Some more graph problems

Counting Islands

- We will look at a problem that has been a common interview question.

- **Given**:
  - grid with 1s and 0s.
  - 1 represents land.
  - 0 represents water.
  - Grid is surrounded by water (in other words, imagine 0s all around the grid)
  - A land mass (aka island ☺) is defined as a bunch of “contiguous” 1s
    - “contiguous” defined as immediate neighbor to the *left* or *right* or *up* or *down.*

- **Problem**:
  - Find the number of islands in the given grid.

- Look at the islands highlighted next.

0	0	1	1	0
1	0	0	1	0
0	0	1	1	0
0	1	0	0	1
0	0	0	0	0
1	1	1	0	1

Counting Islands

- **Given**:
  - grid with 1s and 0s.
  - 1 represents land.
  - 0 represents water.
  - Grid is surrounded by water (in other words, imagine 0s all around the grid)
  - A land mass (aka island ☺) is defined as a bunch of “contiguous” 1s
    - “contiguous” defined as immediate neighbor to the *left* or *right* or *up* or *down.*

- **Problem**:
  - Find the number of islands in the given grid.

0	0	1	1	0
1	0	0	1	0
0	0	1	1	0
0	1	0	0	1
0	0	0	0	0
1	1	1	0	1

Number of islands is 6.
I used colors for ease of differentiation.
Same color doesn’t mean same island ☺

Counting Islands

- How should we proceed to a solution?
- Can we use this as a bunch of nodes in a graph, with each node having neighbors to its left/right/up/down?

0	0	1	1	0
1	0	0	1	0
0	0	1	1	0
0	1	0	0	1
0	0	0	0	0
1	1	1	0	1

Number of islands is 6.
I used colors for ease of differentiation.
Same color doesn't mean same island ☺

Counting Islands

- How should we proceed to a solution?
- Can we use this as a bunch of nodes in a graph, with each node having neighbors to its left/right/up/down?
  - Traversing the graph to determine number of islands:
    - I could start at a cell.
      - If cell has 1, then I could go to its left/right/up/down neighbors as long as they also have a 1.
      - …

0	0	1	1	0
1	0	0	1	0
0	0	1	1	0
0	1	0	0	1
0	0	0	0	0
1	1	1	0	1

Number of islands is 6.
I used colors for ease of differentiation.
Same color doesn't mean same island ☺

Counting Islands

- How should we proceed to a solution?
- Can we use this as a bunch of nodes in a graph, with each node having neighbors to its left/right/up/down?
  - Traversing the graph to determine number of islands:
    - I could start at a cell.
      - If cell has 1, then I could go to its left/right/up/down neighbors as long as they also have a 1.
      - But if a cell has 0, then I will ignore it (and not look at its neighbors).

0	0	1	1	0
1	0	0	1	0
0	0	1	1	0
0	1	0	0	1
0	0	0	0	0
1	1	1	0	1

Number of islands is 6.
I used colors for ease of differentiation.
Same color doesn't mean same island ☺

Counting Islands

numberOfIslands = 0

for each node n_i in the grid
    if (n_i was already visited)
        continue
    if n_i contains 0
        continue

    ++ numberOfIslands

    TraverseStartingAtNode (n_i)

U can think of this function as “marking the island whose “1” we just found.

It will do that by spreading (or traversing) out to neighbors that have a “1”

What r our options for traversing?

0	0	1	1	0
1	0	0	1	0
0	0	1	1	0
0	1	0	0	1
0	0	0	0	0
1	1	1	0	1

Counting Islands

numberOfIslands = 0

for each node n_i in the grid
    if (n_i was already visited)
        continue
    if n_i contains 0
        continue

    ++ numberOfIslands

    ~~TraverseStartingAtNode~~ BFS (n_i)

What r our options for traversing? ➔ BFS

Any other options?

0	0	1	1	0
1	0	0	1	0
0	0	1	1	0
0	1	0	0	1
0	0	0	0	0
1	1	1	0	1

Counting Islands

numberOfIslands = 0

for each node n_i in the grid
    if (n_i was already visited)
        continue
    if n_i contains 0
        continue

    ++ numberOfIslands

    ~~TraverseStartingAtNode~~ DFS (n_i)

What r our options for traversing? ➔ DFS

0	0	1	1	0
1	0	0	1	0
0	0	1	1	0
0	1	0	0	1
0	0	0	0	0
1	1	1	0	1

Counting Islands

- What if we say <u>diagonally</u> connected 1s are part of the same island?
- What modification would need to be done and what would the result look like?

```
numberOfIslands = 0

for each node n_i in the grid
        if ( n_i was already visited )
                continue
        if n_i contains 0
                continue

        ++ numberOfIslands

        DFS ( n_i )
```

0	0	1	1	0
1	0	0	1	0
0	0	1	1	0
0	1	0	0	1
0	0	0	0	0
1	1	1	0	1

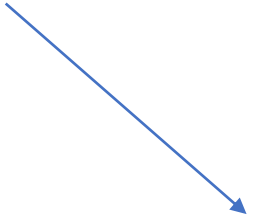
Counting Islands

- What if we say <u>diagonally</u> connected 1s are part of the same island?
- What modification would need to be done and what would the result look like?

```
numberOfIslands = 0

for each node n_i in the grid
        if ( n_i was already visited )
                continue
        if n_i contains 0
                continue

        ++ numberOfIslands

        DFS ( n_i )
```

Diagonally connected 1s r part of same island.

So, now the number of islands is 4.



0	0	1	1	0
1	0	0	1	0
0	0	1	1	0
0	1	0	0	1
0	0	0	0	0
1	1	1	0	1

Counting Islands

numberOfIslands = 0

for each node n_i in the grid
    if (n_i was already visited)
        continue
    if n_i contains 0
        continue

    ++ numberOfIslands

    BFS (n_i)

Using a **different** grid with one big island.

