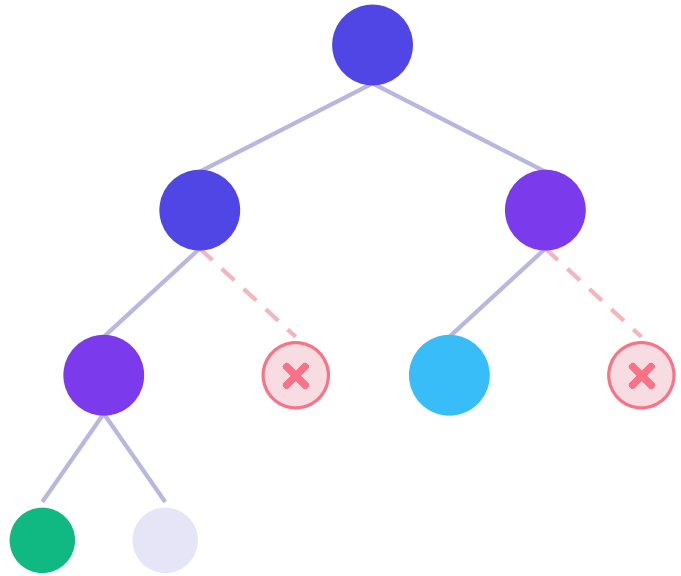
그림으로 쉽게 설명하는 파이썬 알고리즘

CHAPTER 07

백트래킹 기법

Backtracking Techniques

막다른 벽이 나타나면 즉시 발길을 돌린다



가지치기된 상태 공간 트리

CONTENTS

이번 장에서 배울 내용

7.1

탐색의 기초

상태 공간 트리

7.2

DFS와 BFS

탐색의 두 가지 엔진

7.3

백트래킹

DFS에 판단력을 더하다

7.4

카드 뽑기

템플릿 첫 적용

7.5

부분 집합의 합

넣을까, 말까

7.6

N-Queen

백트래킹의 교과서

7.7

그래프 색칠하기

m개의 색으로 칠하기

7.8

다른 방법과 비교

DFS와의 차이, 그리고 한계

7.9

코딩 테스트

실전에서의 백트래킹

"막다른 벽이 나타나면 즉시 발길을 돌려(back), 오던 길로 되돌아가(tracking) 다른 길을 찾는다"

7.1

탐색의 기초

정답으로 가는 공식을 바로 계산할 수 없다면,
가능한 선택을 체계적으로 살펴보는 수밖에 없다.

7.1

탐색이 필요한 문제들

어떤 문제는 정답으로 가는 공식을 바로 계산할 수 없다. 가능한 모든 선택을 하나씩 시도해 보며 답을 찾아야 한다.

비밀번호

각 자리에 어떤 숫자를 넣어야 할지 미리 알 수 없는 문제.

체스판

말을 어디에 놓아야 서로 공격하지 않는지 결정하는 문제.

선택 문제

여러 선택지 중 무엇을 고르고 무엇을 버릴지 정하는 문제.

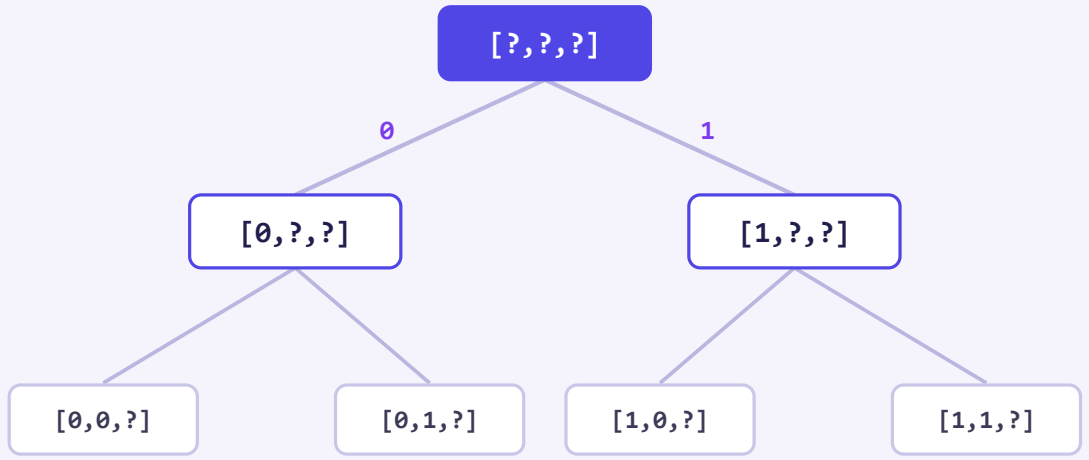
탐색(search)이란?

문제의 해답을 찾기 위해 가능한 선택의 공간을 체계적으로 살펴보는 과정이다. 여기서 핵심은 "정답을 찾았는가"보다 "정답이 아닌 길을 얼마나 똑똑하게 피했는가"에 있다.

탐색 알고리즘의 성능은 방문한 노드 수로 결정된다 — 덜 가보는 쪽이 이긴다.

상태 공간 트리 (state space tree)

선택의 결과를 트리로 표현한 가상의 구조다. 예를 들어 각 자리가 0 또는 1인 3자리 비밀번호는 다음과 같은 이진 트리가 된다.



… 각 단말 노드가 정답 또는 실패가 된다

구성 요소

상태(state): 문제 해결 과정의 한 시점 — 예: [0, 1, ?]
간선(edge): 한 상태에서 다른 상태로의 이동
루트(root): 시작 상태 [?, ?, ?]
단말 노드(leaf): 정답 또는 실패

트리를 미리 다 만들지 않는다!

탐색하면서 필요한 순간에만 다음 가지를 만들고, 확인하고, 필요 없으면 지운다. 손전등을 비추는 곳만 잠시 보이는 셈이다.

메모리에 존재하는 것은 지금 내려가고 있는 한 줄기 경로뿐이다 — 트리 전체가 아니다.

맹목적 탐색 vs 정보 기반 탐색

맹목적 탐색 (blind search)

목표의 위치에 대한 정보 없이 탐색한다.

DFS (깊이 우선) — "한 놈만 팬다"

갈림길에서 한쪽을 골라 끝까지 간 뒤 되돌아온다. 백트래킹의 기본 엔진.

