Topological Sort

207. Course Schedule 1
210. Course Schedule 2
1462. Course Schedule 4
444. Sequence Reconstruction
269. Alien Dict
329. Longest Increasing Path in a Matrix
1203. Sort Items by Groups Respecting Dependencies

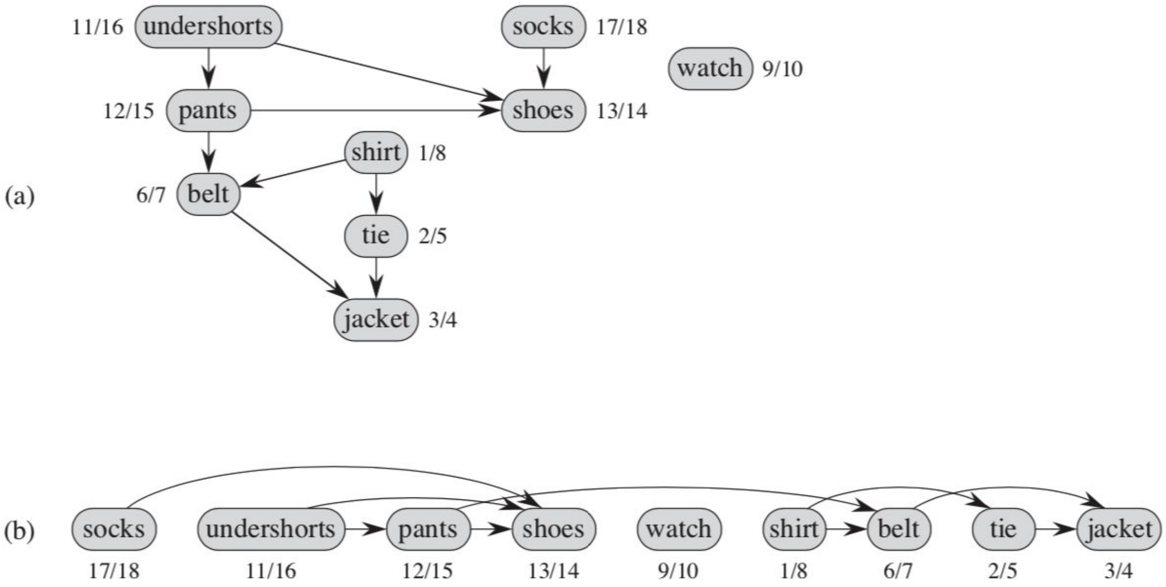
Yifan Fei
Jan 06th 2021

Definition

A *topological sort* of a dag G =(V, E) is a linear ordering of all its vertices such that if G contains an edge (u,v) then u appears before v in the ordering. (If the graph contains a cycle, then no linear ordering is possible.)

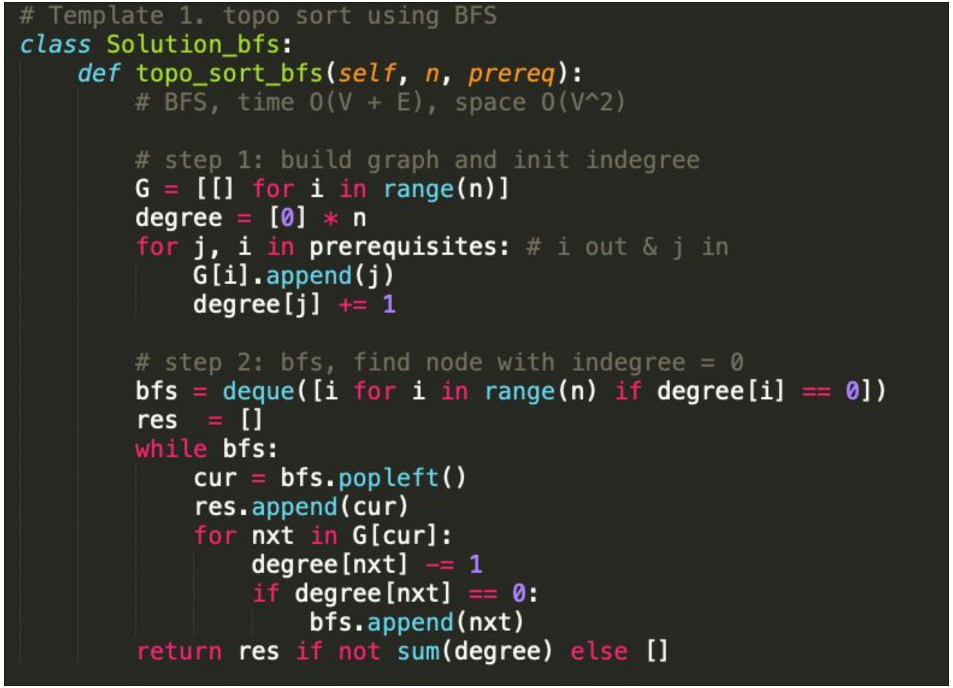
We can view a topological sort of a graph as an ordering of its vertices along a horizontal line so that all directed edges go from left to right. Topological sorting is thus different from the usual kind of “sorting” studied in Part II.

Example



We can view a topological sort of a graph as an ordering of its vertices along a horizontal line so that all directed edges go from left to right.