What cell will this traverse to next for **BFS** ?

0	1_1	1_2	1_4	1_7
0	1_3	1_5	1_8	1
0	1_6	1_9	1	1
0	1_{10}	1	1	1
0	1	1	1	1
0	1	1	1	1

Counting Islands

numberOfIslands = 0

for each node n_i in the grid
    if (n_i was already visited)
        continue
    if n_i contains 0
        continue

    ++ numberOfIslands

    DFS (n_i)

Code is in the IDE

What cell will this traverse to next for **DFS** ?

0	1_1	1_2	1_3	1_4
0	1_8	1_7	1_6	1_5
0	1_9	1_{10}	1_{11}	1_{12}
0	1_{16}	1_{15}	1_{14}	1_{13}
0	1	1	1	1
0	1	1	1	1

Topological Sort

Next problem we will look at involves topological sorting of a graph.

We have talked about this before, so lets revisit this quickly before going to the problem.

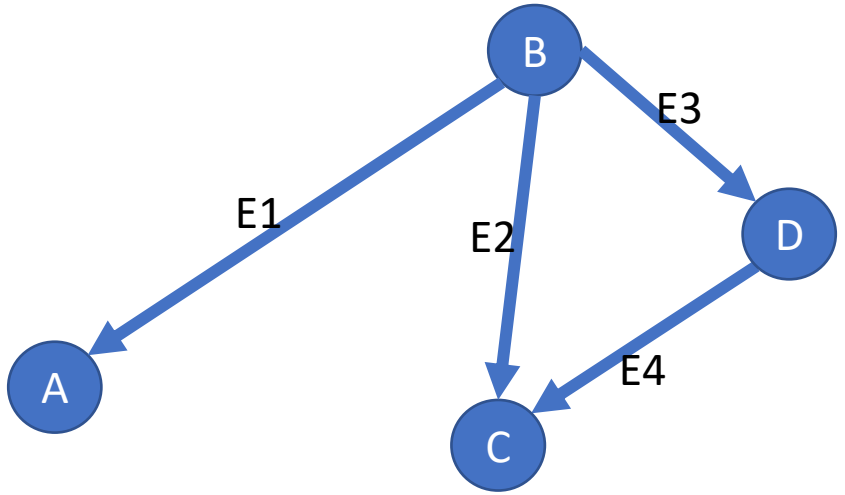
Topological Sort

- A *topological sort* (new term we are looking at) of a directed acyclic graph (DAG) is essentially a "certain" ordering of its vertices.

- We will look at what that ordering really is, and look at a more formal definition, as well as an algorithm to determine this ordering.

Topological Sort of a directed graph

- Lets think as if the vertices in the directed graph represent **some job to be done**.
- And if vertex B has an edge going into vertex A, then u think of it as B needs to happen before A can happen, i.e., A is dependent on B.
  - So, in the topological sort, **B has to be placed before A**
  - And for the same reason, B has to be placed before D and C.
  - And D before C … which means **both** B and D have to be before C
  - In fact, B is NOT dependent on any vertex, so this means B has to be the **first vertex** in this sort.



- So, a topological sort here could be:
- B A D C
- OR
- B D A C

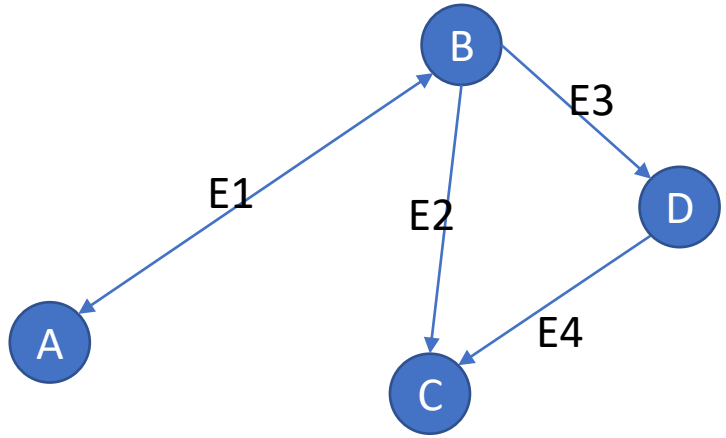
Topological Sort of a directed graph

- Now, earlier we said that this sort can happen on a directed *acyclic* graph.

- Lets look at a graph with cycles, and see what issue we would run into:
  - This graph has and edge (A,B) as well as (B,A).
  - This is a cycle A → B → A

  - This means
    - A needs to come before B in a topo sort
    - But also that B needs to come before A.
    - This is not possible to resolve, and is known as a *cyclic* dependency.

    - So this graph is not a DAG.
    - And we cannot do a topo sort on this.



Topological Sort

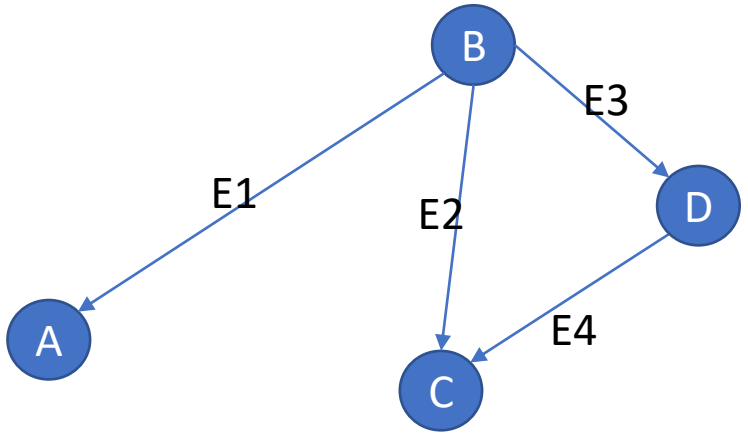
- So, to put it a little formally:

  - Topological sorting of a directed acyclic graph is where
    - a linear ordering of the vertices is created
    - such that if there is an edge (V_1, V_2),
      - then in the ordering, V_1 occurs before V_2.

