11.8

Muhammad Tanveer

Other

PDF generated at: 30 Jul 2024 02:32:41 UTC

View this report on HackerRank

Score

11.8% • 20 / 170

scored in CodePath TIP103: Unit 7 Assessment - Summer 2024 in 4 min 29 sec on 29 Jul 2024 19:26:23 PDT

Candidate Information

Email	tanveerm176@gmail.com
Test	CodePath TIP103: Unit 7 Assessment - Summer 2024
Candidate Packet	View
Taken on	29 Jul 2024 19:26:23 PDT
Time taken	4 min 29 sec/ 90 min
Work Experience	1 years
Invited by	CodePath

Skill Distribution

No.	Skill	Score
1	Problem Solving Intermediate	0%

Tags Distribution

Graphs	3%	Data Structures	0%
Medium	0%	Algorithms	0%
Problem Solving	0%	Strongly Connected Components	0%
Interviewer Guidelines	0%		

Questions

Status	No.	Question	Time Taken	Skill	Score
✓	1	Apply Topological Sort on a Graph Multiple Choice	43 sec	-	5/5
✓	2	Time Complexity: Topological Sort Multiple Choice	8 sec	-	5/5
✓	3	Valid Topological Sort Multiple Choice	42 sec	-	5/5
✗	4	Minimum Weight Path in a Directed Graph Coding	1 min 29 sec	Problem Solving (Intermediate)	0/75
✗	5	Task Master Coding	15 sec	Problem Solving (Intermediate)	0/75

✓	6	Search Comparison Multiple Choice	1 min 6 sec	-	5/5

1. Apply Topological Sort on a Graph

Correct

Multiple Choice Graphs

Question description

Topological sorts can be applied to which of the following graphs?

Candidate's Solution

Options: (Expected answer indicated with a tick)

- ○ Undirected Cyclic Graphs
- ○ Directed Cyclic Graphs
- ○ Undirected Acyclic Graphs
- ● **Directed Acyclic Graphs** ✓

No comments.

2. Time Complexity: Topological Sort

Correct

Multiple Choice

Question description

What is the most efficient time complexity of topological sorting? Where V represents the number of vertices and E represents the number of edges.

Candidate's Solution

Options: (Expected answer indicated with a tick)

- [x] O(V+E)
- [ ] O(V)
- [ ] O(E)
- [ ] O(V*E)

No comments.

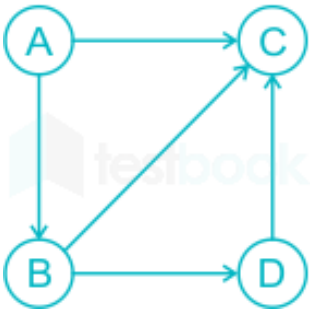
3. Valid Topological Sort

Correct

Multiple Choice

Question description

Which of the following is a valid topological order of graph G? (Select all that apply)



Candidate's Solution

Options: (Expected answer indicated with a tick)

- [ ] ABCD
- [ ] BACD
- [ ] BADC
- [x] ABDC

No comments.

4. Minimum Weight Path in a Directed Graph

Incorrect

Coding | Data Structures | Graphs | Medium | Algorithms | Problem Solving

Question description

Given a directed graph with weighted edges, determine the minimum weighted path from node *1* to the last node.

A directed graph is defined *g* such that:

- The total number of nodes in the graph is *g_nodes*.
- The nodes are numbered sequentially as *1, 2, 3, ..., g_nodes*.
- The total number of edges in the graph is *g_edges*.
- Each edge connects two distinct nodes (i.e., no edge connects a node to itself).
- The edge connecting nodes *g_from[i]* and *g_to[i]* is directed. In other words, it describes a path only in the direction *g_from[i]* → *g_to[i]*.
- The weight of the edge connecting nodes *g_from[i]* and *g_to[i]* is *g_weight[i]*.

