7주차 풀이 읽기자료

풀이와 실행 과정

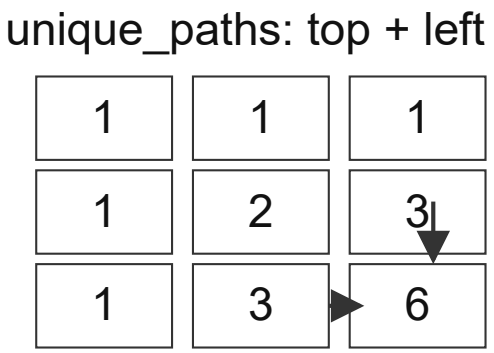
198	322	62	1143

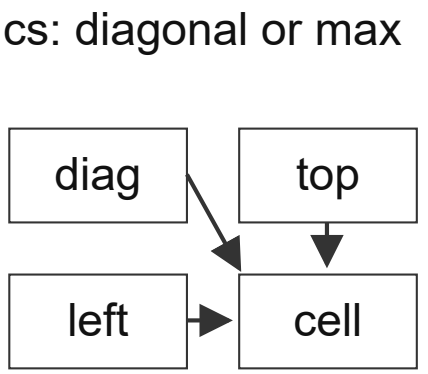
인접한 점검 항목을 함께 선택할 수 없을 때 최대 가치를 구하려면 어떤 정보를 남겨야 할까요? 격자 경로와 두 이벤트 기록의 공통 순서도 작은 부분 문제로 나눌 수 있습니다.

앞에서 계산한 최적값이나 개수를 저장하고 재사용합니다. 상태의 의미에 따라 합·최소·최대 중 연산을 결정합니다.

DP 상태의 뜻과 계산 순서

동적 계획법은 같은 부분 문제의 답을 저장해 재사용합니다. 먼저 “표 한 칸이 무엇의 답인가”를 정해야 초기값과 전이를 결정할 수 있습니다.





문제	상태의 뜻	결합 연산
인접하지 않은 최대 합	접두부에서 얻을 수 있는 최대 합	선택·건너뛰기의 max
동전	금액을 만드는 최소 개수	마지막 동전별 min
격자 경로	이 칸까지의 경로 수	위+왼쪽
공통 부분수열	두 접두부의 최대 공통 길이	일치 시 대각선+1, 아니면 max

시간은 상태 수 × 상태 하나의 계산량으로 셉니다. 공간은 동시에 보관하는 상태 수입니다. 이전 두 상태만 필요한 최대 합은 표를 없앨 수 있지만, 모든 표 구현이 자동으로 O(1) 공간인 것은 아닙니다.

198 · 풀이의 출발점과 실행 추적

LeetCode 198: House Robber · Medium

인접한 위치를 함께 고르지 않고 값의 합을 최대화합니다.

현재 값을 고르면 바로 앞 값은 고를 수 없습니다. 따라서 현재 값까지의 최댓값은 “직전까지의 최댓값을 유지”하거나 “두 칸 전까지의 최댓값에 현재 값을 더함” 중 큰 값입니다. 다음 결정에는 이 두 과거 결과만 필요합니다.

`nums=[3,1,4,2]`. old 값으로 오른쪽 식을 모두 계산한 뒤 갱신합니다.

value	old previous_two	old previous_one	새 previous_one
3	0	0	max(0,0+3)=3
1	0	3	max(3,0+1)=3
4	3	3	max(3,3+4)=7
2	3	7	max(7,3+2)=7

198 · 아이디어를 코드로 옮기기

```
def rob(nums):
    previous_two = previous_one = 0
    for value in nums:
        previous_two, previous_one = previous_one, max(previous_one, previous_two + value)
    return previous_one
```

처음의 두 0은 아무 원소도 고르지 않은 합입니다. 동시 대입 오른쪽은 모두 이전 값을 사용하므로, previous_two를 먼저 덮어써 필요한 값을 잃지 않습니다.

현재 위치를 선택하거나 건너뛰는 두 경우가 모든 가능한 선택을 포함합니다. `[2,1,1,2]`의 답 4는 양 끝을 골라 얻으며 짝수 위치 합·홀수 위치 합만 비교하면 놓칩니다.