BFS (너비 우선) — "돌다리도 두들겨"

한 발자국씩 모든 방향을 고르게 확인하며 퍼져 나간다.

수학적으로 반드시 답을 찾는다

정보 기반 탐색 (informed search)

목표에 대한 힌트(정보)를 활용한다.

휴리스틱 (heuristic)

경험적 추측을 쓴다. "줄이 긴 식당이면 맛집일 거야" 같은 판단.

A* 알고리즘

g(지금까지의 비용) + h(예상 비용)를 합쳐 유력한 길부터 본다.

빠르지만 힌트가 틀리면 헤맨다

이번 장의 주제 — 맹목적 탐색(DFS) + 가지치기 = 백트래킹

7.2

DFS와 BFS

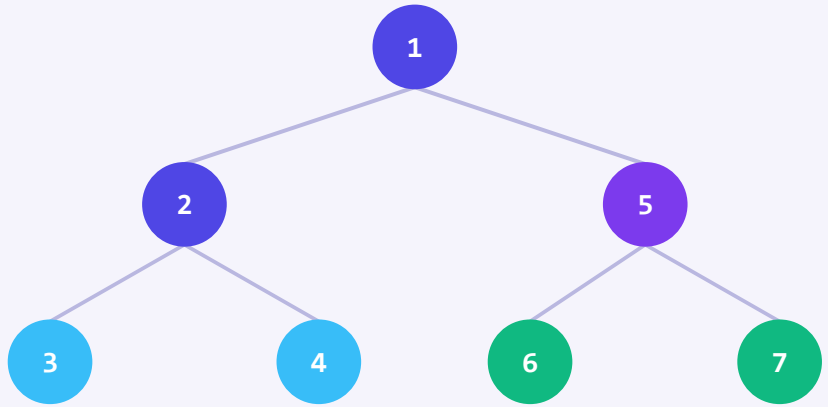
깊이로 파고들 것인가, 너비로 퍼질 것인가.
자료구조 하나가 성격을 완전히 바꾼다.

7.2

§7.2 DFS와 BFS

깊이 우선 탐색 (DFS) — "한 놈만 팬다"

깊이(depth)를 최우선으로 추구하는 전략이다. 갈림길에서 한쪽을 골라 끝까지 직진하고, 막다른 골목을 만나야만 되돌아온다.



번호 = 방문 순서 · 한 갈래를 끝까지 파고든 뒤 되돌아온다

① 한쪽을 골라 끝까지
갈림길에서 하나를 선택해 막다른 곳까지 직진한다.

② 막히면 직전 갈림길로
더 갈 곳이 없으면 바로 앞 갈림길로 복귀한다.

③ 가보지 않은 길로 다시
아직 안 가본 가지를 골라 또 깊숙이 내려간다.

후입선출(LIFO)과 완벽히 일치한다 → 스택 또는 재귀 함수로 구현한다.

장점 — 구현이 쉽고 메모리를 적게 쓴다 (지금 가는 경로만 기억) · 단점 — 무한 경로에 빠질 수 있고 최단 경로를 보장하지 않는다

DFS 유사코드 — 재귀와 스택

재귀 호출 방식 (recursive)

dfs_recursive.pseudo

```
Algorithm DFS_Recursive(G, u)
  1. 정점 u를 "방문함(visited)"으로 표시
  2. u와 인접한 모든 정점 v에 대하여:
       if v가 아직 방문하지 않은 정점이라면:
           DFS_Recursive(G, v)      // 재귀 호출
```

스택 방식 (iterative)

dfs_iterative.pseudo

```
Algorithm DFS_Iterative(G, start)
  1. 스택 S에 start를 push
  2. while S가 비어있지 않은 동안:
       u <- S.pop()
       if u가 미방문이면:
           u를 방문 표시
           for each neighbor v of u:
               if v가 미방문이면: S.push(v)
```

핵심 구조 — 스택 / 재귀 = 후입선출(LIFO)

마지막에 들어온 갈림길이 가장 먼저 되돌아간다. 재귀 함수의 호출 스택이 이 구조를 공짜로 제공하기 때문에, 재귀로 쓰면 스택을 직접 만들 필요조차 없다.

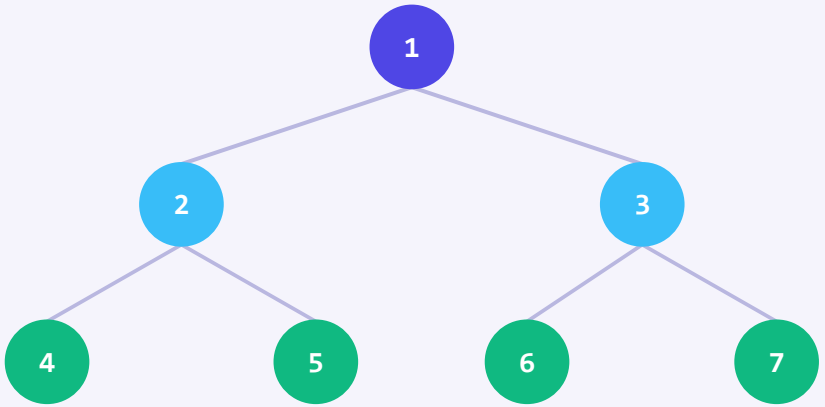
O(V + E)

시간 복잡도

정점 V개를 한 번씩 방문하고, 각 정점에서 인접 간선을 훑는다. 간선 전체를 합치면 E번이다.

너비 우선 탐색 (BFS) — "돌다리도 두들겨 보고"

호수에 돌을 던졌을 때 물결이 퍼져 나가듯, 거리 1 → 거리 2 → 거리 3 순서로 탐색 영역을 넓힌다.



같은 깊이를 전부 확인한 다음에야 한 단계 더 내려간다

bfs.pseudo

```
Algorithm BFS(G, start)
  1. 큐 Q에 start를 enqueue, 방문 표시
  2. while Q가 비어있지 않은 동안:
       u <- Q.dequeue()
       for each neighbor v of u:
           if v가 미방문이면:
               v를 방문 표시
               Q.enqueue(v)
```

장점 — 최단 경로 보장

가까운 곳부터 훑으므로 처음 만난 목표가 곧 최단 경로다.

단점 — 메모리가 폭발한다

선입선출(FIFO) 큐에 같은 깊이의 모든 노드를 담아야 한다. 넓이가 넓은 트리에서는 큐에 쌓이는 노드 수가 감당하기 어려워진다.