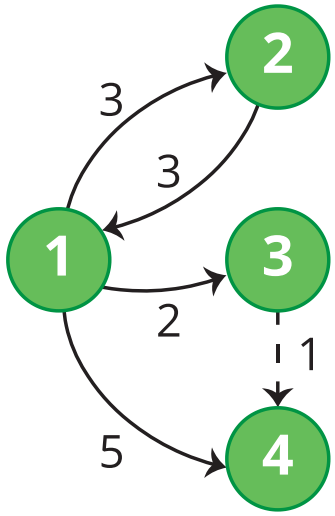
The *weight* of a path from node *1* to node *g_nodes* is the sum of the edges traversed on the path.

Extra directed edges of weight *1* (one) can be added between any two distinct nodes that are not already connected by an edge. Find the path from node *1* to node *g_nodes* with the minimum possible weight.

Example
The following list of edges in a 4-node graph is given:

```
From    To   Weight
1    2    3
1    3    2
2    1    3
1    4    5
```

In the graph shown, an additional edge was added from *3* to *4*. The minimum total cost to get from node *1* to node *4* is *3* with the path $1 \rightarrow_2 3 \rightarrow_1 4$.



Function Description

Complete the function *minCost* in the editor below.

minCost has the following parameter(s):

int g_nodes: the number of nodes in graph *g*
int g_from[g_edges]: edge origin nodes
int g_to[g_edges]: edge target nodes
int g_weight[g_edges]: edge weights

Returns

int: the minimum possible weight of any path from node *1* to node *g_nodes*

Constraints

- $3 \le g\_nodes \le 10^3$
- $1 \le g\_edges \le min(10^4, {}^{(g\_nodes \times (g\_nodes - 1))}/_2)$
- $1 \le g\_weight[i] \le 10^6$

▼ INPUT FORMAT FOR CUSTOM TESTING

Input from stdin will be processed as follows and passed to the function.

The first line contains two space-separated integers, *g_nodes* and *g_edges*.
Each of the next *g_edges* lines contains three space-separated integers, *g_from*, *g_to* and *g_weight*.

▼ SAMPLE CASE 0

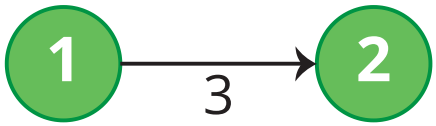
Sample Input

```
STDIN    Function
-----    -----
2 1   → g_nodes = 2, g_edges = 1
1 2 3 → g_from = 1, g_to = 2, g_weight = 3
```

Sample Output

```
3
```

Explanation



A directed edge already exists from node *1* to node *2, and the path 1 → 2 is the minimum-cost* path, so the function returns *3*.

▼ SAMPLE CASE 1

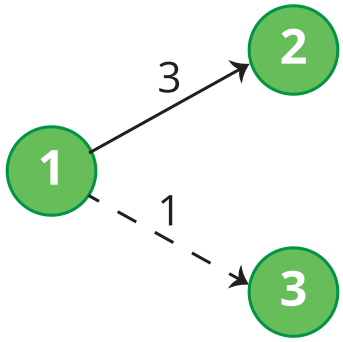
Sample Input

```
STDIN    Function
-----    -----
3 1   → g_nodes = 3, g_edges = 1
1 2 3 → g_from = 1, g_to = 2, g_weight = 3
```

Sample Output

```
1
```

Explanation



As graph *g* has no edge between nodes *1* and *3*, an extra edge can be added from node *1* to node *3* with weight *1*. Thus, the path *1 → 3* is the minimum weight path.

▼ SAMPLE CASE 2

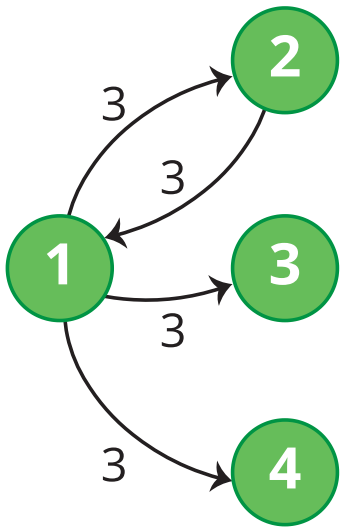
Sample Input

```
STDIN    Function
-----    -----
4 4   → g_nodes = 4, g_edges = 4
1 2 3 → g_from = 1, g_to = 2, g_weight = 3
1 3 3 → g_from = 1, g_to = 3, g_weight = 3
1 4 3 → g_from = 1, g_to = 4, g_weight = 3
2 1 3 → g_from = 2, g_to = 1, g_weight = 3
```

Sample Output

```
3
```

Explanation



A directed edge already exists from node *1* to node *4* and the path *1* → *4* is the minimum-cost path, so the function returns *3*.