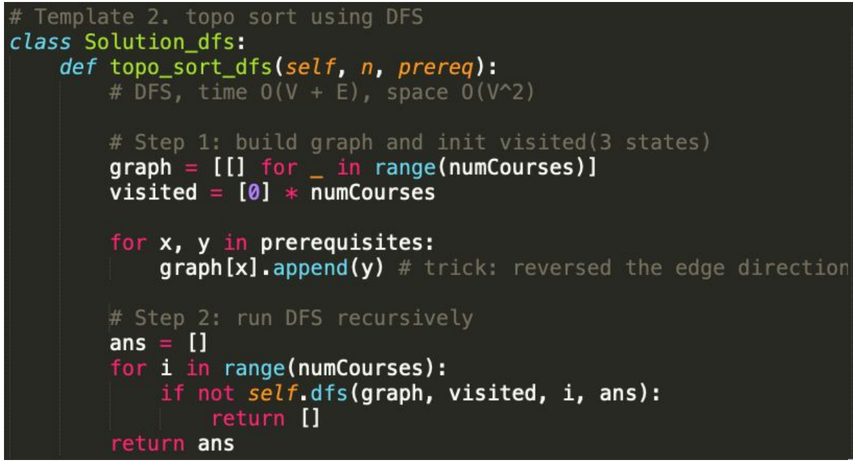
Template-BFS



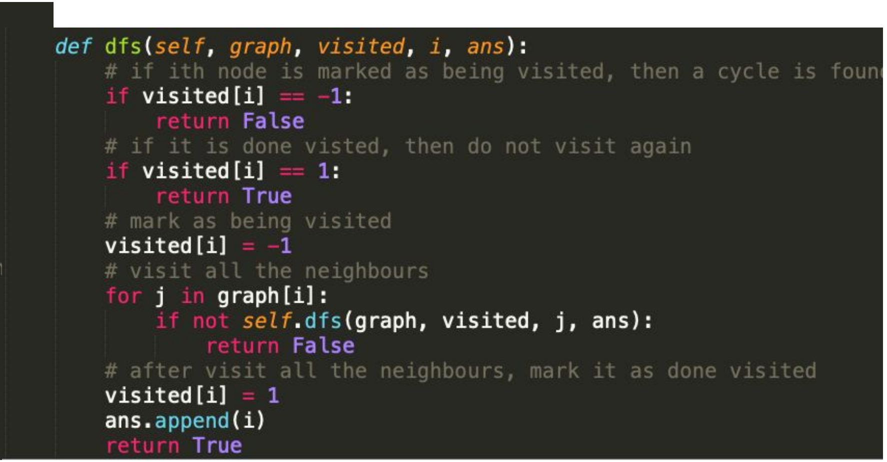
```python
# Template 1. topo sort using BFS
class Solution_bfs:
    def topo_sort_bfs(self, n, prereq):
        # BFS, time O(V + E), space O(V^2)

        # step 1: build graph and init indegree
        G = [[] for i in range(n)]
        degree = [0] * n
        for j, i in prerequisites: # i out & j in
            G[i].append(j)
            degree[j] += 1

        # step 2: bfs, find node with indegree = 0
        bfs = deque([i for i in range(n) if degree[i] == 0])
        res  = []
        while bfs:
            cur = bfs.popleft()
            res.append(cur)
            for nxt in G[cur]:
                degree[nxt] -= 1
                if degree[nxt] == 0:
                    bfs.append(nxt)
        return res if not sum(degree) else []
```

BFS based on indegree[]

Template-DFS



```
# Template 2. topo sort using DFS
class Solution_dfs:
    def topo_sort_dfs(self, n, prereq):
        # DFS, time O(V + E), space O(V^2)

        # Step 1: build graph and init visited(3 states)
        graph = [[] for _ in range(numCourses)]
        visited = [0] * numCourses

        for x, y in prerequisites:
            graph[x].append(y) # trick: reversed the edge direction

        # Step 2: run DFS recursively
        ans = []
        for i in range(numCourses):
            if not self.dfs(graph, visited, i, ans):
                return []
        return ans
```



```
def dfs(self, graph, visited, i, ans):
    # if ith node is marked as being visited, then a cycle is foun
    if visited[i] == -1:
        return False
    # if it is done visted, then do not visit again
    if visited[i] == 1:
        return True
    # mark as being visited
    visited[i] = -1
    # visit all the neighbours
    for j in graph[i]:
        if not self.dfs(graph, visited, j, ans):
            return False
    # after visit all the neighbours, mark it as done visited
    visited[i] = 1
    ans.append(i)
    return True
```