점화식의 표와 두 변수의 대응

D[i]를 앞의 i개 값에서 얻는 최대 합으로 정의합니다. [3,1,4,2]에서 빈 접두부 D[0]=0입니다.

i와 현재 값	현재 값 제외	현재 값 포함	D[i]
1, 값 3	0	0+3	3
2, 값 1	D[1]=3	D[0]+1=1	3
3, 값 4	D[2]=3	D[1]+4=7	7
4, 값 2	D[3]=7	D[2]+2=5	7

값 4를 처리하기 직전 previous_two=D[1], previous_one=D[2]입니다. 계산 후에는 각각 D[2],D[3]이 되어 다음 반복의 의미를 유지합니다. 포함 경로에서 D[i−1]을 더하면 바로 앞 값을 함께 고를 수 있어 인접 금지 조건을 깨뜨립니다.

4를 포함하는 경우 이미 고른 앞부분의 최적값 3에 4를 더합니다. 최적값 3을 매번 처음부터 다시 구하지 않는 것이 DP의 재사용입니다.

198 · 정당성·시간·메모리의 근거

최적 선택은 현재 위치를 선택하거나 선택하지 않는 두 경우로 나뉩니다. 두 경우 모두 남은 접두부를 최적으로 풀어야 하므로 점화식이 모든 최적해를 포함합니다.

n개 값마다 덧셈 한 번, max 비교 한 번, 고정 개수 대입을 수행합니다. 숫자 연산을 O(1)로 보는 입력 범위에서 전체 시간은 O(n)입니다. 과거 최댓값은 이미 계산되어 있으므로 같은 접두부를 다시 탐색하지 않습니다.

입력이 커져도 두 최댓값과 현재 값만 보관하므로 보조 공간은 O(1)입니다. 모든 접두부 최댓값을 dp 리스트에 저장하면 O(n) 공간이지만 이 코드는 다음 단계에 필요한 두 값만 남깁니다. 실제 선택 위치 복원에는 추가 기록이 필요합니다.

[2,1,1,2]에서 짝수 위치만 또는 홀수 위치만 합하면 최적 4를 놓칩니다.

```
nums=[3,1,4,2] → 7
```

322 · 풀이의 출발점과 실행 추적

LeetCode 322: Coin Change · Medium

amount를 만드는 최소 동전 수를 반환하고 불가능하면 -1입니다.

금액 a를 만드는 마지막 동전이 c라고 정하면, 그 앞에는 a-c를 만드는 최선의 방법이 있어야 합니다. 각 동전을 마지막에 놓는 경우를 모두 비교합니다. 큰 동전부터 고르는 규칙은 `[1,3,4]`로 6을 만들 때 4+1+1을 골라 3+3을 놓칩니다.

`coins=[1,3,4], amount=6`. 표의 값은 해당 total의 모든 coin을 검사한 뒤입니다.

total	후보 dp[total-coin]+1	확정 dp[total]
0	동전 없이 가능	0
1,2	0+1 / 1+1	1,2
3	2+1, 0+1	1
4	1+1, 1+1, 0+1	1
5	1+1, 2+1, 1+1	2
6	2+1, 1+1, 2+1	2

322 · 아이디어를 코드로 옮기기

```
def coin_change(coins, amount):
    dp = [float("inf")] * (amount + 1)
    dp[0] = 0
    for total in range(1, amount + 1):
        for coin in coins:
            if coin <= total:
                dp[total] = min(dp[total], dp[total - coin] + 1)
    return -1 if dp[amount] == float("inf") else dp[amount]
```

dp[0]=0이고 나머지는 아직 불가능하다는 뜻의 inf입니다. 동전 c가 양수이므로 total-c는 total보다 작아 이미 계산된 상태입니다. `+1`은 마지막에 놓은 동전 한 개를 셉니다.

inf에 1을 더해도 inf이므로 불가능한 상태가 거짓 최솟값을 만들지 않습니다. 끝까지 inf이면 -1을 반환합니다. `[2]`로 금액 3은 불가능합니다.

동전 [2], amount=3에서는 dp가 `[0,∞,∞,∞]`에서 시작합니다. total=1은 사용할 동전이 없어 그대로, total=2는 dp[0]+1=1, total=3은 dp[1]+1=∞이므로 최종 `[0,∞,1,∞]`에서 −1을 반환합니다.

322 · 정당성·시간·메모리의 근거

최적해의 마지막 동전 c를 제거하면 a-c의 최적해가 남아야 합니다. 모든 가능한 마지막 동전을 비교하므로 최소를 놓치지 않습니다. 양수 동전이라 필요한 상태가 먼저 계산됩니다.