왜 백트래킹은 DFS를 선호하는가

이유 ① 되돌아가기의 본질적 적합성

백트래킹의 핵심은 "가다가 아니면 돌아온다"이다. DFS는 깊이 내려갔다가 조건이 어긋나면 부모로 복귀하는데, 이것이 재귀의 호출과 리턴에 그대로 대응된다. 상태를 저장하고 복구하는 흐름이 자연스럽게 만들어진다.

BFS는 어떤가

문어발식으로 동시에 여러 갈래를 벌려 놓기 때문에, 어느 지점으로 되돌아가야 하는지 관리하는 일 자체가 복잡해진다.

이유 ② 압도적인 메모리 효율

N-Queen (N = 14) 을 예로 들어 보자.

BFS — 수억 개 노드를 큐에

14번째 줄까지 가려면 그 이전 층의 모든 배치를 들고 있어야 한다. RAM이 버티지 못한다.

DFS — 스택에 딱 14개

현재 내려가고 있는 경로만 들고 있으면 된다. 공간 복잡도가 깊이에 선형이다.

결론 — 모든 경우의 수를 따져야 하고 트리가 거대하다면 무조건 DFS

백트래킹이 DFS를 엔진으로 삼는 이유는 취향이 아니라 필연이다. 되돌아가는 동작과 메모리 사용량, 두 측면 모두에서 DFS만이 감당할 수 있다.

DFS vs BFS 한눈에 비교

항목	DFS	BFS
탐색 순서	깊이 우선 (세로)	너비 우선 (가로)
자료구조	스택 / 재귀	큐 (queue)
메모리	**O(깊이) — 적음**	O(너비) — 많음
최단 경로	보장 안 됨	**보장됨**
무한 경로	빠질 수 있음	안전
백트래킹	**찰떡궁합**	비효율적
적합한 문제	경우의 수, 조합 탐색	최단 거리, 레벨 탐색

둘은 우열이 아니라 용도의 문제다

최단 거리를 물으면 BFS, 모든 조합을 물으면 DFS다. 이 장에서 다루는 문제는 전부 후자에 속한다.

7.3

백트래킹

DFS는 성실하지만 고집이 세다.
여기에 판단력을 더하면 백트래킹이 된다.

7.3

DFS의 문제점 — 맹목적인 비효율

DFS는 고집이 너무 세다. 한 번 길을 정하면 바닥을 칠 때까지 멈추지 않는다.

최악의 시나리오 — 100단계 깊이, 각 단계 2갈래

2번째 단계에서 이미 답이 틀렸다는 것이 명백한데도, DFS는 남은 98단계를 꾸역꾸역 내려가며 2^{98}개 노드를 전부 방문하고 나서야 "아, 여기가 아니네?" 하고 돌아온다.

2^{98}	∞	가지치기
방문 노드 수	**최악의 경우**	**해결책**
무의미한 탐색	무한 루프 위험	될 놈만 간다

틀린 것을 일찍 아는 것이 곧 성능이다

탐색 알고리즘의 성능은 정답을 얼마나 빨리 찾느냐가 아니라, 오답 가지를 얼마나 일찍 잘라 내느냐로 결정된다.

가지치기 (pruning)

백트래킹 = DFS + 가지치기

① 유망성 검사 (promising)

"이 길, 가망 있어?" — 현재 상태가 해답이 될 가능성이 있는지 조건을 검사하는 단계다.

예시

N-Queen → 퀸이 서로 공격하는가?
부분 집합의 합 → 지금 합이 목표를 넘었는가?

② 가지치기 (pruning)

"가망 없으면 즉시 Back!" — 유망하지 않으면 자식 노드로 더 이상 내려가지 않는다.

효과

가지 하나를 자르면 그 아래 매달린 수백·수억 개의 자식 노드가 통째로 사라진다.

순서가 뒤집힌다 — 이것이 전부다

DFS는 "방문 → 검사" 순서로 일단 가고 본다. 백트래킹은 "검사 → 방문" 순서로 될 놈만 간다. 코드로 보면 조건문 하나의 위치가 바뀐 것뿐이지만, 방문하는 노드 수는 자릿수가 달라진다.

백트래킹 표준 템플릿

backtrack_template.pseudo

```
Algorithm Backtrack(v)

  if is_solution(v) is TRUE then     // ① 정답 확인
      Print_Solution(v)
      return

  if is_promising(v) is FALSE then   // ② 유망성 검사
      return

  for each child u of v do           // ③ 후보 탐색
      Make_Move(u)                   //    상태 기록 (마킹)
      Backtrack(u)                   //    재귀 호출
      Unmake_Move(u)                 //    상태 복구 (백트래킹)
```

① 정답 확인

목표 깊이에 도달했는가? 답을 찾았다면 결과를 저장하고 종료한다.

② 유망성 검사

현재 상태가 유효한가? False면 즉시 리턴 — 이것이 가지치기다.

③ 후보 탐색

자식마다 선택 → 탐색 → 철회를 반복한다. 세 줄이 한 묶음이다.

핵심은 세 줄의 묶음 — Make_Move → Recursive Call → Unmake_Move

선택하고, 그 선택 아래를 끝까지 탐색하고, 돌아와서 선택을 되돌린다. 이 패턴은 이 장의 모든 문제에서 글자 그대로 반복된다.

이 뼈대만 외워 두면 문제마다 바뀌는 것은 is_promising 하나뿐이다.

상태 변경과 복구 — 백트래킹의 심장

Make Move (선택)

"이 길로 가보자"
발자국을 남긴다
체스판에 퀸을 놓는다
visited = True

Recursive Call (탐색)

"끝까지 가봐"
다음 단계로 깊이 진입한다
재귀 함수를 호출하고
결과를 확인한 뒤 돌아온다

Unmake Move (철회)

"이 길 아니네, 되돌아가자"
발자국을 지운다
퀸을 다시 들어낸다
visited = False

복구를 빠뜨리면 결과가 통째로 어긋난다

이전 선택의 흔적이 남아 있으면 다음 탐색이 그 영향을 그대로 받는다. 퀸을 들어내지 않은 채 다른 칸을 시도하면, 있지도 않은 퀸과 충돌 판정이 난다.