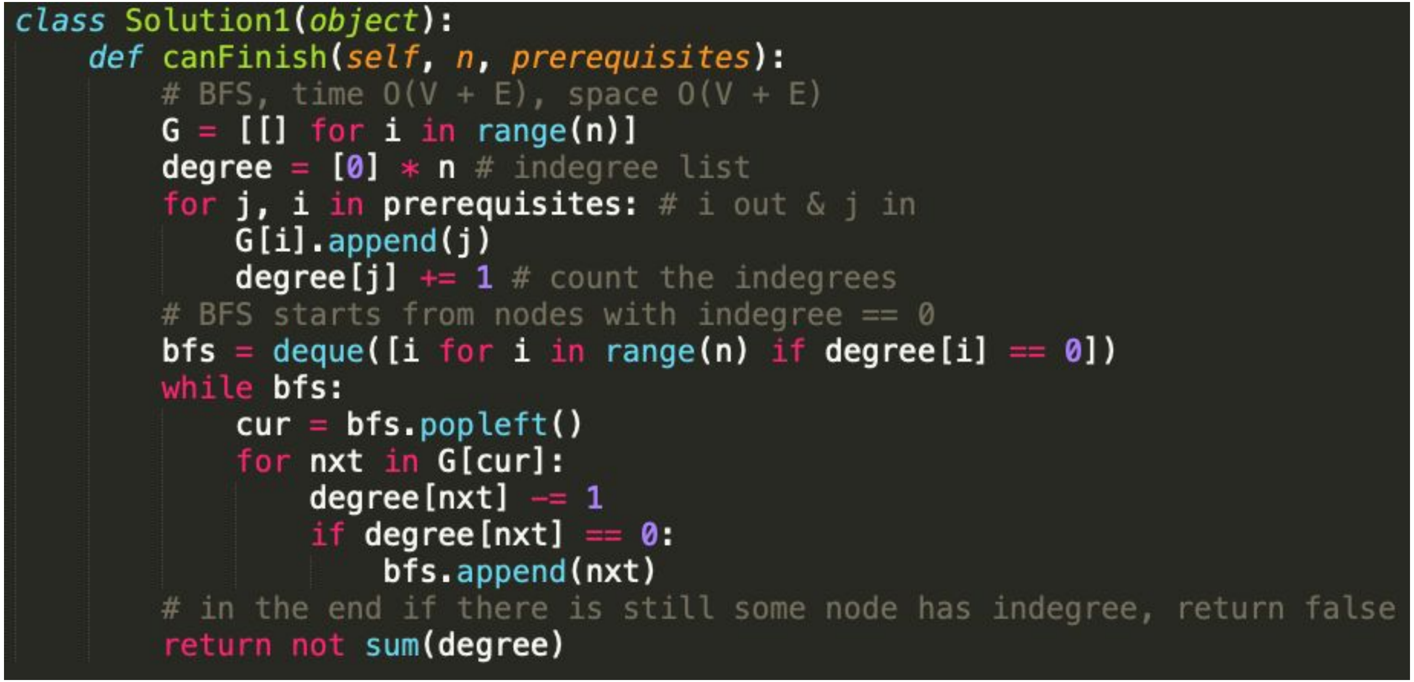
DFS based on cycle detection & recursion(stack)

Note the edge direction should be reversed / or the result should be reversed

207. Course Schedule 1

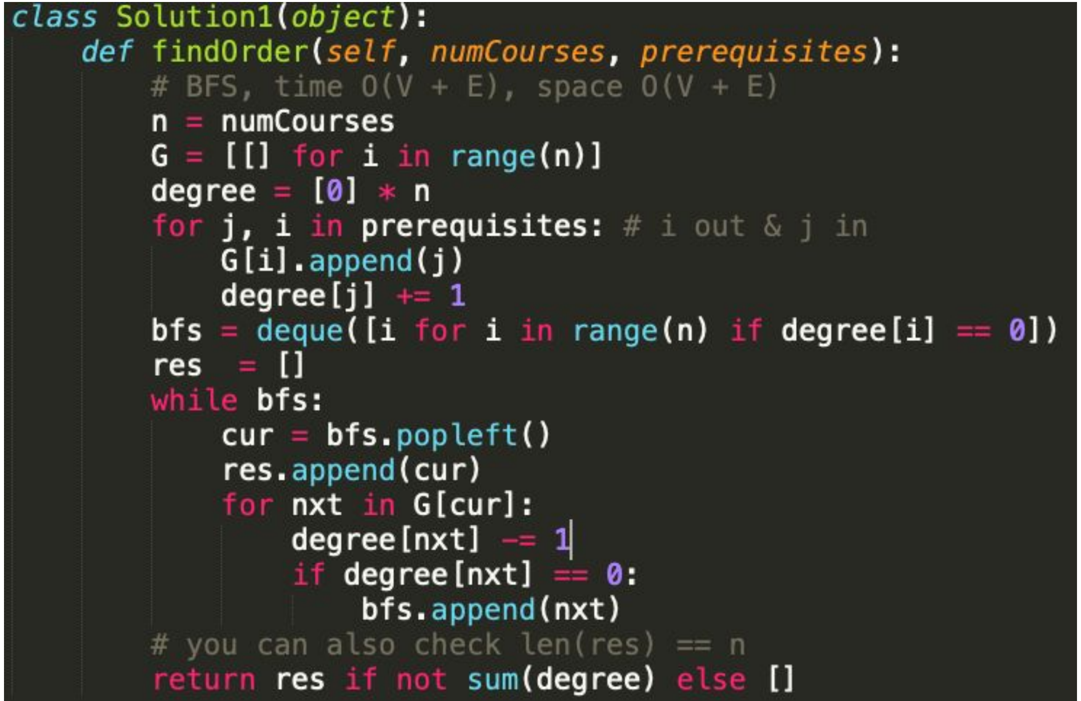
```
Input: numCourses = 2, prerequisites =
[[1,0],[0,1]]
Output: false
```