- Linear ordering = layout the vertices one after another (order matters)

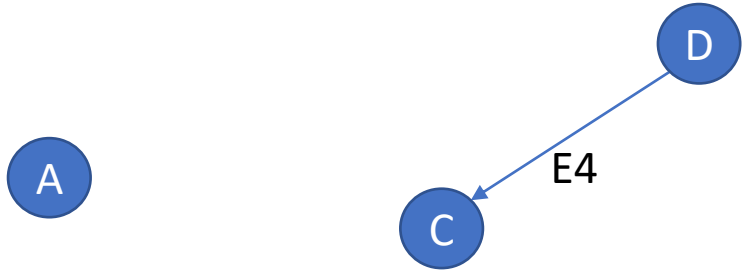
Topological sort algorithm

- Start from a vertex that has no incoming edges (B)
  - This means it has no dependencies, and can be the first one in the sort.
    - **Note**: If u cannot find such a vertex, u have a loop (cycle).
  - Print this vertex to output list
  - output list : B

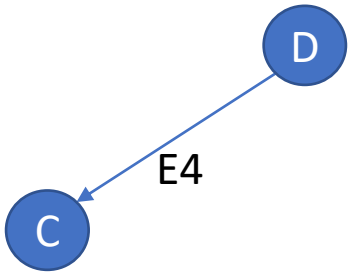


Topological sort algorithm

- So, we took out vertex B and any edges going out of B.
- Find the next vertex with no incoming edges.
  - We have A and D.
  - Lets pick A
  - Print A
  - Remove
    - Vertex A
    - And any edges going out of A (none)

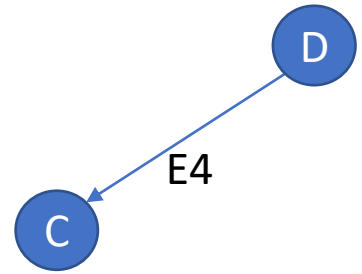


  - output list : B A



Topological sort algorithm

- So, we took out vertex A and any edges going out of A.
- Find the next vertex with no incoming edges.
  - We have D.
  - Print D
  - Remove
    - Vertex D
    - And any edges going out of D
  - output list : B A D



Topological sort algorithm

- Now we are left with only vertex C
  - Print C
  - Remove C
  - No more vertices left.
  - We are done.

- output list : B A D C

- Lets take a look at the algorithm again, this time with the complexity in mind.
- Our first step was:
  - Start from a vertex that has no incoming edges

- In order to achieve this, we need to have these:
  1. computed the in-degree of all vertices in the graph.
     - This would be a one time step (initialization)
     - O (E)
  2. Find a vertex with in-degree zero
     - This is a linear search in the array computed in step 1 above.
     - This means a linear search in an array of size V (number of vertices)
     - And this would need to be done V times
     - $O(V^2)$
  3. Reduce the in-degrees of adjacent vertices
     - This is where we removed the edges going out of the current selected vertex
     - O(E)
  4. Mark vertex
     - O (V)

- So, overall complexity here is $O (V^2) + O (E)$ … which is quadratic, and hence not that good.

- The step that causes the quadratic complexity is the search for vertex with in-degree 0

- So, lets see if we can improve that step.
- During initialization, we add the following:
  - Create a queue. O (1)
  - Initialize it with in-degree 0 vertices. O (V)
- In the step where we reduce the in-degrees of vertices, if the in-degree becomes 0, push that vertex onto this queue.
  - Pushing onto queue is O (1)
  - Do this for a total of V vertices.
    - O (V)

- So, we reduced O (V^2) to O (V), for a total time complexity of
  - O (V) + O (E)

Alien alphabet

An alien language uses English letters, but their letter ordering is different (i.e., its not “abcdef…z”)

Given:
A list of sorted words from the alien language.

Problem:
return a string containing the letters in the right order. (e.g.: For English language, it would be like “abcde….z”)

See examples next.

Alien alphabet

1.

Given words in sorted order:

```
"zb",
"za"    // b --> a
We cannot infer if z comes before ‘b’ and ‘a’ or after or in-between.
Order could be “zba” or “bza” or “baz”
If we added “bz”
“bz"    // z --> b.

Order is “zba”
```

Alien alphabet

2.

Given words in sorted order:

```
"bz",
"zb",    // b --> z
"za",    // b --> a
"ab"     // z --> a
```

Order is “bza”

Alien alphabet

3.

Given words in sorted order: ← Note that I am using the English alphabet order to make it easy to understand.

```
"abc",
"aef",    // b --> e
"bbc",    // a --> b
"bdc",    // b --> d
"bzz"     // d --> z


// From this, we can infer a,b, {d,e}, z  Can’t infer order between ‘d’ and ‘e’ or between ‘e’ and ‘z’
```

Alien alphabet

3.

Given words in sorted order: ← Note that I am using the English alphabet order to make it easy to understand.

```
"abc",
"aef",    // b --> e
"bbc",    // a --> b
"bdc",    // b --> d
"bzz“     // d --> z
// From this, we can infer a,b, {d,e,z}   Can’t infer order between ‘d’ and ‘e’ or between ‘e’ and ‘z’

// if we add "daz" and "eff", we can infer  d --> e
"daz"
"eff"     // d --> e
// But we still cannot infer order between ‘e’ and ‘z’.
```

Alien alphabet

3.