이 "선택 → 탐색 → 철회" 패턴은 모든 백트래킹 문제에 동일하게 적용된다.

7.4

간단 예제 — 카드 뽑기

7.4

1, 2, 3 카드에서 2장을 뽑아 나열하기.
템플릿을 가장 작은 문제에 먼저 얹어 본다.

§7.4 카드 뽑기

숫자 카드 1, 2, 3 에서 2장 뽑기

카드 1, 2, 3 중 2장을 뽑아 순서대로 나열하는 모든 경우를 찾는다. 같은 카드를 두 번 쓸 수는 없다.

[1,2] [1,3] [2,1] [2,3] [3,1] [3,2] → 총 6가지

pick_cards.py

```
selected = []

def pick_cards(count):
    if count == 2:                # ① 기저 조건 (정답 확인)
        print(selected)
        return

    for card in [1, 2, 3]:        # ② 후보 탐색
        if card not in selected:  #    유망성 검사 (가지치기)
            selected.append(card) # ③ 상태 변경
            pick_cards(count + 1) # ④ 재귀 호출
            selected.pop()        # ⑤ 상태 복구

pick_cards(0)
```

① is_solution

count == 2 — 두 장을 다 뽑았으면 출력하고 종료

② 유망성 검사

card not in selected — 이미 쓴 카드면 진입하지 않는다

③ Make_Move

selected.append(card) — 선택을 기록한다

④ Backtrack(u)

pick_cards(count + 1) — 한 단계 더 깊이

⑤ Unmake_Move

selected.pop() — 돌아와서 선택을 되돌린다

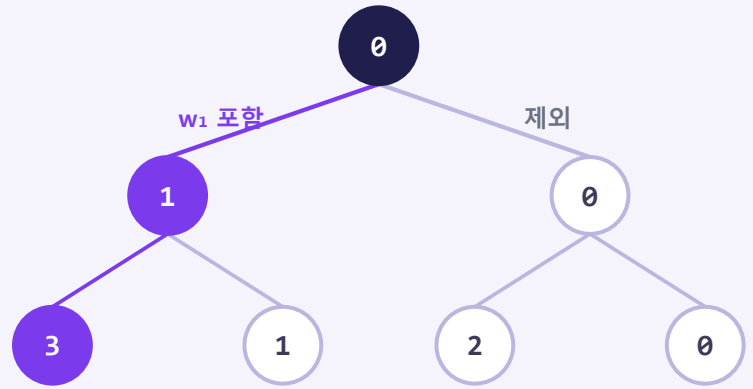
7.5

부분 집합의 합

원소마다 넣을까 말까를 정하면
상태 공간 트리는 깔끔한 이진 트리가 된다.

부분 집합의 합 (subset sum)

양의 정수 집합 W = {w_1, w_2, …, w_n}과 목표 정수 K가 주어질 때, 합이 정확히 K가 되는 부분 집합을 구하라.



노드 안의 수 = 지금까지의 합 · 왼쪽은 포함, 오른쪽은 제외

깔끔한 이진 트리

각 원소마다 포함(왼쪽)과 제외(오른쪽) 두 갈래뿐이므로 트리 모양이 단순하다.

왜 DP가 아니라 백트래킹인가

선택된 숫자들을 직관적으로 뽑아낼 수 있고, K가 매우 크거나 실수가 섞이면 DP 표를 만들기 어렵다. N이 작으면(N ≤ 20) 백트래킹이 매우 빠르다.

가지치기 ① "이미 넘쳤어!"

current_sum + w[i] > K → 되돌아감 (양수만 있으므로 더할수록 커질 뿐)

가지치기 ② "다 더해도 안 돼!"

current_sum + 남은 총합 < K → 되돌아감 (최대로 더해도 K에 못 미침)

부분 집합의 합 — 파이썬 구현

subset_sum.py

```python
def subset_sum(idx, current_sum):
    global found

    if current_sum > Target:      # 가지치기 1
        return
    if current_sum == Target:     # 기저 조건: 성공!
        found = True
        print("해 찾음:", selected_items)
        return
    if idx == N:                  # 모든 원소 검토 완료
        return

    # 선택 1: 현재 원소를 포함한다
    selected_items.append(arr[idx])
    subset_sum(idx + 1, current_sum + arr[idx])
    if found: return
    selected_items.pop()          # 백트래킹 (상태 복구)

    # 선택 2: 현재 원소를 제외한다
    subset_sum(idx + 1, current_sum)
```

run.py

```python
arr = [1, 2, 5, 8]
Target = 7; N = 4
selected_items = []; found = False
subset_sum(0, 0)
```

OUTPUT

```
해 찾음: [2, 5]
```

핵심 패턴 — 이진 분기

포함 → 재귀 → pop()으로 복구, 그다음 제외 → 재귀. 두 갈래를 순서대로 시도하는 것이 전부다.

가지치기 두 줄(합 초과 검사, 원소 소진 검사)이 없으면 2^n 전부를 방문한다.

7.6

N-Queen 문제

N×N 체스판에 N개의 퀸을
서로 공격할 수 없도록 배치하라.

7.6

§7.6 N-Queen

N-Queen 문제 정의

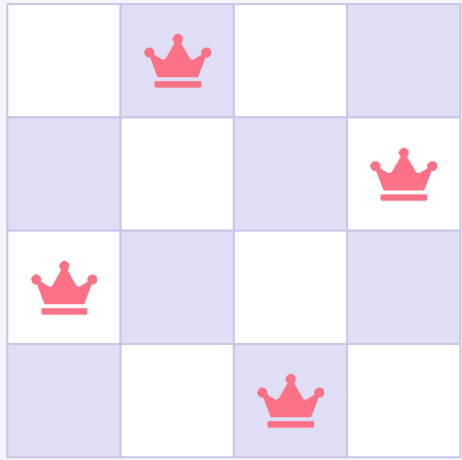
퀸은 가로·세로·대각선 방향으로 거리 제한 없이 공격한다. 따라서 어떤 두 퀸도 같은 행, 같은 열, 같은 대각선에 있으면 안 된다.

~44억	~1677만	수천 번	92
N=8 무식한 접근	**행 제약 적용**	**백트래킹 적용**	**8-Queen**
$_{64}C_{8}$ 조합	$8^8 = N^N$	가지치기 효과	총 해답 수

백트래킹 전략

첫 번째 줄에 퀸을 놓고 두 번째 줄을 시도한다. 충돌이 발견되면 즉시 퀸을 들어내고 옆 칸으로 옮긴다. "이 길은 글렀어"라고 판단하는 순간 그 아래 가지가 통째로 사라진다.