- `1 <= numCourses <= 10^5`



```python
class Solution1(object):
    def canFinish(self, n, prerequisites):
        # BFS, time O(V + E), space O(V + E)
        G = [[] for i in range(n)]
        degree = [0] * n # indegree list
        for j, i in prerequisites: # i out & j in
            G[i].append(j)
            degree[j] += 1 # count the indegrees
        # BFS starts from nodes with indegree == 0
        bfs = deque([i for i in range(n) if degree[i] == 0])
        while bfs:
            cur = bfs.popleft()
            for nxt in G[cur]:
                degree[nxt] -= 1
                if degree[nxt] == 0:
                    bfs.append(nxt)
        # in the end if there is still some node has indegree, return false
        return not sum(degree)
```

210. Course Schedule 2



```
class Solution1(object):
    def findOrder(self, numCourses, prerequisites):
        # BFS, time O(V + E), space O(V + E)
        n = numCourses
        G = [[] for i in range(n)]
        degree = [0] * n
        for j, i in prerequisites: # i out & j in
            G[i].append(j)
            degree[j] += 1
        bfs = deque([i for i in range(n) if degree[i] == 0])
        res  = []
        while bfs:
            cur = bfs.popleft()
            res.append(cur)
            for nxt in G[cur]:
                degree[nxt] -= 1
                if degree[nxt] == 0:
                    bfs.append(nxt)
        # you can also check len(res) == n
        return res if not sum(degree) else []
```

Given the total number of courses numCourses and a list of the prerequisite pairs, return the ordering of courses you should take to finish all courses.

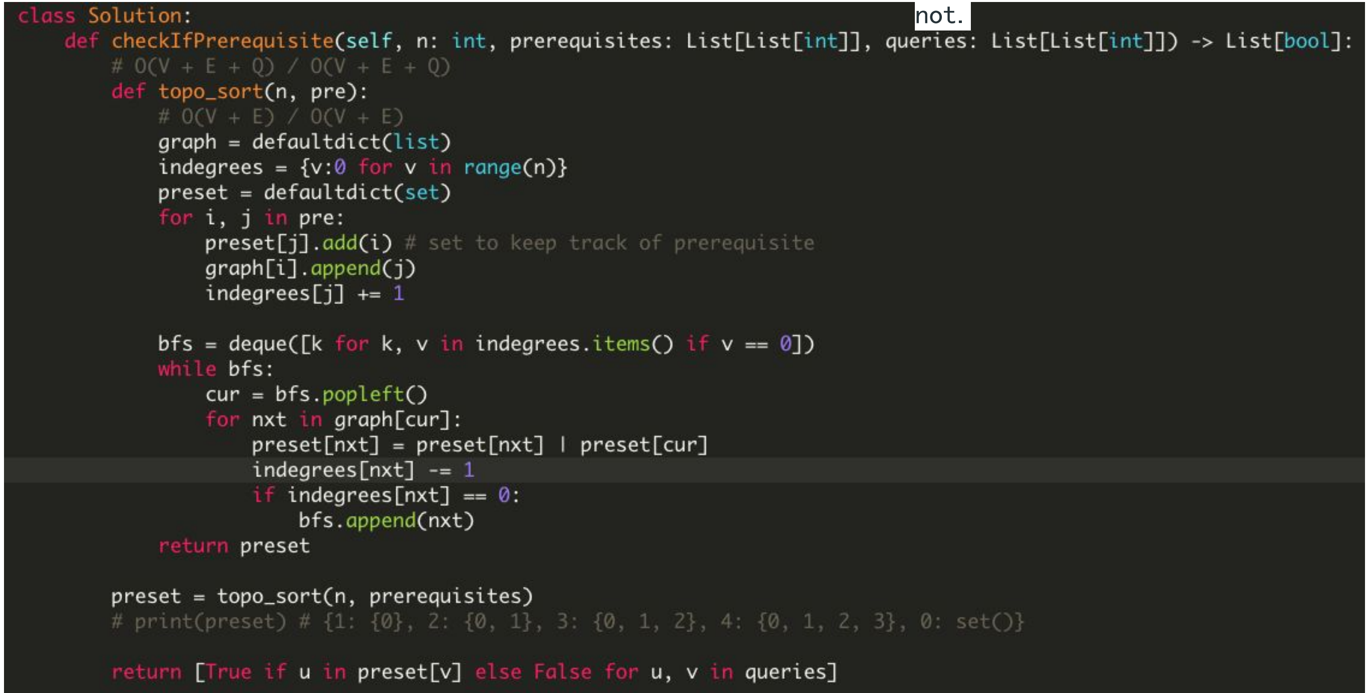
If there are many valid answers, return any of them. If it is impossible to finish all courses, return an empty array.