Given words in sorted order: ← Note that I am using the English alphabet order to make it easy to understand.

```
"abc",
"aef",    // b --> e
"bbc",    // a --> b
"bdc",    // b --> d
"bzz"
// From this, we can infer a,b, {d,e,z}   Can’t infer order between ‘d’ and ‘e’ or between ‘e’ and ‘z’

// if we add "daz" and "eff", we can infer  d --> e
"daz"
"eff"     // d --> e
// But we still cannot infer order between ‘e’ and ‘z’.
// If we add “zzz”:
"zzz"   // e --> z

Order is “abdez”
```

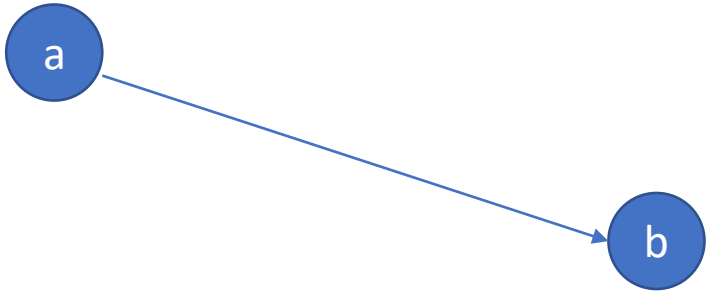
Alien alphabet

To solve this, we will do the following steps.

1. Build the adjacency list.
    1. Letters are the vertices.
    2. If an edge goes from a letter ‘a’ to a letter ‘b’, it means that ‘a’ comes before ‘b’ in the order.

    3.
        1. "abc",
           "bdz"
           adjacency list of ‘a’ will contain ‘b’

        2. "abc",
           "adz"
           adjacency list of ‘b’ will contain ‘d’

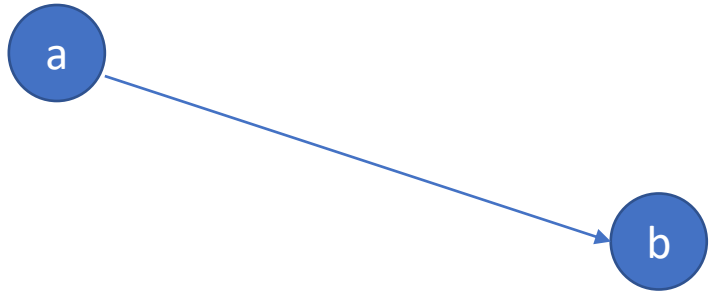


Alien alphabet

1. Build the adjacency list.
    1. Letters are the vertices.
    2. If an edge goes from a letter ‘a’ to a letter ‘b’, it means that ‘a’ comes before ‘b’ in the order.

    3.
        1. `"abc",`
           `"bdz"`
           adjacency list of ‘a’ will contain ‘b’

        2. `"abc",`
           `"adz"`
           adjacency list of ‘b’ will contain ‘d’



2. Compute the indegrees of the vertices.

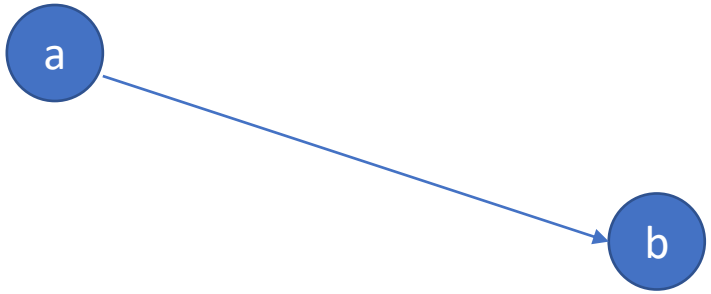
    1. We can do this as we are building the adjacency list.
    2. In the example above: `"abc", "bdz"`

        1. Indegree ‘a’ would be 0.
        2. Indegree ‘b’ would be 1.

Alien alphabet

1. Build the adjacency list.
    1. Letters are the vertices.
    2. If an edge goes from a letter ‘a’ to a letter ‘b’, it means that ‘a’ comes before ‘b’ in the order.
    3.
        1. `"abc",`
           `"bdz"`
           adjacency list of ‘a’ will contain ‘b’

        2. `"abc",`
           `"adz"`
           adjacency list of ‘b’ will contain ‘d’



2. Compute the indegrees of the vertices.

    1. We can do this as we are building the adjacency list.
    2. In the example above: `"abc", "bdz"`

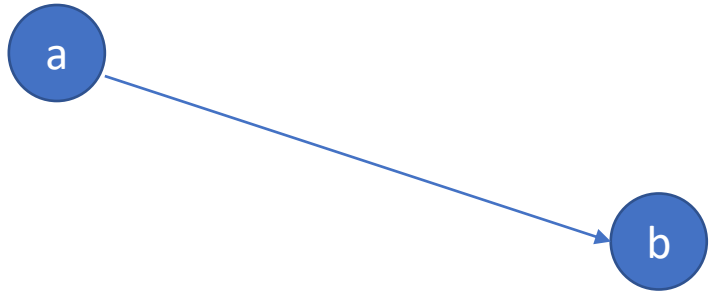
        1. Indegree ‘a’ would be 0.
        2. Indegree ‘b’ would be 1.

    3. The vertex with an indegree of 0 would be the first letter in the alphabet.
    4. Lets call this V_0
        1. Note: If complete info is not given, its possible u will have more than one vertex with indegree of 0.
    5. Now, we will remove the outgoing edges from V_0
        1. The way we do this is to reduce the indegree of descendants of V_0

Alien alphabet

1. Build the adjacency list.
    1. Letters are the vertices.
    2. If an edge goes from a letter ‘a’ to a letter ‘b’, it means that ‘a’ comes before ‘b’ in the order.
    3.
        1. `"abc",`
           `"bdz"`
           adjacency list of ‘a’ will contain ‘b’

        2. `"abc",`
           `"adz"`
           adjacency list of ‘b’ will contain ‘d’



2. Compute the indegrees of the vertices.

    1. We can do this as we are building the adjacency list.
    2. In the example above: `"abc", "bdz"`

        1. Indegree ‘a’ would be 0.
        2. Indegree ‘b’ would be 1.

    3. The vertex with an indegree of 0 would be the first letter in the alphabet.
    4. Lets call this V_0
        1. Note: If complete info is not given, its possible u will have more than one vertex with indegree of 0.
    5. Now, we will remove the outgoing edges from V_0
        1. The way we do this is to reduce (by 1) the indegree of descendants of V_0
        2. In this example, the indegree of ‘b’ would now become 0.

