Week 8 Worked Companion

Worked solutions

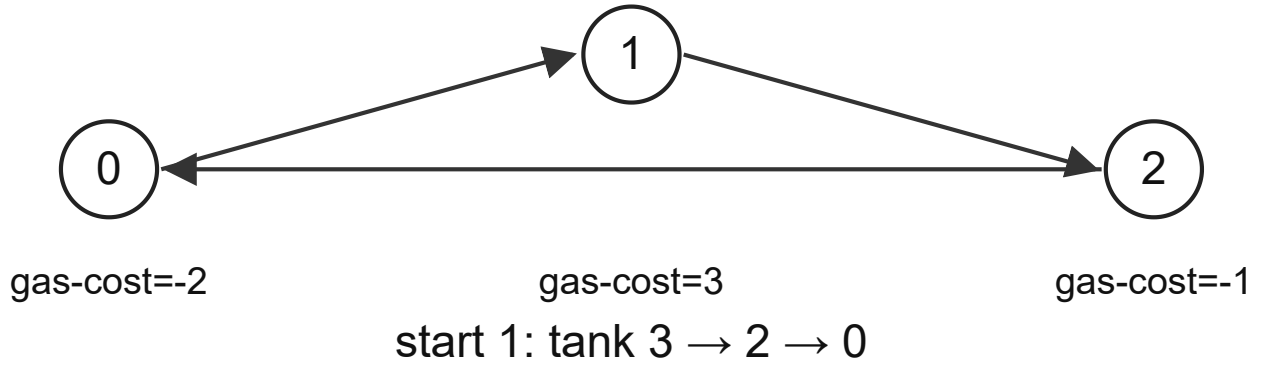
55	45	134	200

When each position allows a different travel distance, determine reachability and the fewest moves. On a circular route, find a starting point that never runs out of fuel.

Solving these tasks without storing every choice requires a reason that discarded candidates cannot be needed. Reachable ranges and prefix sums provide that reason.

Why discarded candidates stay unnecessary

A greedy algorithm eliminates alternatives before storing all choices. It needs a reason why a discarded choice cannot later improve the answer. Merely choosing a large value does not establish that reason.



Problem	Summary retained	Reason for elimination
Reachability	Interval 0..farthest	Earlier positions cover a continuous prefix
Minimum jumps	Current end, next farthest	Inspect every position with fewer jumps first
Fuel circuit	Candidate start and tank	Every start inside a failed segment also fails

The pictured total balance is zero. Start 0 fails immediately, whereas start 1 completes the circuit with balances 3,2,0. Testing total fuel and selecting a starting point serve different roles.

55 · Idea and execution trace

LeetCode 55: Jump Game · Medium

Decide whether maximum forward jump lengths permit reaching the last index.

Each value is a maximum jump length, not a mandatory jump. Reachable intervals therefore form a continuous prefix starting at zero. Keep only its farthest endpoint to decide whether the current index can be visited.

`nums=[2,0,1,0]`. Test reachability using the old farthest before updating it.

index	step	Old farthest	New farthest
0	2	0	max(0,0+2)=2
1	0	2	max(2,1+0)=2
2	1	2	max(2,2+1)=3
3	0	3	max(3,3+0)=3

Index 3 is reachable, so return True.

55 · Turning the idea into code

```
def can_jump(nums):
    farthest = 0
    for index, step in enumerate(nums):
        if index > farthest:
            return False
        farthest = max(farthest, index + step)
    return True
```

Extending the range using an unreachable position would invent a nonexistent path. Test `index > farthest` before updating the range.

For `[1,0,2]`, farthest remains 1 when index 2 is reached, so return False. A large step at an unreachable cell cannot help.

55 · Why it works; time and memory

Reachable positions form the interval 0..farthest. Allowed jumps include every shorter distance, preserving an interval; an index beyond it cannot be crossed.

Read n elements once, using only comparison, addition, and max at each step. Paths are not enumerated, so time is O(n); early failure may shorten a run.

Instead of storing every reachable position in a set, retain one endpoint of a continuous interval. A fixed number of variables gives O(1) auxiliary space. Allowing all smaller jump lengths is what makes this interval summary valid.