- 1 <= numCourses <= 2000
- 0 <= prerequisites.length <= numCourses * (numCourses - 1)
- prerequisites[i].length == 2
- 0 <= a_i, b_i < numCourses
- a_i != b_i
- All the pairs [a_i, b_i] are

1462. Course Schedule 4

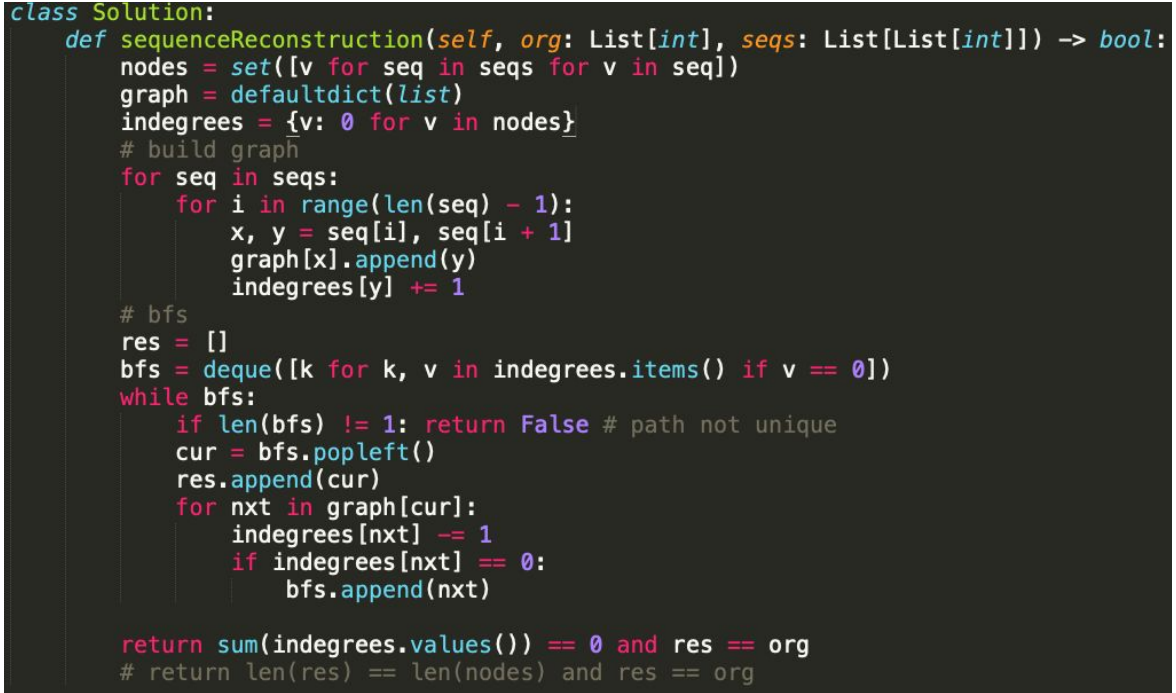
Given the total number of courses `n`, a list of direct `prerequisite` pairs and a list of `queries` pairs.

You should answer for each `queries[i]` whether the course `queries[i][0]` is a prerequisite of the course `queries[i][1]` or not.



```
class Solution:
    def checkIfPrerequisite(self, n: int, prerequisites: List[List[int]], queries: List[List[int]]) -> List[bool]:
        # O(V + E + Q) / O(V + E + Q)
        def topo_sort(n, pre):
            # O(V + E) / O(V + E)
            graph = defaultdict(list)
            indegrees = {v:0 for v in range(n)}
            preset = defaultdict(set)
            for i, j in pre:
                preset[j].add(i) # set to keep track of prerequisite
                graph[i].append(j)
                indegrees[j] += 1

            bfs = deque([k for k, v in indegrees.items() if v == 0])
            while bfs:
                cur = bfs.popleft()
                for nxt in graph[cur]:
                    preset[nxt] = preset[nxt] | preset[cur]
                    indegrees[nxt] -= 1
                    if indegrees[nxt] == 0:
                        bfs.append(nxt)
            return preset

        preset = topo_sort(n, prerequisites)
        # print(preset) # {1: {0}, 2: {0, 1}, 3: {0, 1, 2}, 4: {0, 1, 2, 3}, 0: set()}

        return [True if u in preset[v] else False for u, v in queries]
```

444. Sequence Reconstruction



```python
class Solution:
    def sequenceReconstruction(self, org: List[int], seqs: List[List[int]]) -> bool:
        nodes = set([v for seq in seqs for v in seq])
        graph = defaultdict(list)
        indegrees = {v: 0 for v in nodes}
        # build graph
        for seq in seqs:
            for i in range(len(seq) - 1):
                x, y = seq[i], seq[i + 1]
                graph[x].append(y)
                indegrees[y] += 1
        # bfs
        res = []
        bfs = deque([k for k, v in indegrees.items() if v == 0])
        while bfs:
            if len(bfs) != 1: return False # path not unique
            cur = bfs.popleft()
            res.append(cur)
            for nxt in graph[cur]:
                indegrees[nxt] -= 1
                if indegrees[nxt] == 0:
                    bfs.append(nxt)

        return sum(indegrees.values()) == 0 and res == org
        # return len(res) == len(nodes) and res == org
```