Alien alphabet

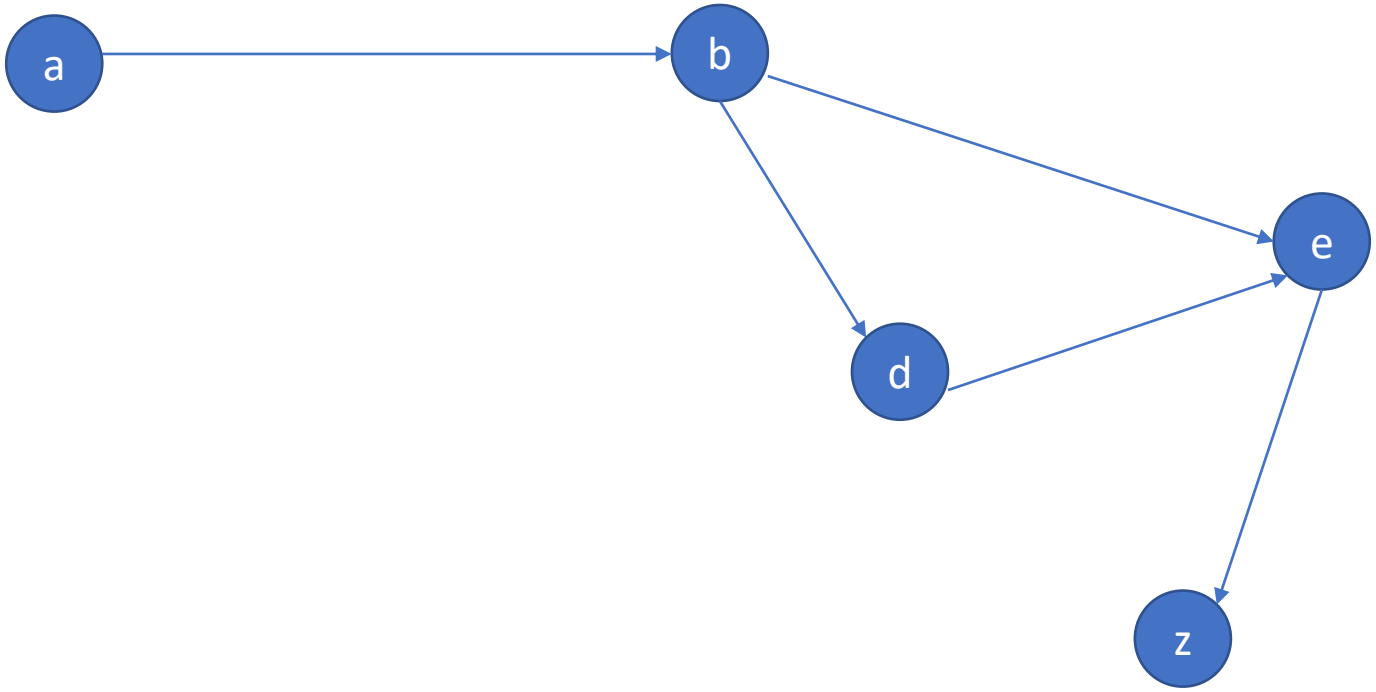
1. ‘b’ would be the next letter in the alphabet.
2. Now we would remove the outgoing edge from ‘b’
    1. This means we reduce (by 1) the indegree of all vertices in ‘b’ adjacency list.

3. We would continue doing this until all indegrees become 0.

Alien alphabet

1.

