Week 6 Worked Companion

Worked solutions

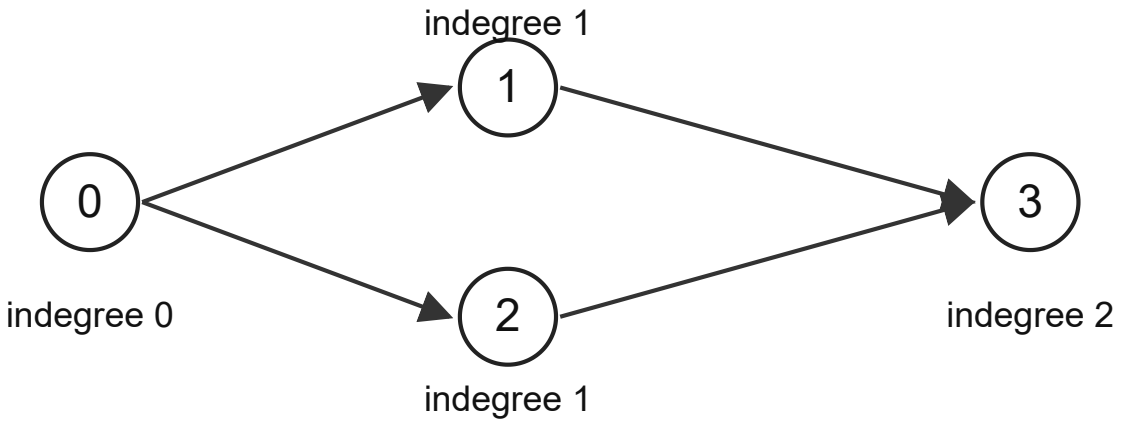
207	210	78	39

A pipeline stops when its jobs wait on each other. Find a feasible execution order, then enumerate the choices that can be made from a given set of items.

Use remaining prerequisites to schedule jobs. Choose, recurse, and restore to keep one search branch from changing the next.

Edge direction and adjacency lists

`[after,before]` means before must finish before after can begin. Store the edge before → after. indegree counts unfinished prerequisites.



Stored item	Value in this graph	Meaning
graph	[[1,2],[3],[3],[]]	Tasks affected when each task finishes
indegree	[0,1,1,2]	Remaining prerequisite counts
Initial queue	[0]	Tasks ready now

The graph stores E edge entries. Unlike a V-by-V matrix, an adjacency list uses storage proportional to V+E.

207 · Idea and execution trace

LeetCode 207: Course Schedule · Medium

Decide whether all courses can be completed under prerequisite constraints.

If no remaining course can start, look for circular waiting. Repeatedly remove zero-prerequisite courses and check whether all courses were removed. Any nonempty remainder contains a directed cycle.

`n=3, prerequisites=[[1,0],[2,1],[1,2]]`: 0 → 1, 1 → 2, 2 → 1

Stage	indegree	queue	Processed
Initial	[0,2,1]	[0]	[]
Complete 0	[0,1,1]	[]	[0]
Stop	1 and 2 wait on each other	[]	1 course < 3

`course_order` returns [] rather than a partial order; `can_finish` returns False.

207 · Turning the idea into code

```
def can_finish(n, prerequisites):
    return len(course_order(n, prerequisites)) == n
```

The one-line call in `can_finish` executes the entire `course_order` traversal below. A function call written on one line does not imply one unit of work.

On a cycle, every course retains an unfinished prerequisite. Every nonempty finite acyclic directed graph has a zero-indegree vertex, so removal can continue until nothing remains.

The called routine processes V vertices and E edges at most once, taking O(V+E) time. Only len and the final comparison are O(1); the earlier traversal still counts.

Adjacency storage costs O(V+E), while indegrees, queue, and order cost O(V). The returned bool occupies O(1), but peak auxiliary space during execution is O(V+E). Output size alone does not determine workspace.

210 · Idea and execution trace

LeetCode 210: Course Schedule II · Medium

Return any prerequisite-respecting order, or an empty list if impossible.