Check whether the original sequence `org` can be uniquely reconstructed from the sequences in `seqs`. The `org` sequence is a permutation of the integers from 1 to n, with 1 ⩽ n ⩽ 104. Reconstruction means building a shortest common supersequence of the sequences in `seqs` (i.e., a shortest sequence so that all sequences in `seqs` are subsequences of it). Determine whether there is **only one sequence** that can be reconstructed from `seqs` and it is the `org` sequence.

```
Input: org = [1,2,3], seqs =
[[1,2],[1,3]]
```

```
Output: false
```

- `1 <= n <= 10^4`
- `org` is a permutation of {1, 2, . . . , n}.
- `1 <= segs[i].length <= 10^5`
- `seqs[i][j]` fits in a 32-bit signed integer

269. Alien Dict

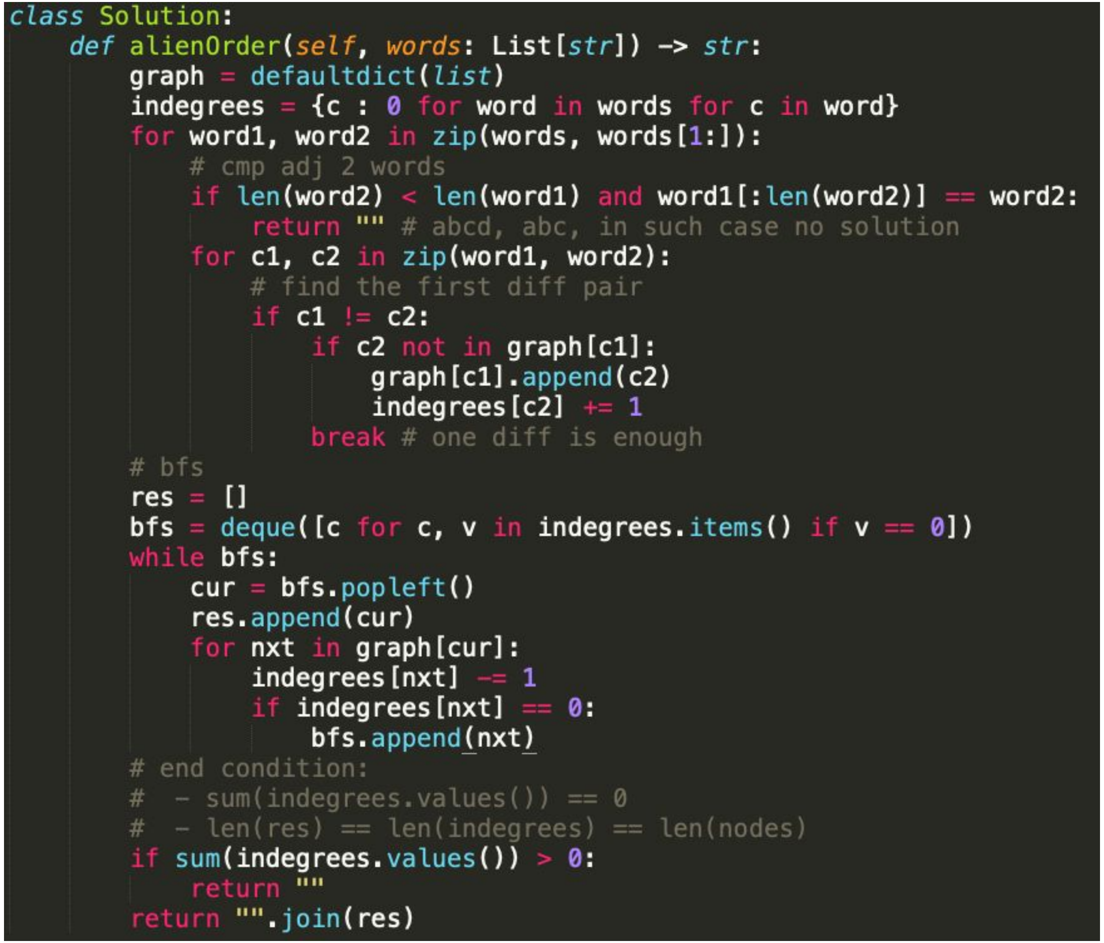
You are given a list of strings `words` from the dictionary, where `words` are sorted lexicographically by the rules of this new language.

Derive the order of letters in this language, and return it. If the given input is invalid, return `""`. If there are multiple valid solutions, return any of them.

```
Input: words =
["wrt","wrf","er","ett","rftt"]
```

```
Output: "wertf"
```