We build the graph



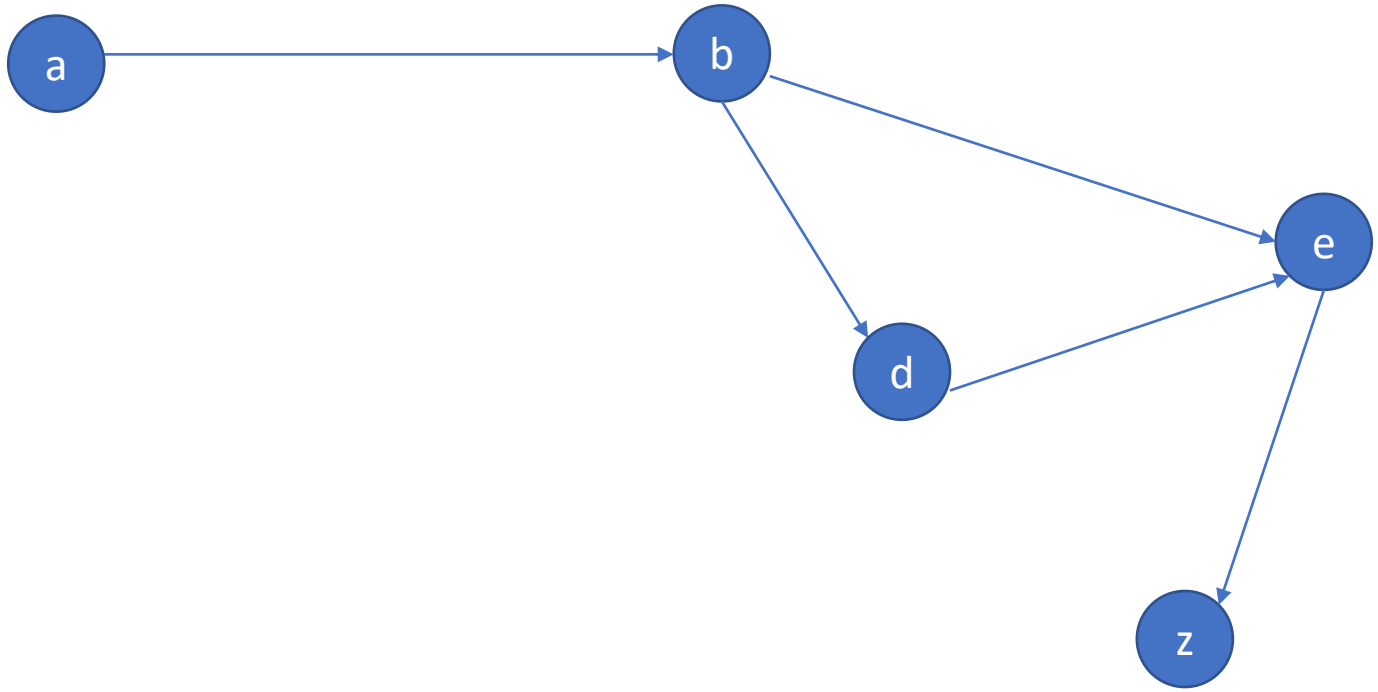
Alien alphabet

‘a’ has indegree of 0

Add ‘a’ to order string (order will contain the letters in the required order)

order = “a”

Now, remove the outgoing edges from ‘a’



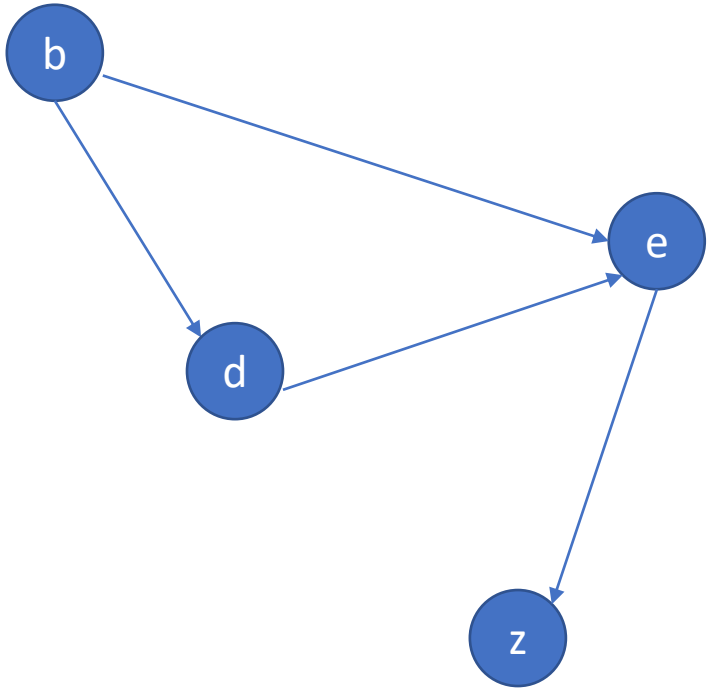
Alien alphabet

‘a’ has indegree of 0

Add ‘a’ to order string (order will contain the letters in the required order)

order = “a”

Now, remove the outgoing edges from ‘a’ ➔ Reduce indegree of ‘b’ by 1



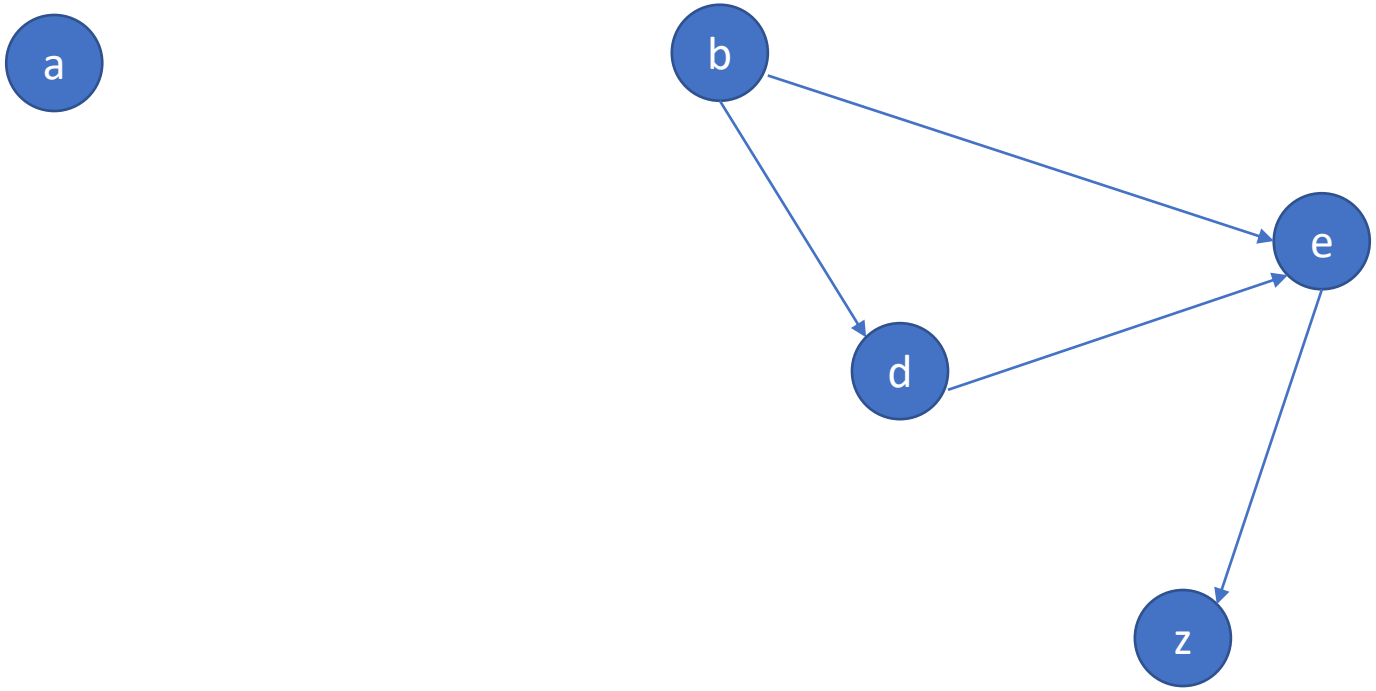
Alien alphabet

Now, ‘b’ has indegree of 0

Add ‘b’ to order string (order will contain the letters in the required order)

order = “ab”