금액 상태 A개마다 동전 m개를 검사하므로 최대 A·m번의 전이가 있습니다. 각 전이는 리스트 접근·덧셈·비교이므로 O(1), 전체 시간은 O(Am)입니다. 바깥 범위가 동전 수가 아니라 목표 금액의 값 A라는 점에 주의합니다.

dp는 0부터 A까지 A+1칸이고 나머지 변수는 고정 개수라서 보조 공간 O(A)입니다. A를 입력에 쓰는 비트 수는 log A이므로 O(Am)은 그 비트 길이에 대해서는 의사다항 시간입니다. 입력 동전 목록은 보조 공간에 다시 세지 않습니다.

큰 동전 우선은 [1,3,4], 6에서 4+1+1을 골라 실패합니다. [2],3은 -1이고 amount=0은 0입니다.

```
coins=[1,3,4], amount=6 → 2
```

62 · 풀이의 출발점과 실행 추적

LeetCode 62: Unique Paths · Medium

격자에서 오른쪽·아래로만 이동하는 경로 수를 반환합니다.

오른쪽·아래로만 이동할 때 현재 칸에 도착하는 마지막 이동은 위에서 내려오거나 왼쪽에서 오는 경우뿐입니다. 두 경우는 마지막 이동이 달라 서로 겹치지 않으므로 경로 수를 더합니다. 최댓값을 고르는 문제가 아닙니다.

rows=3, cols=3. 첫 행·열은 한 방향으로만 올 수 있어 1입니다.

처리한 칸	계산	현재 dp의 행들
초기화	경계값 1	[1,1,1] / [1,1,1] / [1,1,1]
(1,1)	1+1=2	[1,1,1] / [1,2,1] / [1,1,1]
(1,2)	1+2=3	[1,1,1] / [1,2,3] / [1,1,1]
(2,1)	2+1=3	[1,1,1] / [1,2,3] / [1,3,1]
(2,2)	3+3=6	[1,1,1] / [1,2,3] / [1,3,6]

62 · 아이디어를 코드로 옮기기

```
def unique_paths(rows, cols):
    dp = [[1] * cols for _ in range(rows)]
    for r in range(1, rows):
        for c in range(1, cols):
            dp[r][c] = dp[r-1][c] + dp[r][c-1]
    return dp[-1][-1]
```

행 우선 순서로 채우면 `dp[r-1][c]`와 `dp[r][c-1]`이 모두 계산된 뒤입니다. 아직 갱신하지 않은 내부 칸의 초기 1은 임시 값이며, 그 칸을 사용할 때는 전이가 이미 끝나 있습니다.

행마다 새 리스트를 만드는 comprehension을 사용합니다. `[[1]*cols]*rows`는 같은 행을 공유하므로 한 칸 수정이 여러 행에 나타납니다.

62 · 정당성·시간·메모리의 근거

각 경로의 마지막 이동은 아래 또는 오른쪽 하나입니다. 두 집합은 겹치지 않고 전체 경로를 덮으므로 수를 더합니다. 행 순서로 필요한 선행 상태가 준비됩니다.

초기화에 R·C칸, 전이에 (R-1)(C-1)칸을 처리합니다. 각 칸에서 두 값 조회와 덧셈 한 번을 수행하므로 시간 O(RC)입니다. 경로를 직접 하나씩 나열하지 않아 답의 개수만큼 반복하지 않습니다.

전체 dp 표를 보관하므로 보조 공간 O(RC)입니다. 이 구현의 공간을 한 행만 저장하는 최적화의 O(C)와 혼동하면 안 됩니다. 공식 입력의 답 범위에서는 정수 연산을 상수 비용으로 보며, 제한 없는 큰 격자에서는 큰 정수의 비트 비용도 증가합니다.

`[[1]*C]*R`은 같은 행을 공유합니다. `[[1]*C for _ in range(R)]`로 만듭니다.

```
rows=2, cols=4 → 4
```

1143 · 풀이의 출발점과 실행 추적

LeetCode 1143: Longest Common Subsequence · Medium

두 문자열에서 순서를 보존하는 공통 부분수열의 최대 길이를 반환합니다.

두 문자열의 앞부분끼리 비교하는 답을 표에 저장합니다. 마지막 문자가 같으면 두 문자를 함께 사용하여 더 짧은 두 접두부의 답에 1을 더합니다. 다르면 둘을 서로 짝지을 수 없으므로 한쪽 끝을 제외하는 두 경우 중 큰 값을 택합니다.

