**NO nOProblem: Optimal Scheduling for Task Completion**

**Scenario**

You are managing a project that involves a series of tasks, each with a specific duration and a list of prerequisite tasks that must be completed before it can start. Your goal is to find the minimum amount of time required to complete all tasks, considering the dependencies.

**Problem Statement**

Given an integer n representing the number of tasks, and a list of tasks where each task is defined by a tuple (task\_duration, [prerequisite\_tasks]), find the minimum amount of time required to complete all tasks. You can assume that tasks with no prerequisites can start immediately and tasks can be executed in parallel if their prerequisites are met.

**Input Format**

* The first line contains an integer n, representing the number of tasks.
* The next n lines each contain an integer task\_duration followed by a space-separated list of prerequisite tasks (0-indexed) for the respective task.

**Constraints**

* 1 <= n <= 1000
* 1 <= task\_duration <= 1000 for each task
* 0 <= number of prerequisites <= n-1

**Output Format**

* Print a single integer representing the minimum amount of time required to complete all tasks.

**Sample Input**

5

3 0 1

2 1

1 0

4 2 3

2 3

**Sample Output**

4

**Explanation**

Tasks can be represented as follows:

* Task 0: duration 3, prerequisites [0, 1]
* Task 1: duration 2, prerequisites [1]
* Task 2: duration 1, prerequisites [0]
* Task 3: duration 4, prerequisites [2, 3]
* Task 4: duration 2, prerequisites [3]

The optimal way to schedule these tasks is:

1. Start Task 0 immediately (takes 3 units of time).
2. Task 1 can start after Task 0 finishes (takes 3+2=5 units of time).
3. Task 2 can start after Task 0 finishes (takes 3+1=4 units of time).
4. Task 3 can start after Task 2 finishes (takes 4+4=8 units of time).
5. Task 4 can start after Task 3 finishes (takes 8+2=10 units of time).

However, since Task 3 and Task 4 can be executed in parallel, the total time required is 4 units.

**Solution**

Here's a solution using dynamic programming and topological sorting:

python

from collections import deque, defaultdict

def minTimeToCompleteTasks(n, tasks):

indegree = [0] \* n

graph = defaultdict(list)

time = [0] \* n

# Build the graph and time arrays

for i in range(n):

task = tasks[i]

duration = task[0]

prerequisites = task[1:]

time[i] = duration

for prereq in prerequisites:

graph[prereq].append(i)

indegree[i] += 1

# Topological sort

queue = deque()

for i in range(n):

if indegree[i] == 0:

queue.append(i)

dp = [0] \* n

while queue:

node = queue.popleft()

for neighbor in graph[node]:

dp[neighbor] = max(dp[neighbor], dp[node] + time[node])

indegree[neighbor] -= 1

if indegree[neighbor] == 0:

queue.append(neighbor)

return max(dp[i] + time[i] for i in range(n))

# Input reading

n = int(input())

tasks = []

for \_ in range(n):

task\_info = list(map(int, input().split()))

tasks.append(task\_info)

# Output

print(minTimeToCompleteTasks(n, tasks))

**Additional Test Cases**

**Test Case 1**

**Input:**

3

5

8 0

3 1

**Output:**

16

**Test Case 2**

**Input:**

4

2

4 0

6 0

1 1 2

**Output:**

9

**Test Case 3**

**Input:**

6

2

3 0

4 0 1

6 1 2

5 3

**Output:**

17

These test cases ensure that the solution correctly computes the minimum time required to complete all tasks in various scenarios, considering the dependencies and allowing for parallel execution where possible