Now, remove the outgoing edges from ‘b’ ➔ Reduce indegree of ‘d’ and ‘e’ by 1



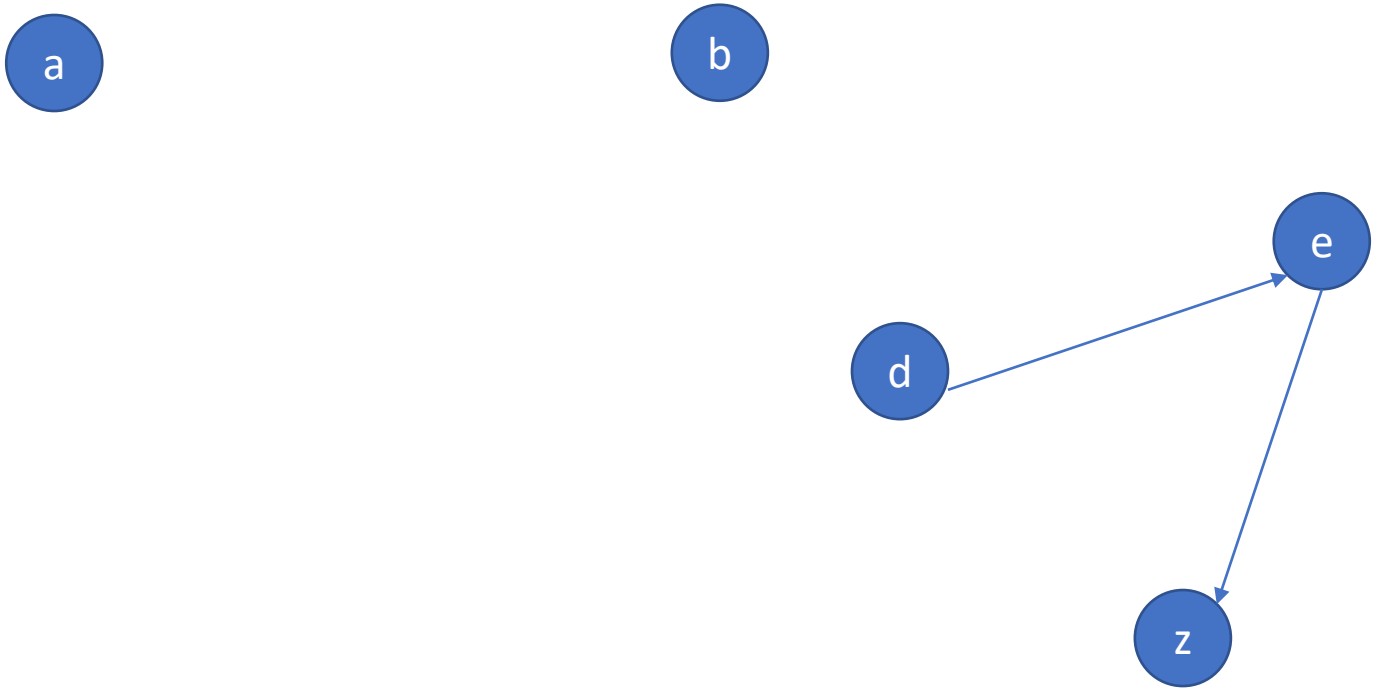
Alien alphabet

Now, ‘d’ has indegree of 0

Add ‘d’ to order string (order will contain the letters in the required order)

order = “abd”