`first="abc", second="ac"`. 행은 첫 문자열 접두부, 열은 두 번째 접두부입니다.

행 계산 완료	빈 접두부	a	ac
빈 접두부	0	0	0
a	0	1 (=0+1)	1 (=max(0,1))
ab	0	1 (=max(1,0))	1 (=max(1,1))
abc	0	1 (=max(1,0))	2 (=1+1)

답 2는 ac의 길이입니다. b를 건너뛰므로 연속 부분문자열과 다릅니다.

1143 · 아이디어를 코드로 옮기기

```
def lcs(first, second):
    rows, cols = len(first), len(second)
    dp = [[0] * (cols + 1) for _ in range(rows + 1)]
    for r in range(1, rows + 1):
        for c in range(1, cols + 1):
            if first[r-1] == second[c-1]:
                dp[r][c] = dp[r-1][c-1] + 1
            else:
                dp[r][c] = max(dp[r-1][c], dp[r][c-1])
    return dp[rows][cols]
```

dp[r][c]는 길이 r, c인 접두부의 답입니다. 마지막 문자의 인덱스는 각각 r-1, c-1입니다. 0행·0열은 한쪽 문자열이 비어 있어 답 0입니다.

같으면 왼쪽 위 대각선+1, 다르면 위·왼쪽의 max를 사용합니다. 두 선택은 같은 부분수열을 포함할 수 있으므로 더하지 않습니다. 이 코드는 길이만 반환합니다.

1143 · 정당성·시간·메모리의 근거

끝 문자가 같으면 두 끝을 맞춘 최적 부분수열을 택할 수 있습니다. 다르면 둘을 동시에 마지막으로 맞출 수 없어 적어도 한쪽을 생략해야 합니다. 두 경우의 최댓값을 취합니다.

길이가 n, m인 문자열에는 (n+1)(m+1)개 접두부 쌍이 있습니다. 각 내부 칸은 문자 비교와 최대 한 번의 덧셈 또는 max로 계산하므로 O(nm) 시간입니다. n과 m이 같은 크기일 때만 이를 $O(n^2)$로 쓸 수 있습니다.

표 전체가 (n+1)(m+1)칸이므로 보조 공간 O(nm)입니다. 문자열 길이를 반환하는 한 정수는 O(1) 출력입니다. 실제 공통 부분수열을 복원하려면 표를 역추적하고 별도 출력 문자를 저장해야 합니다.

부분수열은 문자의 순서를 유지하면서 중간 문자를 건너뛸 수 있습니다. “abc”에서 b를 건너뛰면 “ac”가 되므로 공통 부분수열의 길이는 2입니다.

```
first="abca", second="aca" → 3
```

끝 문자가 모두 c라면 어떤 최적 부분수열도 마지막 c들을 짝지은 형태로 바꿀 수 있습니다. c를 사용하지 않는 더 짧은 양쪽 접두부의 해에는 c를 붙일 수 있고, 한쪽의 더 이른 c를 사용했다면 그 짝을 마지막 c로 옮겨도 순서가 유지됩니다. 그래서 일치할 때는 대각선 답+1만으로 충분합니다.

중간 점검의 답과 근거

1. 4. 첫 값과 마지막 값을 선택합니다. 홀수·짝수 위치의 합만 비교하면 놓칩니다.
2. 2. 3+3이며 큰 동전부터 고른 4+1+1보다 적습니다.
3. 1과 4. 두 번째 행은 왼쪽부터 1,2,3,4입니다.
4. 2. 부분수열은 순서를 보존하며 중간 문자를 건너뛸 수 있습니다.

개념 질문 해설

`dp[a-c]+1`의 1은 마지막으로 추가한 동전입니다. 금액이 0이면 동전 0개이고, 만들 수 없는 금액은 무한대로 유지합니다.

경로 수는 마지막 이동이 위·왼쪽인 두 집합의 크기를 더합니다. LCS는 가능한 길이 중 최댓값을 구하므로 같은 합 연산을 사용할 수 없습니다.

실습과 참고자료

LeetCode · 198, 322, 62, 1143 · 풀이 · Python ZIP

```
python demo_13.py
python demo_14.py
python verify.py
```

학생 파일을 구현한 뒤 `python verify.py --student`로 검사합니다.

Erik Demaine, Srini Devadas. MIT 6.006, Introduction to Algorithms (2011).