Week 7 Worked Companion

Worked solutions

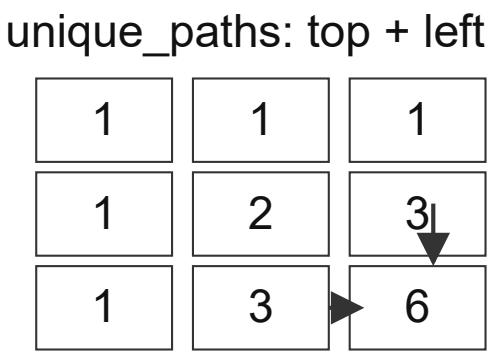
198	322	62	1143

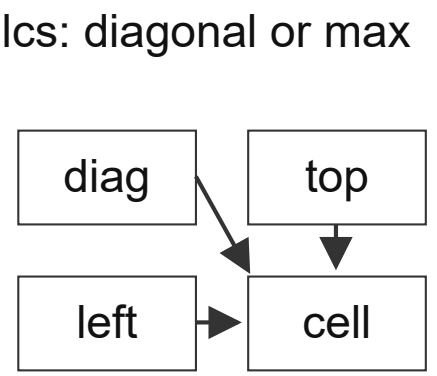
What information is needed to maximize inspection value when adjacent items cannot both be selected? Grid paths and common order in two event logs can also be divided into smaller problems.

Store and reuse previously computed optima or counts. The meaning of the state determines whether to add, minimize, or maximize.

DP state meaning and evaluation order

Dynamic programming stores and reuses answers to repeated subproblems. Define what one table entry means before choosing base cases and transitions.





Problem	Meaning of a state	Combination
Maximum nonadjacent sum	Best value for a prefix	max(select,skip)
Coins	Minimum count for an amount	min over final coins
Grid paths	Number of paths reaching a cell	top+left
Common subsequence	Best length for two prefixes	diagonal+1 on match; otherwise max

Time counts states times work per state. Space counts states retained simultaneously. The nonadjacent-sum problem needs only two prior states, but a table implementation is not automatically constant-space.

198 · Idea and execution trace

LeetCode 198: House Robber · Medium

Maximize selected value without selecting adjacent positions.

Selecting the current value excludes its immediate predecessor. The best prefix value is therefore the larger of skipping the current value or adding it to the best prefix ending two positions earlier. Only those two prior results are needed for the next decision.

`nums=[3,1,4,2]`. Evaluate the entire right-hand side using old values before assignment.

value	Old previous_two	Old previous_one	New 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 · Turning the idea into code

```
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
```

The initial zeros represent selecting nothing. Simultaneous assignment evaluates both right-hand expressions using the old state, so an early overwrite cannot destroy a needed predecessor.

Selecting or skipping the current position covers every possible solution. On `[2,1,1,2]`, the optimum 4 uses both ends and is missed by comparing only even-position and odd-position sums.

A recurrence table and its two variables

Let D[i] be the maximum sum from the first i values. For [3,1,4,2], the empty prefix has D[0]=0.

i and current value	Skip current	Take current	D[i]
1, value 3	0	0+3	3
2, value 1	D[1]=3	D[0]+1=1	3
3, value 4	D[2]=3	D[1]+4=7	7
4, value 2	D[3]=7	D[2]+2=5	7

Before processing 4, previous_two=D[1] and previous_one=D[2]. After the update they become D[2] and D[3], preserving their roles for the next iteration. Taking the current value with D[i−1] could also take its neighbor, violating the constraint.

The branch taking 4 adds it to the already computed prefix optimum 3. Reusing that optimum rather than solving the prefix again is the DP step.

198 · Why it works; time and memory

Every optimum either selects the current item or skips it. Each case leaves an optimal prefix, so the recurrence covers all optima.

For each of n values, perform one addition, one max comparison, and a fixed number of assignments. With constant-cost arithmetic for the input range, time is O(n). Prior prefix optima are already computed rather than searched again.

Only two optima and the current value are retained, independent of n, giving O(1) auxiliary space. A full prefix table would use O(n); this implementation retains only the two dependencies. Reconstructing selected positions requires additional information.

Summing only even or only odd positions misses optimum 4 for [2,1,1,2].

```
nums=[3,1,4,2] → 7
```

322 · Idea and execution trace

LeetCode 322: Coin Change · Medium

Return the minimum coin count for amount, or -1 if impossible.

If the last coin for amount a is c, the preceding coins must form an optimum for a-c. Compare every possible final coin. Choosing the largest coin first fails for amount 6 with `[1,3,4]`: 4+1+1 uses more coins than 3+3.

`coins=[1,3,4], amount=6`. Values are finalized after checking every coin for that total.

total	Candidates dp[total-coin]+1	Final dp[total]
0	No coins needed	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 · Turning the idea into code

```
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]
```

Set dp[0]=0 and all other entries to inf, meaning unreachable so far. Positive c makes total-c smaller than total and therefore already computed. The `+1` counts the chosen last coin.