가지치기 하나로 1,677만 번이 수천 번이 된다 — 실제 방문 노드 수가 획기적으로 줄어든다.



4-Queen 정답 배치 (해답 2가지 중 하나)

```
row = [1, 3, 0, 2]
```

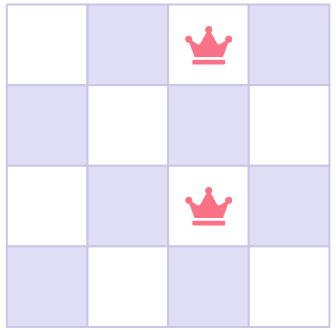
§7.6 N-Queen

N-Queen 가지치기 조건

각 행마다 퀸을 하나씩 놓는 방식으로 구현하면 행 충돌은 애초에 생기지 않는다. 남은 검사는 열과 대각선 둘뿐이다.

열(column) 검사

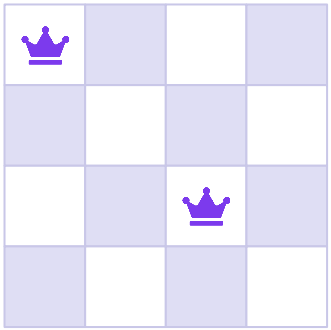
`cols[i] == cols[k]`



두 퀸이 같은 열에 있는지 확인한다.
예: 0행 2열과 2행 2열 → 세로로 마주 본다 → 탈락!

대각선(diagonal) 검사

`|cols[i] - cols[k]| == |i - k|`



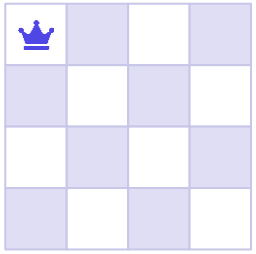
열의 차이 절댓값이 행의 차이 절댓값과 같으면 대각선상이다.
예: (0,0)과 (2,2) → |0-2| == |0-2| → 탈락!

둘 중 하나라도 걸리면 "유망하지 않다" → 더 깊이 들어가지 않고 부모로 되돌아간다.

§7.6 N-Queen

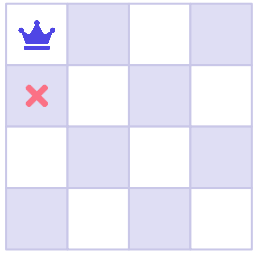
4-Queen — 놓고, 물리는 과정

① (0,0) 퀸 배치



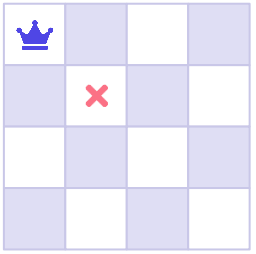
성공!

② (1,0) 세로 충돌



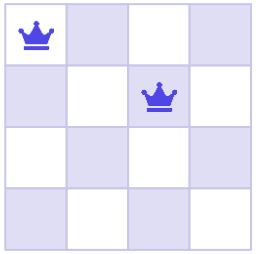
가지치기!

③ (1,1) 대각선 충돌



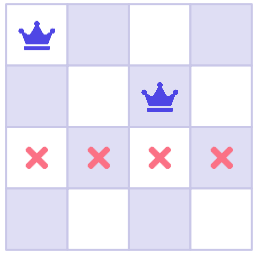
가지치기!

④ (1,2) 안전!



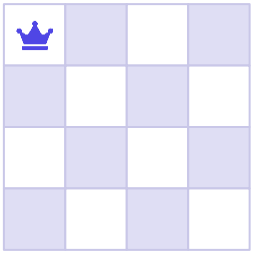
탐색 진행

⑤ 3행에서 난관



놓을 곳이 없음

⑥ Backtrack!



(1,2) 퀸 철회

최종 정답 — (0행 1열), (1행 3열), (2행 0열), (3행 2열)

N-Queen 유사코드

n_queens.pseudo

```
Algorithm n_queens(i)
  if promising(i) then
    if i == n then
      print_solution(col)      // 모든 행에 퀸 배치 완료
    else
      for j = 1 to n do
        col[i+1] <- j          // 다음 행의 j열에 퀸 배치
        n_queens(i + 1)        // 재귀 호출
```

promising.pseudo

```
Algorithm promising(i)
  k <- 1, flag <- true
  while k < i and flag do
    if col[i] == col[k] or              // 같은 열?
       abs(col[i]-col[k]) == abs(i-k)   // 대각선?
    then
      flag <- false
    k <- k + 1
  return flag
```

n_queens

행마다 모든 열을 시도한다. promising이면 다음 행으로 진행하고, 아니면 그 가지는 열지 않는다.

promising

이미 놓인 퀸들과 하나씩 비교한다. 같은 열인지, 대각선인지 — 둘 다 통과해야 true를 반환한다.

N-Queen 파이썬 구현

n_queens.py

```python
def n_queens(x):
    global count
    if x == N:                  # 모든 행에 퀸 배치 완료!
        count += 1
        return

    for col in range(N):        # 현재 행의 각 열을 시도
        row[x] = col
        if is_promising(x):     # 유망성 검사
            n_queens(x + 1)     # 다음 행으로 진행

def is_promising(x):
    for i in range(x):
        if row[x] == row[i]:
            return False                    # 같은 열
        if abs(row[x]-row[i]) == abs(x-i):
            return False                    # 대각선
    return True
```

run.py

```python
N = 8
row = [0] * N
count = 0
n_queens(0)
print(count)
```

OUTPUT

```
92
```

1차원 배열의 마법

row[i] = j 는 "i행 j열에 퀸"을 뜻한다. 2차원 배열이 필요 없다.

암묵적 백트래킹 — 되돌아가는 코드가 보이지 않는 이유

is_promising이 False면 재귀를 호출하지 않고 for문이 다음 col로 넘어간다. 이 동작이 곧 부모로 되돌아가는 것이며, 동시에 row[x]가 덮어써지므로 상태 복구까지 함께 일어난다.