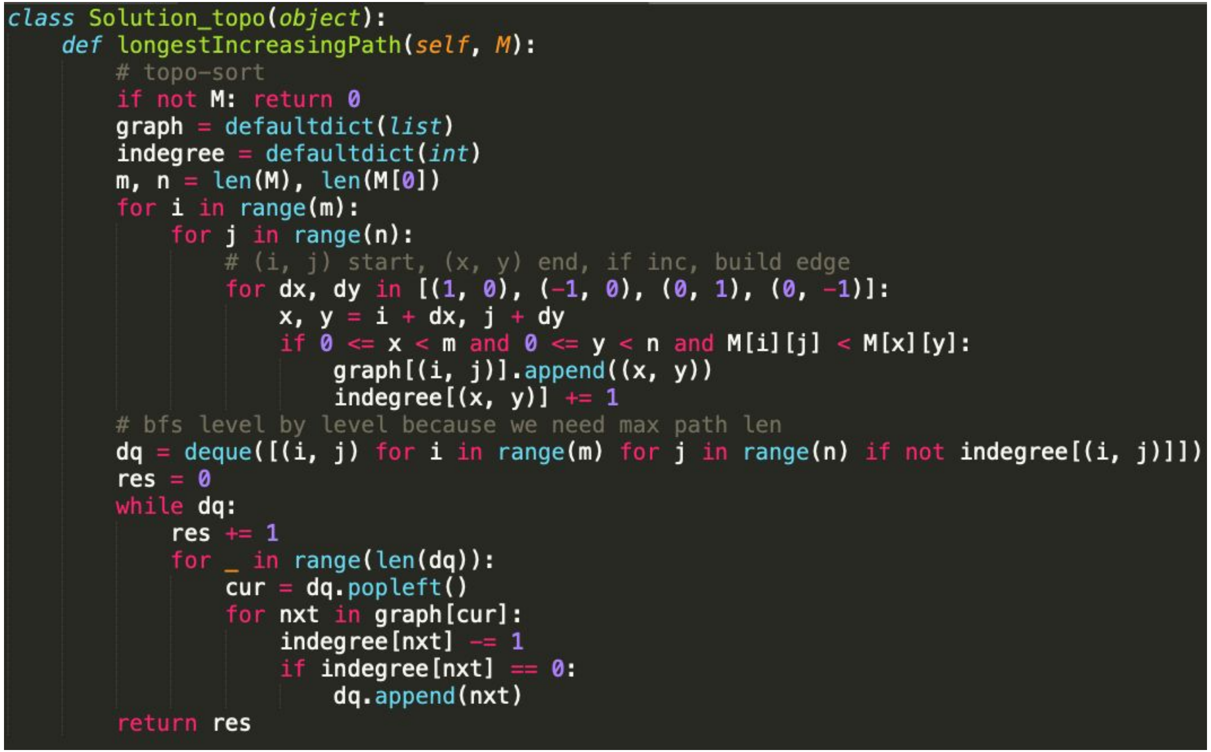
- `1 <= words.length <= 100`
- `1 <= words[i].length <= 100`
- `words[i]` consists of only



```python
class Solution:
    def alienOrder(self, words: List[str]) -> str:
        graph = defaultdict(list)
        indegrees = {c : 0 for word in words for c in word}
        for word1, word2 in zip(words, words[1:]):
            # cmp adj 2 words
            if len(word2) < len(word1) and word1[:len(word2)] == word2:
                return "" # abcd, abc, in such case no solution
            for c1, c2 in zip(word1, word2):
                # find the first diff pair
                if c1 != c2:
                    if c2 not in graph[c1]:
                        graph[c1].append(c2)
                        indegrees[c2] += 1
                    break # one diff is enough
        # bfs
        res = []
        bfs = deque([c for c, v in indegrees.items() if v == 0])
        while bfs:
            cur = bfs.popleft()
            res.append(cur)
            for nxt in graph[cur]:
                indegrees[nxt] -= 1
                if indegrees[nxt] == 0:
                    bfs.append(nxt)
        # end condition:
        #   - sum(indegrees.values()) == 0
        #   - len(res) == len(indegrees) == len(nodes)
        if sum(indegrees.values()) > 0:
            return ""
        return "".join(res)
```

329. Longest Increasing Path in a Matrix

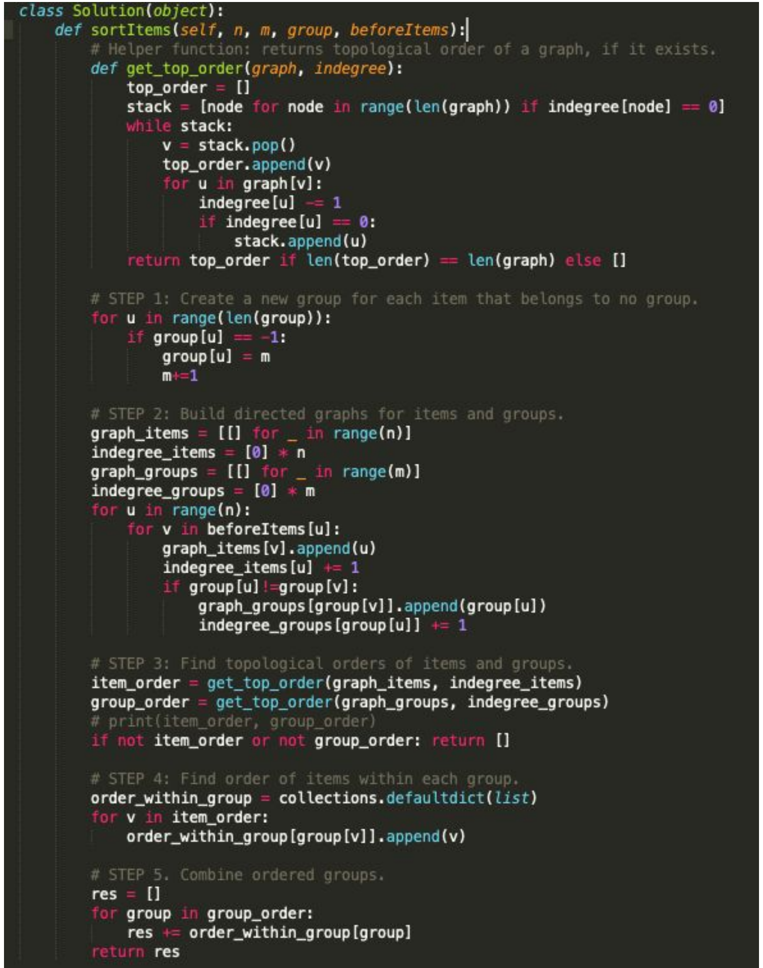
https://leetcode.com/problems/longest-increasing-path-in-a-matrix/



```python
class Solution_topo(object):
    def longestIncreasingPath(self, M):
        # topo-sort
        if not M: return 0
        graph = defaultdict(list)
        indegree = defaultdict(int)
        m, n = len(M), len(M[0])
        for i in range(m):
            for j in range(n):
                # (i, j) start, (x, y) end, if inc, build edge
                for dx, dy in [(1, 0), (-1, 0), (0, 1), (0, -1)]:
                    x, y = i + dx, j + dy
                    if 0 <= x < m and 0 <= y < n and M[i][j] < M[x][y]:
                        graph[(i, j)].append((x, y))
                        indegree[(x, y)] += 1
        # bfs level by level because we need max path len
        dq = deque([(i, j) for i in range(m) for j in range(n) if not indegree[(i, j)]])
        res = 0
        while dq:
            res += 1
            for _ in range(len(dq)):
                cur = dq.popleft()
                for nxt in graph[cur]:
                    indegree[nxt] -= 1
                    if indegree[nxt] == 0:
                        dq.append(nxt)
        return res
```