Adding one to inf remains inf, so unreachable states cannot produce a false minimum. Return -1 if the target stays infinite; amount 3 with coins `[2]` is unreachable.

With coins [2] and amount=3, dp starts as `[0,∞,∞,∞]`. total=1 has no usable coin; total=2 gets dp[0]+1=1; total=3 gets dp[1]+1=∞. The final `[0,∞,1,∞]` therefore returns −1.

322 · Why it works; time and memory

Removing the last coin c from an optimum leaves an optimum for a-c. Comparing all final coins finds the minimum; positive values ensure dependencies were filled earlier.

Each of A amount states checks m coins, giving at most A·m transitions. Each uses constant-time indexing, addition, and comparison, so time is O(Am). The outer bound is the numeric target A, not the number of coins.

The table has A+1 entries and other variables are constant-size, giving O(A) auxiliary space. Since A takes only O(log A) input bits, this is pseudopolynomial time in that bit length. The input coin list is not counted again as workspace.

Largest-first chooses 4+1+1 instead of 3+3 for [1,3,4],6. [2],3 gives -1; amount zero gives zero.

```
coins=[1,3,4], amount=6 → 2
```

62 · Idea and execution trace

LeetCode 62: Unique Paths · Medium

Count grid paths using only right and down moves.

With only right and down moves, the last move into a cell comes either from above or from the left. These cases have different last moves and are disjoint, so add their path counts rather than taking a maximum.

rows=3, cols=3. The first row and column have one possible route.

Cell processed	Calculation	Current dp rows
Initialize	Boundary values 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 · Turning the idea into code

```
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]
```

Row-major order computes both `dp[r-1][c]` and `dp[r][c-1]` before they are used. Initial ones in interior cells are placeholders that are overwritten before serving as dependencies.

Create each row independently with a comprehension. `[[1]*cols]*rows` aliases one row, so one cell update can appear in multiple rows.

62 · Why it works; time and memory

Every path arrives from top or left. These disjoint classes cover all paths, so counts add; row order respects dependencies.

Initialization touches R·C cells; transitions fill (R-1)(C-1) cells. Two lookups and one addition per cell give O(RC) time. Paths are counted rather than enumerated, so runtime is not proportional to the number of paths.

Keeping the whole dp table uses O(RC) auxiliary space. This implementation is not the one-row O(C) optimization. Arithmetic is treated as constant within the official bounded-answer contract; unrestricted large grids would add large-integer bit costs.

`[[1]*C]*R` aliases rows; use a comprehension creating each row.

```
rows=2, cols=4 → 4
```

1143 · Idea and execution trace

LeetCode 1143: Longest Common Subsequence · Medium

Return the length of a longest common subsequence preserving order.

Store answers for pairs of prefixes. If their final characters match, pair them and add one to the shorter-prefix answer. Otherwise they cannot be paired with each other, so take the better result from omitting either final character.

`first="abc", second="ac"`. Rows and columns are prefixes of the two strings.

Completed row	Empty prefix	a	ac
Empty prefix	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)

The answer 2 is the length of `ac`. Skipping b distinguishes a subsequence from a contiguous substring.

1143 · Turning the idea into code

```
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] refers to prefixes of lengths r and c; their final-character indices are r-1 and c-1. Row and column zero represent an empty string, whose common-subsequence length is zero.

On a match use diagonal+1; otherwise use max(top,left). The two alternatives can include the same subsequence, so do not add them. This code returns length only.

1143 · Why it works; time and memory

Equal final characters can be paired in an optimum. Unequal ends cannot both be the final match, so at least one is skipped; maximize over both cases.

For lengths n and m, there are (n+1)(m+1) prefix pairs. Each interior cell uses one character comparison and an addition or max, giving O(nm) time. It becomes O(n^2) only when the two lengths have the same scale.

The full table uses O(nm) auxiliary space, while the returned length occupies O(1). Reconstructing an actual common subsequence would require traceback and output-character storage.

A subsequence preserves character order while allowing characters to be skipped. Skipping b in “abc” gives “ac”, so their longest common subsequence has length 2.

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

When both final characters are c, an optimal subsequence can be chosen to match those final positions. A subsequence using neither final position can be extended by c; a match using an earlier c can move to the final c without changing order. This is why the matching case needs only the diagonal optimum plus one.

Checkpoint answers and reasoning

1. 4, selecting the first and last values. Comparing only odd and even positions misses this choice.
2. 2, using 3+3 rather than the greedy 4+1+1.
3. 1 and 4. The second row contains 1,2,3,4.
4. `2`. A subsequence preserves order and may skip intermediate characters.

Concept-check explanations

The 1 in `dp[a-c]+1` counts the final added coin. Amount zero needs zero coins; unreachable amounts remain infinite.

Path counting adds the sizes of two disjoint sets distinguished by their final move. LCS maximizes possible lengths, so it cannot use the same addition rule.

Exercises and sources

LeetCode · 198, 322, 62, 1143 · Worked solutions · Python ZIP

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

After implementing the student file, check it with `python verify.py --student`.

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