§7.6 N-Queen

N-Queen 코드 핵심 분석

row 리스트의 마법

2차원 board[N][N] 대신 1차원 row[N]을 쓴다. "한 행에 퀸이 하나"라는 전제를 활용한 것이며, 공간 복잡도가 O(N²) → O(N)이 된다.

abs()로 대각선 검사

abs(row[x]-row[i]) == abs(x-i) 한 줄로 기울기 +1과 -1인 두 대각선을 동시에 처리한다.

암묵적 백트래킹

명시적인 return이 없어 보이지만, is_promising이 False면 재귀를 호출하지 않으므로 for문의 다음 col로 자동 이동한다 — 이것이 곧 부모로 되돌아가는 행위다.

시간 복잡도

최악의 경우는 완전 탐색과 같다. 그러나 실제로는 가지치기 덕분에 훨씬 적은 노드만 방문한다. N=8에서 1,600만 번이 수천 번으로 줄어든다.

"자료구조를 바꿔 검사를 싸게 만든다"는 것도 가지치기만큼 중요한 최적화다.

§7.6 N-Queen

N-Queen 해답 개수

N이 커질수록 해답의 수 자체가 급격히 늘어난다. 완전 탐색으로는 셀 수조차 없고, 백트래킹이 사실상 유일한 선택지가 된다.

N	4	5	6	7	8	9	10	11	12
해답 수	2	10	4	40	**92**	352	724	2680	**14200**

해답 수가 폭증한다

N=8에서 92개, N=12에서 14,200개. 개수 자체가 지수적으로 늘어난다.

단조 증가가 아니다

N=5의 10개에서 N=6은 4개로 오히려 줄어든다. 판의 크기와 해답 수는 단순 비례하지 않는다.

그래서 가지치기다

모든 배치를 만들어 보고 세는 방식으로는 N=12조차 감당하기 어렵다.

N이 커질수록 완전 탐색과 백트래킹의 격차는 더 벌어진다.

7.7

그래프 색칠하기

7.7

인접한 정점끼리 같은 색을 쓸 수 없다.
m개의 색으로 전부 칠할 수 있을까?

그래프 색칠하기 문제

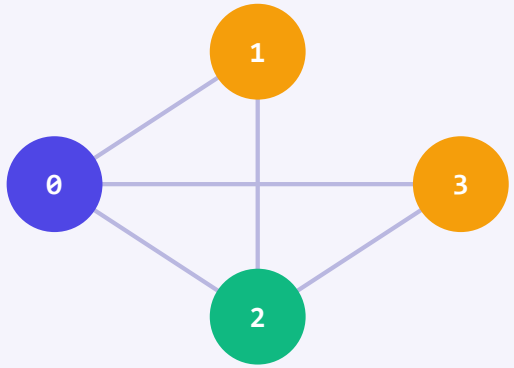
지도에서 그래프로

정점(vertex) = 각 나라 또는 지역
간선(edge) = 국경을 맞댄 두 나라

"인접한 정점끼리는 같은 색을 가질 수 없다"

m-색칠 결정 문제

N개 정점과 간선 정보를 가진 그래프 G, 그리고 m개의 색상이 주어진다
.
간선 (i, j)가 존재하면 color[i] ≠ color[j] 여야 한다.



정점 4개 · 간선 5개
색 3개로 칠한 결과

colors = [1, 2, 3, 2]

$$m^n$$

완전 탐색의 경우의 수

정점 20개에 색 3개면 약 34억 번 — 백트래킹이 필수다.

그래프를 칠할 수 있는 최소 m값을 색채 수(chromatic number)라고 한다.

그래프 색칠 — 유망성 검사

현재 i번째 정점을 칠할 때, 나와 연결된 이웃 중 이미 색칠된 정점과 색이 겹치는지만 확인하면 된다.

연결 안 됨	간선이 없는 정점끼리는 색이 같아도 아무 문제가 없다.	**통과**
미래 정점	연결되어 있지만 아직 칠하지 않은 정점은 지금 판단할 대상이 아니다.	**보류**
과거 정점	연결되어 있고 이미 칠해진 정점이라면 반드시 색이 달라야 한다.	**검사!**

promising.pseudo

```
Algorithm promising(k)
  for i = 1 to k - 1 do
    if W[k][i] is true AND vcolor[k] == vcolor[i] then
      return false     // 인접한데 색이 같음 -> 유망하지 않음
  return true
```

그래프 색칠 알고리즘

m_coloring.pseudo

```
Algorithm m_coloring(k)

  if k > n then
    print_solution(vcolor)    // 모든 정점 색칠 완료
    return

  for color = 1 to m do
    vcolor[k] <- color        // k번 정점에 color를 칠해 본다
    if promising(k) then
      m_coloring(k + 1)       // 유효하면 다음 정점으로
```

① 정점에 색 칠하기

vcolor[k]에 색을 하나 대입한다 — 상태 변경.

② 인접 정점과 충돌?

promising으로 이미 칠한 이웃과 비교한다.

③ 충돌 없으면 전진

다음 정점으로 재귀 호출한다.

④ 모든 색 실패하면

for문이 끝나고 자연스럽게 부모로 되돌아간다.

상태 복구가 코드에 보이지 않는 이유

for문이 다음 색으로 넘어가면서 vcolor[k]를 새 값으로 덮어쓴다. 즉 반복문 자체가 Unmake_Move 역할을 대신한다. N-Queen의 row[x] = col도 같은 구조다.
다만 방문 배열처럼 덮어쓰기로 지워지지 않는 상태는 반드시 명시적으로 복구해야 한다.

그래프 색칠 파이썬 구현

m_coloring.py

```python
def m_coloring(node_index):
    if node_index == N:
        print("성공!", colors)
        return

    for c in range(1, m + 1):
        colors[node_index] = c
        if is_promising(node_index):
            m_coloring(node_index + 1)

def is_promising(i):
    for j in range(i):
        if graph[i][j] == 1 and colors[i] == colors[j]:
            return False
    return True
```

run.py

```python
N = 4
graph = [[0,1,1,1],
         [1,0,1,0],
         [1,1,0,1],
         [1,0,1,0]]
m = 3
colors = [0] * N
m_coloring(0)
```

OUTPUT

```
성공! [1, 2, 3, 2]
성공! [1, 3, 2, 3]
성공! [2, 1, 3, 1]  ...
```

인접 행렬

graph[i][j] = 1 이면 두 정점이 연결되어 있다는 뜻이다.