Interviewer guidelines

Python 3 Solution

```
def minCost(g_nodes, g_from, g_to, g_weight):
    dic = {}
    for fr, to, w in zip(g_from, g_to, g_weight):
        dic[(fr,to)] = w
    res = 0
    for i in range(1,g_nodes):
        min_wt = 999999999999
        cur = 0
        for j in range(i+1,g_nodes+1):
            if (i,j) in dic:
                if(dic[(i,j)] <= min_wt):
                    cur = j
                    min_wt = dic[(i,j)]
                if cur == g_nodes:
                    return res + min_wt
            elif (i,j) not in dic:
                min_wt = 1
                cur = j
                if cur == g_nodes:
                    return res + 1
        i = cur-1
```

```
        res += min_wt
    return res
```

Candidate's Solution Language used: **Python 3**

```python
#!/bin/python3

import math
import os
import random
import re
import sys



#
# Complete the 'minCost' function below.
#
# The function is expected to return an INTEGER.
# The function accepts WEIGHTED_INTEGER_GRAPH g as parameter.
#

#
# For the weighted graph, <name>:
#
# 1. The number of nodes is <name>_nodes.
# 2. The number of edges is <name>_edges.
# 3. An edge exists between <name>_from[i] and <name>_to[i]. The weight of the edge is <name>_weight[i].
#
#

def minCost(g_nodes, g_from, g_to, g_weight):
    # Write your code here

if __name__ == '__main__':
    fptr = open(os.environ['OUTPUT_PATH'], 'w')

    g_nodes, g_edges = map(int, input().rstrip().split())

    g_from = [0] * g_edges
    g_to = [0] * g_edges
    g_weight = [0] * g_edges
```

```
38
39      for i in range(g_edges):
40          g_from[i], g_to[i], g_weight[i] = map(int, input().rstrip().split())
41
42      result = minCost(g_nodes, g_from, g_to, g_weight)
43
44      fptr.write(str(result) + '\n')
45
46      fptr.close()
47
```

Result: Compilation Failed

Compile Message

Sorry: IndentationError: expected an indented block after function definition on line 27 (Solution.py, line 30)

No comments.

5. Task Master

Incorrect

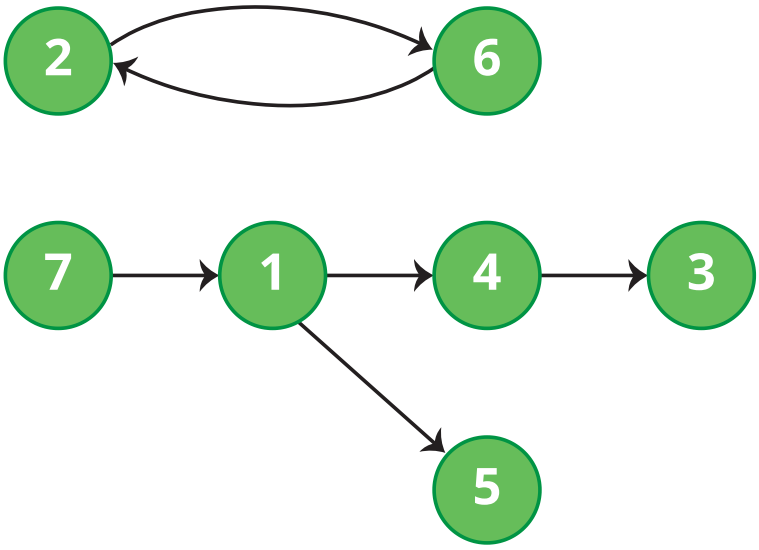
Coding | Graphs | Strongly Connected Components | Medium | Algorithms | Problem Solving | Interviewer Guidelines

Question description

Eric is the most methodical employee at the Acme company. His manager assigned him a number of tasks for the quarter, and gave him a list of notes regarding the order they must be performed. Each note states that some task must be completed before some related task. If he goes to perform some task and sees that a rule exists requiring that this task be performed *before* an already completed task, then he cannot perform the task. Help Eric determine the maximum number of tasks he can complete.