In [2,3,0,0,0], taking the longest first jump stops at the zero at index 2. Exploring reachable index 1 also exposes a path to the last index. The algorithm maintains the range reachable from all eligible positions.

```
nums=[2,0,1,0] → True
```

45 · Idea and execution trace

LeetCode 45: Jump Game II · Medium

Return the minimum jumps needed to reach the last index.

Treat all positions reachable with one more jump as a layer. Inspect the whole current layer before fixing the next farthest endpoint. The first layer covering the destination gives the minimum jump count; no single longest jump is chosen in advance.

`nums=[2,3,0,1,0]`. end marks the layer boundary for the current jump count.

index	Updated farthest	Old end	Updated jumps,end
0	2	0	1,2
1	4	2	1,2
2	4	2	2,4
3	4	4	2,4

Route 0 → 1 → 4 needs two jumps; committing to 0 → 2 gets stuck.

45 · Turning the idea into code

```
def min_jumps(nums):
    end = farthest = jumps = 0
    for index in range(len(nums) - 1):
        farthest = max(farthest, index + nums[index])
        if index == end:
            jumps += 1
            end = farthest
    return jumps
```

First include the current position's reach in farthest. When `index == end`, the current layer is fully inspected: increase jumps and move end to farthest.

Use `range(len(nums)-1)` because no jump is needed from the destination. The problem guarantees reachability; without that guarantee, check can_jump first.

45 · Why it works; time and memory

Scan every reachable position in the current level before fixing the next frontier. These are BFS levels in an implicit unweighted graph, so the jump count is minimal.

Explicitly constructing all possible jump edges could create many edges per position. This implementation summarizes each continuous layer with endpoints and scans only n-1 indices, taking O(n) time. Expanding farthest never restarts earlier scans.

Only end, farthest, jumps, and loop variables are stored, giving O(1) auxiliary space. It uses BFS layers conceptually without building a queue. Length one executes zero iterations and returns zero jumps.

The target is n-1. Processing the last index overcounts; [0] gives zero. Unreachable inputs are outside this contract.

`nums=[2,3,0,1,0] → 2`

In the example, layer 0 is {0}, the positions newly reached with one jump are {1,2}, and the next layer is {3,4}. Even after index=1 discovers endpoint 4, the scan finishes the current layer through index=2 before committing jumps=2. end is the last index to examine in a layer, not a chosen landing position.

134 · Idea and execution trace

LeetCode 134: Gas Station · Medium

Return a start completing a circular route from an empty tank, or -1.

When a segment first runs out of fuel, every intermediate starting point also fails by that endpoint. The original start accumulated a nonnegative balance before reaching each intermediate point; starting there with no balance cannot improve the failed suffix. Discard the whole failed segment at once.

`gas=[1,4,2], cost=[3,1,3]`. total is the global balance; tank is the current candidate's balance.

index	gas-cost	total	tank after failure check	start
0	-2	-2	0 (reset)	1
1	+3	1	3	1
2	-1	0	2	1

Starting at 1, balances along 1 → 2 → 0 are 3 → 2 → 0.

134 · Turning the idea into code

```
def gas_start(gas, cost):
    total = tank = start = 0
    for index, (supply, use) in enumerate(zip(gas, cost)):
        total += supply - use
        tank += supply - use
        if tank < 0:
            start, tank = index + 1, 0
    return start if total >= 0 else -1
```

Do not reset total on failure: it measures whether total supply covers total consumption. Reset only tank and move start past the failure point.

A negative total makes every start impossible. Otherwise the final candidate follows a minimum prefix balance and completes the circuit. Tank capacity is unlimited and gas/cost lengths agree.

134 · Why it works; time and memory

At the first failure over s..i, each earlier prefix s..j-1 is nonnegative. Removing such a prefix leaves j..i negative too, eliminating all intermediate starts. If total is nonnegative, the final candidate also has enough surplus to wrap through the eliminated prefixes.

Trying n steps independently from each start would cost O(n^2). Eliminating every start in a failed segment lets index advance without restarting, so there are n constant-cost updates: O(n) time.

zip reads one pair at a time without constructing a difference array. A fixed number of counters gives O(1) auxiliary space. Check both the size of the returned index and the absence of growing intermediate storage.

Choosing the largest gas supply is unjustified: gas=[5,0,4],cost=[6,1,2] fails at station 0; station 2 is the unique valid start.

```
gas=[1,4,2], cost=[3,1,3] → 1
```

Removing a failed segment and wrapping around

For differences [2,−1,−3,3], starting at 0 gives fuel 2→1→−2. The first failure at 2 eliminates starts 0,1,2. Start 1 would also fail with −1−3=−4; start 2 fails with −3.

