 최솟값 min(max(load₀, ..., load_{K-1}))을 구하라.

1

이미 나쁜 서버

현재 서버 중 하나라도 이미 ans보다 부하가 크면 그 아래는 볼 필요가 없다 → 즉시 중단.

2

추가하면 넘는 경우

이 서버에 작업을 얹었을 때 ans 이상이 된다면 그 배정 자체를 시도하지 않는다.

3

대칭성 제거

빈 서버에 작업을 배정했다면, 다른 빈 서버에 배정하는 경우는 본질적으로 같으므로 건너뛴다.

효율 전략 — 큰 작업부터 배정한다

큰 작업을 먼저 놓으면 부하가 일찍 커져서 가지치기 조건에 더 빨리 걸린다. 정렬 한 줄이 탐색량을 크게 줄인다.

로드 밸런싱 — 파이썬 코드

load_balancing.py

```python
def solve_load_balancing(jobs, K):
    N = len(jobs)
    jobs.sort(reverse=True)        # 큰 작업부터 (가지치기 강화)
    server_loads = [0] * K
    ans = float("inf")

    def dfs(idx):
        nonlocal ans
        if idx == N:               # 모든 작업 분배 완료
            ans = min(ans, max(server_loads))
            return

        # [가지치기 1] 이미 ans보다 나쁜 서버가 있으면 중단
        if max(server_loads) >= ans:
            return

        for i in range(K):
            # [가지치기 2] 없으면 ans 이상이 되는 서버는 건너뜀
            if server_loads[i] + jobs[idx] >= ans:
                continue

            server_loads[i] += jobs[idx]
            dfs(idx + 1)
            server_loads[i] -= jobs[idx]

            # [가지치기 3] 대칭성 제거
            if server_loads[i] == 0:
                break

    dfs(0)
    return ans
```

run.py

```python
jobs = [1, 2, 3, 4, 5, 6, 7]
K = 3
print(solve_load_balancing(jobs, K))
```

OUTPUT

```
10
```

이론적 하한

전체 합 28, 평균 9.33 → 하한 10. 실제 최적값과 일치한다.

최적 분배

[7,3] [6,4] [5,2,1] → 부하 10, 10, 8

[가지치기 3] 대칭성(symmetry) 제거

서버에는 이름이 없고 성능도 같다 → 서버 0에 주든 서버 1에 주든 결과가 같다.

예를 들어 jobs = [7, 6, 5], K = 3, 초기 server_loads = [0, 0, 0] 인 상태에서 작업 7을 배정한다고 하자.

서버 0 ← 7

[7, 0, 0]

기준이 되는 배정

서버 1 ← 7

[0, 7, 0]

본질적으로 동일하다

✂

서버 2 ← 7

[0, 0, 7]

본질적으로 동일하다

✂

빈 서버에 작업을 배정한 뒤 break → 중복 탐색이 3에서 1로 줄어든다

같은 모양의 상태를 세 번 탐색하던 것을 한 번으로 줄인다. K가 커질수록 이 절약은 곱해져서 커진다.

대칭성 제거는 한정값 계산 없이도 쓸 수 있는 가장 값싼 가지치기다.

8.8

8장을 마치며

분기한정으로도 감당할 수 없는 문제들이 있다.
그것들의 정체는 무엇일까?

§8.8 8장을 마치며

8장 핵심 정리

분기한정 = 백트래킹 + 수학적 예측

단순 조건 검사를 넘어, 한정값이라는 숫자로 미래를 계산한다.

3박자 — 분기, 한정, 가지치기

나누고, 예측하고, 기대에 못 미치면 잘라 낸다. 이 순환이 전부다.

최대화 → 상한(UB), 최소화 → 하한(LB)

어느 쪽이든 "현실보다 유리하게" 낙관적으로 계산해야 안전하다.

Best-First + 우선순위 큐가 표준

유망한 노드부터 꺼내면 좋은 해를 일찍 찾고, 기준이 올라가 가지치기가 더 잘 된다.

최악 $O(2^n)$, 실제로는 훨씬 빠르다

복잡도 표기는 최악의 이야기다. 좋은 한정값이 실효 성능을 결정한다.

DP와는 용도가 다르다

용량이 작은 정수면 DP, 용량이 크거나 실수면 분기한정이 유리하다.

§8.8 8장을 마치며

분기한정의 한계와 다음 장 예고

분기한정이 "항상" 빠를까? 가지치기 조건이 느슨하면 거의 모든 노드를 방문해야 할 수도 있다.

도시 20개 TSP

몇 초에서 몇 분이면 최적해를 얻을 수 있다.

도시 50개 TSP

우주가 멸망해도 답을 구하지 못한다.

"도대체 이 문제들의 정체는 무엇일까?"

다음 장 — P vs NP 문제

입력 크기가 조금만 커져도 현실적인 시간 안에 답을 구할 수 없는 문제들이 있다. 무엇을 빠르게 풀 수 있고 무엇은 끝내 풀 수 없는가 — 9장에서 그 경계를 다룬다.

알고리즘 설계의 마지막 질문은 "더 빠른 방법"이 아니라 "빠른 방법이 존재하는가"이다.

EXERCISE

연습문제 1 ~ 5

1 분기한정 기법과 백트래킹 기법의 공통점과 차이점을 서술하라. (키워드: 상태 공간 트리, DFS, BFS, 최적화 문제, 결정 문제)

2 한정 함수(bounding function)의 역할을 설명하고, 유망하지 않은 노드를 판별해 가지치기하는 일반적인 조건식을 최소화 문제 기준으로 서술하라.

3 분기한정 기법을 구현할 때 우선순위 큐를 사용하는 이유를 탐색 전략의 관점에서 설명하라.

4 (0/1 배낭) W = 10kg, 아이템 4개 — (2kg, $40), (5kg, $30), (10kg, $50), (5kg, $10). 루트 노드의 상한값(UB)을 분할 가능 배낭 방식으로 계산하라.

5 (TSP) 인접 행렬 [∞,10,15,20] / [5,∞,9,10] / [6,13,∞,12] / [8,8,9,∞] 에 대해 최단 2간선 합을 이용하여 루트 노드의 LB를 계산하라.

EXERCISE

연습문제 6 ~ 10

6 Best-First Search에서 큐의 노드 한정값이 A(25), B(18), C(30), D(18)일 때 다음에 확장할 노드는? (최소화 문제, B가 D보다 먼저 생성되었다)

7 분기한정의 최악 시간 복잡도와, 그럼에도 실무에서 사용하는 실용적인 이유를 설명하라.

8 한정 함수가 지나치게 큰 값(loose bound)을 산출하면 알고리즘 성능에 어떤 영향이 있는가? 가지치기 효율과 방문 노드 수의 관점에서 설명하라.

9 Best-First Search는 최적해를 빨리 찾지만 DFS에 비해 메모리 사용량이 기하급수적으로 증가할 위험이 있다. 그 이유를 우선순위 큐의 특성과 관련지어 설명하고, 완화 전략을 하나 제안하라.

10 0/1 배낭 문제에서 아이템을 가성비(v/w) 내림차순으로 정렬하는 전처리가 가지치기 효율에 미치는 결정적인 영향을 설명하라.

CHAPTER 08

Q & A

가장 유망한 질문부터 꺼내 봅시다.
좋은 질문 하나가 나머지를 전부 정리해 줍니다.