For example, Eric has *n=7* tasks to complete. His manager gives him *m=6* notes on the order tasks must be performed. Here is a graph of the dependencies.The dependent array, *a = [1, 2, 3, 4, 6, 5]*. His principal tasks array, *b = [7, 6, 4, 1, 2, 1]*. Here is a graph of the dependencies.

```
a = [1,2,3,4,6,5]
b = [7,6,4,1,2,1]
```



From the graph, it is easy to see that task 6 must be performed *before* task *2* and vice versa. He can only complete one of those two tasks before the other, so he must choose either task *6* or *2*. He can complete *7 - 1 = 6* tasks.

Function Description

Complete the function *tasks* in the editor below. It must return an integer denoting the maximum number of tasks that Eric can complete.

tasks has the following parameter(s):

n: integer, the number of tasks

a[a[0],...a[n-1]]: an array of integers, the dependent tasks

b[b[0],...b[n-1]]: an array of integers, the primary tasks

Constraints

- $1 \leq n \leq 10^5$
- $0 \leq m \leq n$
- $1 \leq a[i], b[i] \leq n$
- Each task depends on at most one other task.

INPUT FORMAT FOR CUSTOM TESTING

Input from stdin will be processed as follows and passed to the function.

The first line contains an integer *n*, the number of tasks.
The next line contains an integer *m*, the size of the array *a*.
Each of the next *m* lines contains an integer *a[i]* where $0 \leq i < n$.
The next line contains the integer *m*, the size of the array *a*.
Each of the next *m* lines contains an integer *b[i]* where $0 \leq i < n$.

SAMPLE CASE 0

Sample Input 0

```
2
0
0
```

Sample Output 0

```
2
```

Explanation 0

The following arguments are passed to your function:

n = 2
a = {}
b = {}

There are two tasks and no dependencies. Eric can complete *2* tasks.

SAMPLE CASE 1

Sample Input 1

```
2
1
1
```

```
1
2
```

Sample Output 1

```
2
```

Explanation 1

The following arguments are passed to your function:

n = 2
a = {1}
b = {2}

There is only one dependency, so it can be satisfied. Eric can complete task *2* then task *1*.

▼ SAMPLE CASE 2

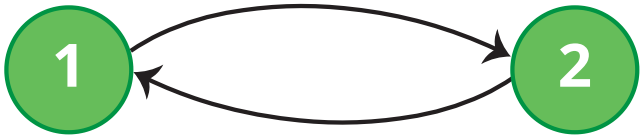
Sample Input 2

```
2
2
1
2
2
2
1
```

Sample Output 2

```
1
```

Explanation 2



There are two tasks and two dependencies. Each of the tasks must be performed before the other. Eric can choose only *1* task to complete.

Interviewer guidelines

▼ HINT 1

Pay close attention to the additional constraint given in the problem: "Each task depends on at most one other task." Draw a few cases in a paper to visualize what's happening.

▼ HINT 2

The graph given is a Directed Cyclic Graph. Think of an algorithm which simplifies the graph; maybe convert it to Directed Acyclic Graph?

▼ SOLUTION

Concepts covered: Strongly Connected Components, Greedy

Optimal Solution:

We will represent the relations in the form of a directed graph. Because of the constraint that each task depends on at most one other task, the out-degree of each node will be 1 at most. Therefore, there will be a unique path you can travel from each node. Since the graph is a Directed Cyclic Graph it is hard to apply any observation here. We will use a neat trick: first convert the graph into Directed Acyclic Graph (DAG) and then the answer will be much simpler to calculate. The algorithm to convert the graph into a DAG can be easily implemented using Kosaraju's Algorithm and the DAG thus obtained is called a Condensation Graph. If you are unfamiliar with the algorithm you can read the brief abstract below:

Let's first define Strong Connected Components: It is a subset of vertices of the original graph such that each vertex is reachable from any other node. Now, we can assume that the original graph is built as a union of many such components which are disjoint. If we represent these components as a single node and add edges between these "component nodes", we obtain a DAG called a Condensation

Graph. Once you have found all the strongly connected components of the graph, the problem is much simpler to solve. The components can be of the following two types:

The size of the component is 1, i.e. the component contains only one node.
The size of the component is greater than 1, i.e. the component contains more than one node. In this case, there is only one type of graph possible where each node in the component is in a cycle. This is because of the first constraint that we stated (each vertex is reachable from every other vertex).