Instead of rechecking every prerequisite repeatedly, store each task's remaining prerequisite count. A count of zero means it can run now. Completing a task decrements only the counts of tasks that depend on it.

`n=4, prerequisites=[[1,0],[2,0],[3,1],[3,2]]`. Edges point before → after.

Completed node	indegree (0,1,2,3)	queue	order
Initial	[0,1,1,2]	[0]	[]
0	[0,0,0,2]	[1,2]	[0]
1	[0,0,0,1]	[2]	[0,1]
2	[0,0,0,0]	[3]	[0,1,2]
3	[0,0,0,0]	[]	[0,1,2,3]

210 · Turning the idea into code

```
from collections import deque

def course_order(n, prerequisites):
    graph, indegree = [[] for _ in range(n)], [0] * n
    for after, before in prerequisites:
        graph[before].append(after)
        indegree[after] += 1
    queue = deque(v for v in range(n) if indegree[v] == 0)
    order = []
    while queue:
        node = queue.popleft()
        order.append(node)
        for neighbor in graph[node]:
            indegree[neighbor] -= 1
            if indegree[neighbor] == 0:
                queue.append(neighbor)
    return order if len(order) == n else []
```

For `[after,before]`, append after to `graph[before]`. When removing a node, decrement its neighbors' indegrees and enqueue a neighbor exactly when its count reaches zero.

Problem 207 asks whether all tasks can be included; problem 210 returns the order itself. Choosing any currently available task preserves prerequisite validity.

210 · Why it works; time and memory

A course enters only after all prerequisites were removed, so every before precedes after in the result. A short result cannot be a complete order.

Initialization stores V vertices and E edges in O(V+E) work. Each vertex enters the queue once; each edge is read once when its source is processed. Degrees vary, but their sum is E, giving total O(V+E) time.

There are V adjacency lists with E entries, plus up to V entries in indegree, queue, and order. Total extra space is O(V+E), including O(V) output order. Although 207 returns only a bool, this implementation calls the same order routine and uses O(V+E) space.

Reading [[1,0]] as 1→0 reverses the dependency. Validate edges instead of comparing with one chosen valid order.

```
n=4, prerequisites=[[2,0],[2,1],[3,2]] → [0,1,2,3]
```

`[[] for _ in range(n)]` creates a separate neighbor list per vertex. `[[]]*n` shares one list, so `graph[0].append(1)` would add 1 to every vertex's neighbors. An indegree reaches zero only once, which prevents duplicate queue insertion on valid inputs.

78 · Idea and execution trace

LeetCode 78: Subsets · Medium

Return every subset of an array of distinct values.

Restrict each next selection to a later index. This generates [2,5] once rather than also generating [5,2]. Every current path is itself a subset, so record it starting with the empty path.

`nums=[2,5]`. A snapshot is stored immediately on entering visit.

Execution	path	Snapshot stored
visit(0)	[]	[]
append(2), visit(1)	[2]	[2]
append(5), visit(2)	[2,5]	[2,5]
pop(), pop()	[]	None
append(5), visit(2)	[5]	[5]

Output order: `[[],[2],[2,5],[5]]`.

78 · Turning the idea into code

```
def subsets(nums):
    result, path = [], []

    def visit(start):
        result.append(path.copy())
        for index in range(start, len(nums)):
            path.append(nums[index])
            visit(index + 1)
            path.pop()

    visit(0)
    return result
```

Append one choice, then explore from `index+1`. On return, pop that choice so another branch starts from the same parent path.

`path.copy()` stores an independent snapshot. Appending path itself would make outputs refer to one mutable list and change together after later pops. Input elements are distinct.

78 · Why it works; time and memory

Every subset has one increasing index sequence. start permits exactly that sequence, and restoration returns the parent path to its prior state.

Each element can be included or excluded, giving 2^n subsets. For n>=1, each element occurs in half of them, so snapshots copy $n \cdot 2^{n-1}$ elements in total. Materializing the output takes $O(n \cdot 2^n)$ time and output space.