1203. Sort Items by Groups Respecting Dependencies

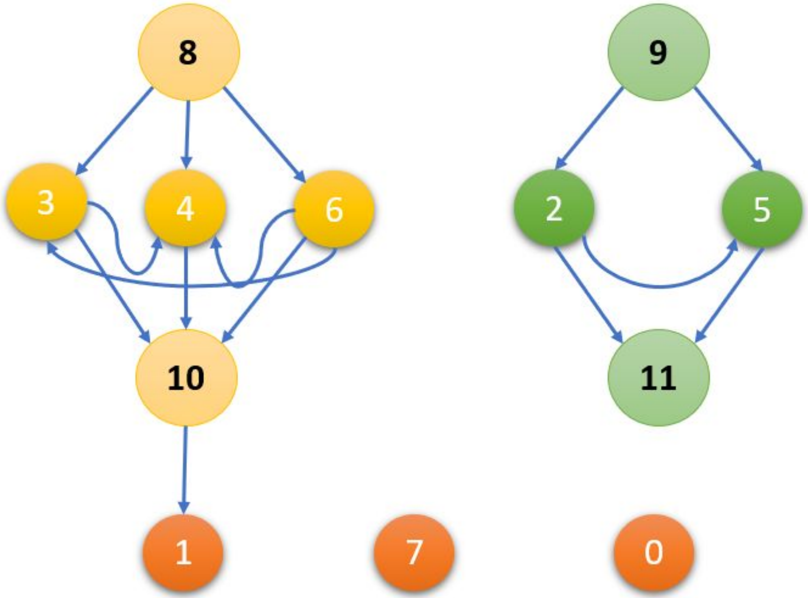
2 level topo-sort



```
class Solution(object):
    def sortItems(self, n, m, group, beforeItems):
        # Helper function: returns topological order of a graph, if it exists.
        def get_top_order(graph, indegree):
            top_order = []
            stack = [node for node in range(len(graph)) if indegree[node] == 0]
            while stack:
                v = stack.pop()
                top_order.append(v)
                for u in graph[v]:
                    indegree[u] -= 1
                    if indegree[u] == 0:
                        stack.append(u)
            return top_order if len(top_order) == len(graph) else []

        # STEP 1: Create a new group for each item that belongs to no group.
        for u in range(len(group)):
            if group[u] == -1:
                group[u] = m
                m+=1

        # STEP 2: Build directed graphs for items and groups.
        graph_items = [[] for _ in range(n)]
        indegree_items = [0] * n
        graph_groups = [[] for _ in range(m)]
        indegree_groups = [0] * m
        for u in range(n):
            for v in beforeItems[u]:
                graph_items[v].append(u)
                indegree_items[u] += 1
                if group[u]!=group[v]:
                    graph_groups[group[v]].append(group[u])
                    indegree_groups[group[u]] += 1

        # STEP 3: Find topological orders of items and groups.
        item_order = get_top_order(graph_items, indegree_items)
        group_order = get_top_order(graph_groups, indegree_groups)
        # print(item_order, group_order)
        if not item_order or not group_order: return []

        # STEP 4: Find order of items within each group.
        order_within_group = collections.defaultdict(list)
        for v in item_order:
            order_within_group[group[v]].append(v)

        # STEP 5. Combine ordered groups.
        res = []
        for group in group_order:
            res += order_within_group[group]
        return res
```

1203. Sort Items by Groups Respecting Dependencies

2 level topo-sort


8
3
4
6
10
1
7
9
2
5
11
0