It is always optimal to first complete the tasks of the component present above other components. The tasks in this component can be performed before completing the tasks of the components in it's subtree. And hence, we can say that the final answer would be the sum of the maximum number of tasks we can complete in each component. For the components of size 1, we can always perform it. But for components of a size greater than 1, optimally, we won't be able to complete at least one task. You can prove this by drawing a sample graph.

```
E = []
Er = []
vis = []
order = []
component = []

def dfs1(u):
    vis[u] = True
    for v in E[u]:
        if not vis[v]:
            dfs1(v)
    order.append(u)

def dfs2(u):
    vis[u] = True
    component.append(u)
    for v in Er[u]:
        if not vis[v]:
            dfs2(v)

def tasks(n, a, b):
    E[:] = [[] for _ in range(n)]
    Er[:] = [[] for _ in range(n)]
    vis[:] = [False] * n
    m = len(a)
    for i in range(m):
        u = a[i] - 1
        v = b[i] - 1
```

```
        E[u].append(v)
        Er[v].append(u)
    for i in range(n):
        if not vis[i]:
            dfs1(i)
    vis[:] = [False] * n
    ans = 0
    for i in range(n):
        v = order[n - i - 1]
        if not vis[v]:
            dfs2(v)
            global component
            ans += max(1, len(component)-1)
            component = []
    return ans
```

Brute Force Approach:

A simple brute force approach could be to take all the subsets of tasks and find whether they can be completed under the given constraints. The complexity would be worse here, O(n*2^n).

Error Handling: There are lots of sources of error in the problem, some of which are:

The graph can be disconnected, so we need to find the strongly connected component (SCC) for every connected component of the graph.

The candidate may get confused in coming up with the conclusion that for each connected component the contribution would be max(1, len(component)-1).

▼ COMPLEXITY ANALYSIS

Time Complexity - O(n).

The Kosaraju's Algorithm taken O(n) time, and the final answer could be computed as a subroutine of Kosaraju's Algorithm, hence overal time complexity is O(n).

Space Complexity - O(n)

We need to build the graph/reverse graph and keep track of visited nodes and current component for the implementation of Kosaraju's Algorithm, each would take the space of O(n), hence the overall space complexity is O(n).

Candidate's Solution Language used: **Python 3**

```
1 #!/bin/python3
2
3 import math
4 import os
```

```
import random
import re
import sys



# Complete the tasks function below.
def tasks(n, a, b):

if __name__ == '__main__':
    fptr = open(os.environ['OUTPUT_PATH'], 'w')

    n = int(input().strip())

    a_count = int(input().strip())

    a = []

    for _ in range(a_count):
        a_item = int(input().strip())
        a.append(a_item)

    b_count = int(input().strip())

    b = []

    for _ in range(b_count):
        b_item = int(input().strip())
        b.append(b_item)

    res = tasks(n, a, b)

    fptr.write(str(res) + '\n')

    fptr.close()

```

Result: Compilation Failed

Compile Message

Sorry: IndentationError: expected an indented block after function definition on line 12 (Solution.py, line 14)

No comments.

6. Search Comparison

Correct

Multiple Choice

Question description

When is it better to use a **breadth first search** on a graph versus a depth first search?

Candidate's Solution

Options: (Expected answer indicated with a tick)

- BFS explores vertices in the order of their distance from the source vertex, where distance is the minimum length of a path from the source vertex to the node.
- If our tree is broad, use DFS as BFS will take too much memory.
- If we know the solution lies somewhere deep in a tree or far from the source vertex in the graph, use DFS. If we know the solution is not that far from the source vertex, use BFS. Similarly, if our tree is very deep, choose BFS over DFS.
- **All are true.** ✓

No comments.