The current path and recursion depth are each at most n, so auxiliary space excluding output is O(n). Time counts all work; space counts paths and calls alive at the same moment, not every call ever made.

result.append(path) stores one shared object; later pops can turn multiple results into empty lists.

```
nums=[2,5] → [[],[2],[2,5],[5]]
```

39 · Idea and execution trace

LeetCode 39: Combination Sum · Medium

Return unique combinations reaching target with unlimited candidate reuse.

Choose positive candidates to decrease the remaining sum. Reuse is allowed, so recurse from the selected index again. Never return to earlier indices; this avoids permutation duplicates such as [2,3] and [3,2].

`candidates=[2,3], target=5`. remaining is the sum still required.

path	remaining	Next action
[]	5	Choose 2
[2]	3	Reuse 2
[2,2]	1	2 > 1 → return
[2,3]	0	Save snapshot; return
[3]	2	3 > 2 → stop

Output: `[[2,3]]`.

Returning to the parent’s next choice

For candidates [2,3] and target 5, each call owns its start and remaining variables; only path is shared. Returning from a child resumes the parent’s loop at the next statement.

Execution point	Shared path	This call’s remaining
Parent visit(0,3), append 2	[2,2]	3
Child visit(0,1), stop because 2>1	[2,2]	1
Resume parent, path.pop()	[2]	still 3
Parent’s next index=1, append 3	[2,3]	3
Child visit(1,0), save a copy and return	[2,3]	0
Resume parent, path.pop()	[2]	3

The parent in the first row has already selected the first 2. pop undoes the last shared path choice. The parent’s remaining was never changed, so it needs no addition to restore it. An implementation that mutates a global remaining would have to restore that value too.

39 · Turning the idea into code

```
def combinations(candidates, target):
    values, result, path = sorted(candidates), [], []

    def visit(start, remaining):
        if remaining == 0:
            result.append(path.copy())
            return
        for index in range(start, len(values)):
            value = values[index]
            if value > remaining:
                break
            path.append(value)
            visit(index, remaining - value)
            path.pop()

    visit(0, target)
    return result
```

After sorting, if `value > remaining`, every later candidate is also too large. This permits break rather than continue. Save a snapshot only when remaining becomes zero.

`visit(index, ...)` allows reuse; the subsets routine uses `index+1` to forbid it. Positivity ensures remaining decreases and recursion terminates.

39 · Why it works; time and memory

Positive choices decrease remaining and ensure termination. Nondecreasing indices avoid permutations while representing every combination.

With m candidates, target T, and smallest candidate c, path length is at most L=floor(T/c). Let B be visited calls and S the total output elements. Sorting costs O(m log m), each call checks at most m choices, and copying costs O(S): total O(m log m+mB+S). B depends strongly on the input; this is not a linear scan.

A conservative bound is B=O($(m+1)^{L+1}$). The sorted candidates plus the path and call stack use O(m+L) auxiliary space. Output uses O(S); the full search tree is not stored simultaneously.

Recursing at index+1 misses [2,2] for [2], target=4. Zero is outside the contract and breaks termination.

```
candidates=[2,5], target=7 → [[2,5]]
```

For m=2,T=5,c=2, depth is at most L=2. The loose bound of at most m children per call gives at most $1+m+m^2=7$ calls. Ordering and sum pruning reduce the actual count; copying the S output elements is counted separately.

Checkpoint answers and reasoning

1. None exists: each course waits for the other, so return `[]`.
2. Yes. Both 0 and 1 only need to precede 2.
3. Four subsets, including `[]`. Copy the path so later pops do not alter saved results.
4. Reuse index 0. Advancing to the next index misses `[2,2]`.

Concept-check explanations

DFS state 1 is on the active path, so an edge back to it identifies a cycle. State 2 has left that path. In `0→1`, `0→2`, `2→1`, revisiting completed node 1 is not a cycle.

A recursive return ends only that call. The parent resumes with pop and the next loop choice. Use Kahn's algorithm for long graphs to avoid recursion-depth limits.

Exercises and sources

LeetCode · 207, 210, 78, 39 · Worked solutions · Python ZIP

```
python demo_11.py
python demo_12.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).