3색으로 칠하는 방법이 여러 가지 존재한다 — 모든 해를 나열하는 형태의 백트래킹이다.

백트래킹 문제들의 공통 패턴

문제는 달라 보여도 코드의 뼈대는 똑같다. 네 자리에 무엇을 끼워 넣느냐만 바뀐다.

단계	N-Queen	부분 집합 합	그래프 색칠
상태 변경	`row[x] = col`	`selected.append()`	`colors[i] = c`
유망성 검사	열 / 대각선 충돌?	합 > Target ?	인접 색이 같은가?
재귀 호출	`n_queens(x+1)`	`subset_sum(idx+1)`	`m_coloring(i+1)`
상태 복구	for문이 덮어씀	**`selected.pop()`**	for문이 덮어씀

"일단 저지르고, 수습 가능한지 보고, 안 되면 되돌린다"
상태 변경 → 유망성 검사 → 재귀 → 상태 복구. 이 네 박자를 문제마다 갈아 끼우는 것이 백트래킹 학습의 전부라고 해도 지나치지 않다.

pop()처럼 명시적 복구가 필요한 경우와 for문이 덮어쓰는 경우를 구분해야 실수가 줄어든다.

7.8

다른 방법들과의 비교

7.8

DFS와 무엇이 다른가,
그리고 백트래킹으로도 안 되는 것은 무엇인가.

DFS vs 백트래킹

DFS — "성실한 청소부"

모든 방을 순서대로 다 들어간다. 침대 밑과 장롱 안까지 샅샅이 뒤지고, 보물이 없다는 것을 직접 확인해야만 나와서 다음 방으로 이동한다.

모든 경우의 수 = 완전 탐색

방문 → 검사 (일단 가고 본다)

백트래킹 — "눈치 빠른 탐정"

방문을 열기 전에 문패를 먼저 본다. "관계자 외 출입 금지?" → 패스. "이 복도 끝은 막혀 있네?" → 되돌아감. 가망이 없으면 진입조차 하지 않는다.

백트래킹 = DFS + 가지치기

검사 → 방문 (될 놈만 간다)

백트래킹은 DFS라는 엔진에 브레이크를 잘 밟는 운전 기술이다

엔진을 바꾼 것이 아니라 운전 습관을 바꾼 것이다. 그래서 최악의 경우 성능은 DFS와 같지만, 현실의 문제에서는 비교할 수 없을 만큼 빠르다.

백트래킹의 한계

여전히 지수 시간 복잡도

가지치기를 아무리 잘해도 최악의 경우에는 DFS와 다를 바가 없다. N이 30, 50으로 커지면 백트래킹으로도 해결이 불가능해진다 — 지수 폭발 문제다.

탐색 순서의 한계

DFS 기반이므로 언제나 왼쪽 길부터 탐색한다. 정답이 오른쪽에 있다면 왼쪽의 거대한 오답 트리를 다 헤매고 나서야 발견하게 된다.

"유망한 쪽부터 먼저 가볼 수는 없을까?"

다음 단계 — 분기한정법 (branch and bound), 8장

우선순위 큐로 각 경로에 점수(비용 하한선)를 매기고, 가장 유망한 노드부터 먼저 방문한다. 미래를 예측해 점수를 매기는 이 수학적 기법으로 외판원 문제(TSP)에 도전한다.
백트래킹은 스택으로 순서가 고정되지만, 분기한정은 우선순위 큐로 유망한 순서를 만든다.

7.9

코딩 테스트 전략

재귀 설계 능력과 가지치기 감각을
한꺼번에 묻는 유형이다.

7.9

코딩 테스트에서의 백트래킹

출제 경향

삼성 SW 역량테스트는 시뮬레이션과 함께 백트래킹을 사실상 고정 출제한다. 카카오·네이버에서도 변별력 있는 문제로 자주 등장하며, 평가 포인트는 재귀 함수 설계 능력과 가지치기로 효율을 끌어올리는 감각이다.

핵심 역량

재귀 호출과 종료 조건 숙달
가지치기 조건을 스스로 설정하는 능력
순열과 조합을 생성하는 표준 패턴
상태 변경과 복구의 정확한 구현

대표 유형

순열 / 조합 생성
N-Queen 변형
미로 탐색 + 추가 조건
부분 집합 / 배낭 문제
스도쿠, 그래프 색칠

문제를 읽자마자 "상태는 무엇인가, 언제 가지를 자를 수 있는가" 두 가지를 먼저 적어 두면 절반은 끝난다.

CT 7.1 숫자 카드로 목표 수

실전 ① 숫자 카드로 목표 수 만들기

서로 다른 숫자 카드 N장을 순서를 고려해 나열한 수가 정수 K 이하가 되는 경우의 수를 구하라. 예: 카드 [1,2,3], K=300 → 13가지

card_number.py

```
def dfs(cur_val, picked):
    global count

    if picked > 0:
        if cur_val <= K:
            count += 1
        else:
            return              # 가지치기: K 초과시 중단

    for i in range(N):
        if not used[i]:
            used[i] = True
            dfs(cur_val * 10 + arr[i], picked + 1)
            used[i] = False     # 백트래킹 (상태 복구)
```

핵심 가지치기

cur_val > K 이면 뒤에 숫자를 더 붙일수록 값은 커지기만 한다. 그러므로 그 자리에서 탐색을 끊는다.

used 배열이 상태

카드를 뽑을 때 True, 돌아올 때 False. 이 두 줄이 선택과 철회다.

자릿수를 하나 늘릴 때마다 값이 10배가 되므로, 이 가지치기 한 줄이 탐색량을 대폭 줄인다.