Now, remove the outgoing edges from ‘d’ ➔ Reduce indegree of ‘e’ by 1



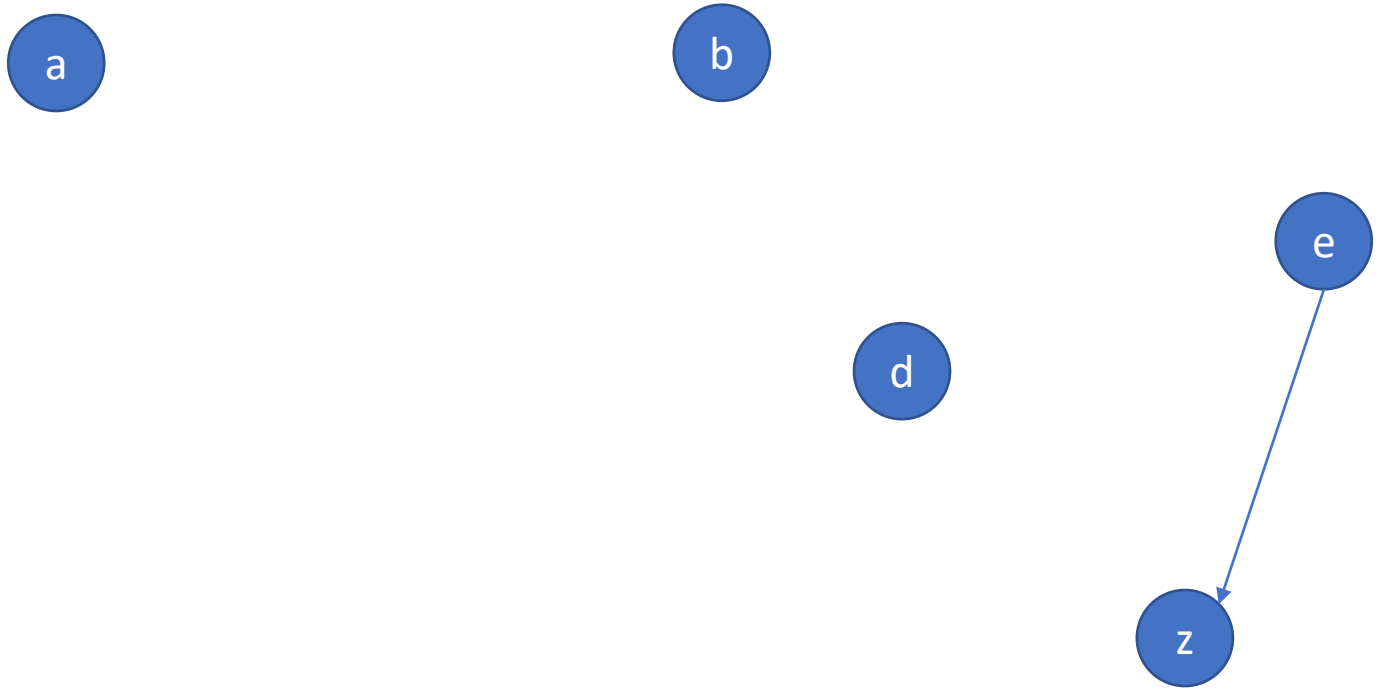
Alien alphabet

Now, ‘e’ has indegree of 0

Add ‘e’ to order string (order will contain the letters in the required order)

order = “abde”

Now, remove the outgoing edges from ‘e’ ➔ Reduce indegree of ‘z’ by 1



Alien alphabet

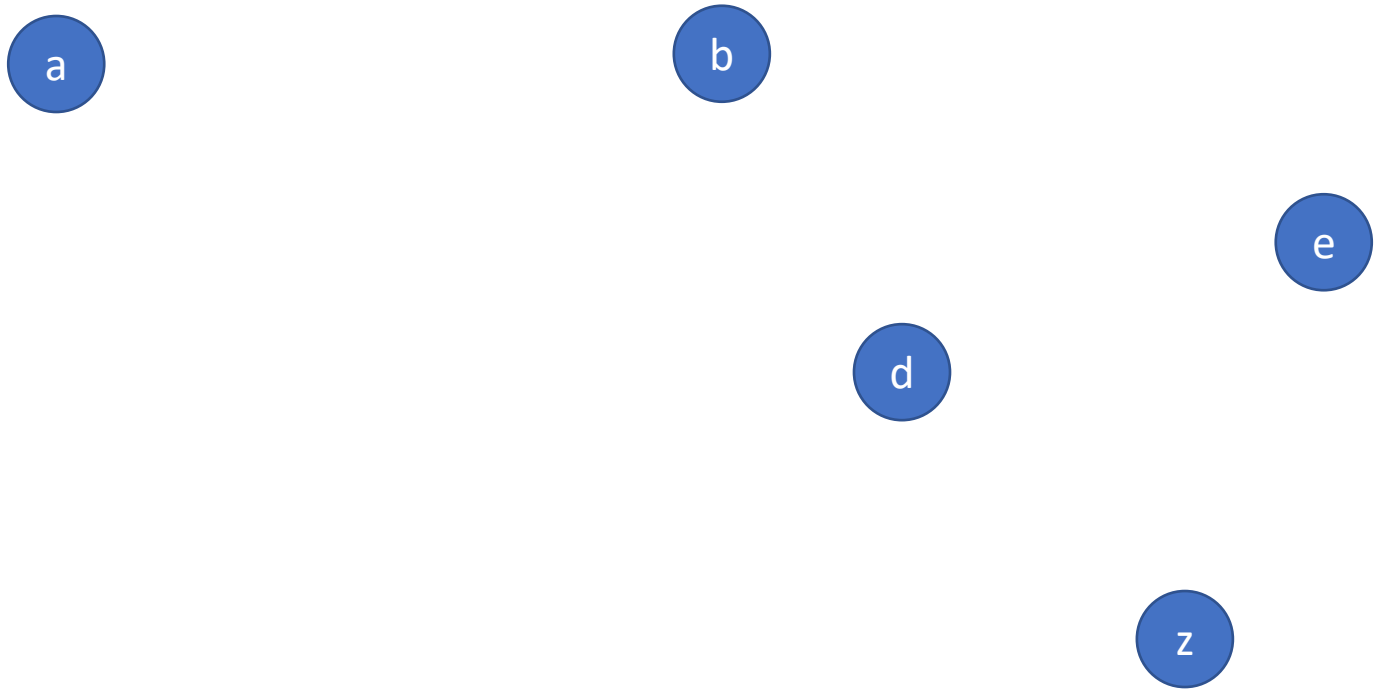
Now, ‘z’ has indegree of 0

Add ‘e’ to order string (order will contain the letters in the required order)

order = “abdez”

There are no outgoing edges from ‘z’

We r done.



Final order determined is “abdez”

Alien alphabet verification

Something u can try in ur own time, not to be submitted

An alien language uses English letters, but their letter ordering is different (i.e., its not “abcdef…z”)

Given:
Order of letters in this alien alphabet.
Sequence of words written in the alien language.

Problem:
Determine if the sequence of words given are sorted lexicographically in this language.
if sorted, return true, else return false.

1.
Given
order = "hlabcdefgijkmnopqrstuvwxyz“ words = ["hello", "leetcode"]
Return true in this case, because 'h' comes before 'l’ , hence the word sequence is sorted.

2.
order = "worldabcefghijkmnpqstuvxyz" words = ["word", "world", "row"]
Return false because 'd' comes after 'l’ , so “word” should have been *after* “world” (‘r’ comes after ‘w’, so “row” being at the end is fine)