Prefix sum P[k] is the sum of the first k differences. Here P=[0,2,1,−2,1] has its minimum at 3, which is also the algorithm’s new start.

Station visited from start 3	Calculation	Fuel
3	3	3
0	3+2	5
1	3+2−1	4
2	3+2−1−3	1

In general, fuel after a minimum-prefix position s is P[k]−P[s]≥0. After wrapping, fuel is the full sum P[n] plus P[k]−P[s]. If the full sum is nonnegative, this is also nonnegative, proving that the final candidate completes the wrapped portion too.

200 · Idea and execution trace

LeetCode 200: Number of Islands · Medium

Count land components connected horizontally or vertically in a character grid.

Starting from one land cell and following all four-directional connections reaches its whole island. Start a traversal and increase the count only at unvisited land, preventing duplicate counting. Diagonals do not connect cells.

Grid: `[["1","1","0"],["0","1","0"],["1","0","1"]]`. Coordinates start at zero; stack top is on the right.

Action	stack	Newly added to seen	count
Start (0,0)	[(0,0)]	(0,0)	1
Pop (0,0)	[(0,1)]	(0,1)	1
Pop (0,1)	[(1,1)]	(1,1)	1
Pop (1,1)	[]	None	1
Traverse (2,0), (2,2) separately	[]	(2,0), (2,2)	3

200 · Turning the idea into code

```
def islands(grid):
    rows, cols = len(grid), len(grid[0])
    seen, count = set(), 0
    for row in range(rows):
        for col in range(cols):
            if grid[row][col] != "1" or (row, col) in seen:
                continue
            count += 1
            stack = [(row, col)]
            seen.add((row, col))
            while stack:
                r, c = stack.pop()
                for nr, nc in ((r-1,c),(r+1,c),(r,c-1),(r,c+1)):
                    if (0 <= nr < rows and 0 <= nc < cols
                            and grid[nr][nc] == "1" and (nr,nc) not in seen):
                        seen.add((nr,nc))
                        stack.append((nr,nc))
    return count
```

The outer loops locate traversal starts; the inner while finishes one connected island. Mark a cell in seen just before pushing it, preventing another neighbor from pushing it again.

Cells contain strings “1” and “0”. Check coordinate bounds before reading a cell. The input grid remains unchanged.

200 · Why it works; time and memory

A traversal reaches exactly its starting component. Marked components cannot restart; the number of starts equals the component count.

The outer scan reads R·C cells. Each land cell is pushed and popped once because of seen, and each pop checks four neighbors. Four is independent of input size, giving expected O(RC) time under expected hash-set costs. The inner traversal is not restarted fully for every cell.

Both seen and the pending stack can hold up to RC coordinates, so auxiliary space is O(RC). Marking the grid itself could replace seen, but would mutate the input and would not eliminate stack storage.

Use == for values; is tests identity. Diagonal land in a 2x2 grid forms two components.

```
grid=[["1","0","1"],["1","0","0"]] → 2
```

Checkpoint answers and reasoning

1. No. The reachable boundary remains 2.
2. 2, following 0→1→4.
3. Zero-based index 1. The tank after each leg is 3,2,0.
4. `0`. The start is already the last position, so no jump is counted.

Concept-check explanations

A successful example does not prove a greedy rule. Every position in the current reachable range must be considered before fixing the next level's boundary.

Why the final start completes the circuit

Let $d[i]=gas[i]-cost[i]$, with prefix sums $P[0]=0$ and $P[k]=d[0]+\dots+d[k-1]$. Resetting the candidate when tank becomes negative records a new smaller prefix sum. The final start s is a position of a minimum prefix sum.

After s, accumulated fuel is $P[k]-P[s]$, which is nonnegative. After wrapping to an earlier k, it is $P[n]-P[s]+P[k]$. If $P[n]\geq 0$, this is also nonnegative because $P[k]\geq P[s]$. Thus a nonnegative total guarantees that the final candidate completes the circuit.

Exercises and sources

LeetCode · 55, 45, 134, 200 · Worked solutions · Python ZIP

```
python demo_15.py
python demo_16.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).

Mixed assessment