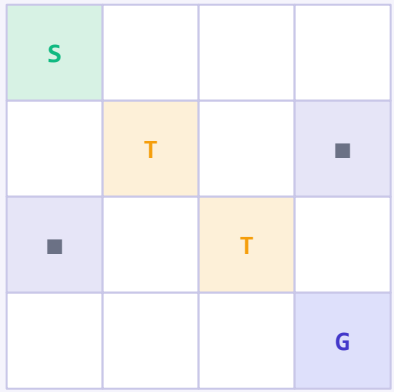
실전 ② 미로 속 보물 찾기

N×M 미로에서 (0,0)을 출발해 보물(T)을 정확히 K개 모으며 도착하는 경로 수를 구하라. 같은 칸은 다시 밟을 수 없다.

maze_treasure.py

```python
def dfs(x, y, treasure_count):
    global ans
    if grid[x][y] == "T":
        treasure_count += 1
    if treasure_count > K:
        return                      # 가지치기
    if x == N-1 and y == M-1:
        if treasure_count == K: ans += 1
        return

    visited[x][y] = True
    for i in range(4):
        nx, ny = x + dx[i], y + dy[i]
        if 0 <= nx < N and 0 <= ny < M:
            if not visited[nx][ny] and grid[nx][ny] != "1":
                dfs(nx, ny, treasure_count)
    visited[x][y] = False           # 백트래킹
```



S 출발 · T 보물 · ■ 벽 · G 도착

가지치기

보물이 K개를 넘는 순간 더 갈 이유가 없다.

결과 — 4×4, K=2 → 80가지 경로

연습문제 여섯 가지

1. 올바른 괄호 쌍

N쌍의 올바른 괄호 조합을 모두 출력한다. open < close 가 되는 순간 가지치기.

2. 휴대폰 자판 문자 조합

"23" 입력 → ad, ae, af, bd … 숫자 길이만큼 깊이 탐색한다.

3. 암호 만들기

C개 문자 중 L개를 뽑아 암호를 만든다. 모음 1개 이상 + 자음 2개 이상 조건.

4. 스도쿠 채우기

빈칸마다 1~9를 시도하고 행·열·3×3 박스로 유망성을 검사한다.

5. 연산자 끼워넣기

N개 수 사이에 연산자를 배치해 최댓값과 최솟값을 구한다.

6. 팰린드롬 분할

모든 조각이 회문이 되도록 분할한다. 회문이 아니면 즉시 백트래킹.

백트래킹 코딩 실전 팁

Tip 1 — 템플릿을 외운다

is_solution → is_promising → for each child → Make → Recurse → Unmake. 이 뼈대만 있으면 어떤 문제든 시작할 수 있다.

Tip 2 — 가지치기부터 생각한다

코드를 쓰기 전에 "어떤 조건이면 더 이상 탐색이 불필요한가"를 먼저 정리한다. 가지치기가 효율의 90%를 결정한다.

Tip 3 — 재귀 깊이를 조심한다

파이썬의 기본 재귀 제한은 1000이다. 깊은 탐색이 필요하면 sys.setrecursionlimit(10**7)을 반드시 넣는다.

Tip 4 — 상태 복구를 잊지 않는다

pop(), visited = False 같은 복구를 빠뜨리면 결과가 엉킨다. for문이 자동으로 덮어쓰는 경우도 있지만 명시적 복구가 안전하다.

sys.setrecursionlimit(10**7) 한 줄을 습관처럼 먼저 적어 두는 것도 좋은 방법이다.

백트래킹 적용 판단 가이드

이 문제에 백트래킹을 쓸 수 있을까? 세 질문에 모두 "예"라면 주저할 이유가 없다.

질문 ①	모든 경우를 따져 봐야 하는 문제인가?	Yes
질문 ②	경우의 수가 지수급 또는 팩토리얼급으로 늘어나는가?	Yes
질문 ③	중간에 "이건 안 된다"고 판단할 수 있는 조건이 존재하는가?	Yes

적합한 경우

N ≤ 20 정도의 작은 입력 · 조합과 순열 탐색 · 제약 조건 만족(CSP) 문제 · 존재 여부 판단 또는 모든 해 나열

부적합한 경우

N이 매우 큰 경우(1000 이상) · 최적화 문제(DP가 더 적합) · 가지치기 조건이 없는 경우(완전 탐색과 같아진다)

시간 복잡도 비교

문제	완전 탐색	백트래킹	비고
N-Queen	N^N	가지치기로 대폭 감소	N=8: 1,600만 → 수천
부분 집합 합	2^N	합 초과 시 조기 종료	N ≤ 20이면 빠르다
그래프 색칠	m^n	인접 충돌 시 즉시 차단	m이 작을수록 효과적
순열 생성	$N!$	중복 제거 가지치기	N ≤ 8 정도가 적합

최악은 같지만 평균은 전혀 다르다

백트래킹의 최악 시간 복잡도는 완전 탐색과 동일하다. 그러나 실제 입력에서는 가지치기가 대부분의 가지를 잘라 내기 때문에 비교할 수 없을 만큼 빠르게 동작한다.

따라서 복잡도 표기만 보고 "쓸모없다"고 판단하면 안 된다 — 실측이 답이다.

핵심 키워드 정리

키워드	의미
상태 공간 트리	선택의 결과를 트리로 표현한 가상의 구조
DFS	깊이 우선 탐색 — 스택 / 재귀, 백트래킹의 엔진
BFS	너비 우선 탐색 — 큐, 최단 경로 보장
백트래킹	DFS + 가지치기. "될 놈만 간다"
유망성 검사	현재 상태가 해답이 될 가능성이 있는지 조건을 검사하는 단계
가지치기(pruning)	유망하지 않은 경로를 조기에 차단하는 기법
Make / Unmake Move	상태 변경(선택)과 상태 복구(철회)

일곱 단어만 정확히 설명할 수 있으면 이 장의 내용은 손에 잡힌 것이다.

SUMMARY

7장 정리

상태 공간 트리

선택의 결과를 트리로 본다. 트리를 미리 만들지 않고 필요한 가지만 그때그때 생성한다.

DFS와 BFS

DFS는 스택·재귀로 깊이 우선, BFS는 큐로 너비 우선이며 최단 경로를 보장한다.

백트래킹 = DFS + 가지치기

"방문 → 검사"를 "검사 → 방문"으로 뒤집는다. 조건문 하나의 위치가 자릿수를 바꾼다.

선택 → 탐색 → 철회

Make_Move, 재귀 호출, Unmake_Move. 이 세 줄 묶음이 모든 문제에서 반복된다.

N-Queen과 그래프 색칠

열·대각선 충돌 검사, 인접 정점 색 충돌 검사. 문제마다 바뀌는 것은 유망성 검사뿐이다.

한계, 그리고 다음 장

여전히 지수 시간이고 탐색 순서가 고정된다. 8장 분기한정법이 유망한 순서를 만든다.

CHAPTER 07

Q & A

막다른 길이라고 느껴지면 바로 물어보세요.
되돌아가는 것도 탐색의 일부입니다.