Fulati Aizihaer **Discussion:** 1F

Time: 2:00pm - 3:50pm, Friday

TA name: Parshan

1. Exercise 3 Page 107

Algorithm: To compute a topological ordering of G

- Create a output list to store the ordering of the nodes
- While there exists a node **v** that has no incoming edges:
    - Delete **v** from G and append **v** to the output list
        - Also delete all the edges **e** that connects **v** to other nodes in G
    - If there are more than one node that has no incoming edges, then arbitrarily pick one of the node **u:**
        - Recursively compute a topological ordering of the **u** and append **u** to the output list after **v**
- After the while loop finishes running:
    - If there are still nodes in the graph:
        - Return the nodes that are left in the graph (a cycle in G)
    - Else:
        - Return the output list which contains the topological ordering of G

Proof:

- Assume that G has a topological ordering v1, v2, …, vn.
- Using the algorithm above, we cycle through the nodes that doesn't have incoming edges one by one:
    - We append the node to the list then delete the node and its edges
- After a while, if we can't find anymore nodes that has no incoming edges in our graph, we check the graph:
    - If there are no more nodes in the graph, then our graph G is a DAG and we output our list of topological ordering
    - If there are still nodes in our graph, then our graph contains a cycle. (where all the nodes are connected and have incoming edges)

Time Complexity:

- It takes O(1) to append the nodes onto the list
- While loop takes worst case O(v + e) time since it uses DFS, where v is number of nodes and e is the number of edges, which means that it has to go through all of the nodes and the edges in graph G.
- It takes O(v) time to delete node v from G in worst case
- It takes O(e) to delete all the edges from the graph in the worst case
- It takes O(v) time to check if there are remaining nodes in the graph and print it out

Fulati Aizihaer **Discussion:** 1F

Time: 2:00pm - 3:50pm, Friday

TA name: Parshan

2. Exercise 4 on Page 107

Algorithm:

- Create an undirected graph G = (v,e), where the butterflies are represented by **nodes** and only “same” or “different” judgements are represented by the **edges** that are between two nodes.
- Initialize an empty list ‘visited’ that will track visited nodes.
- Initialize an empty queue and append the initial node **v** to it
- While the queue is not empty:
    - Append **v** to the visited list
    - Dequeue the front node **v** from the queue
    - Arbitrarily choose a label for **v** as one of the two species
    - For each edge **e** between **v** and another node **u**
        - If **u** is not in the visited list:
            - Append **u** to the visited list
            - Enqueue **u** into the queue
            - If **e** is labeled as **same** judgment:
                - Set the label of **u** will be the same as the label of **v**
            - Else (if **e** is labeled as **different** judgment):
                - Set the label of **u** will be different from the label of **v**
        - Else (u is already in the visited list):
            - If **e** is labeled as **same** judgment, and the label of **u** is different from the label of **v:**
                - Terminate the loop and return that the judgment is **inconsistent**
            - Else if **e** is labeled as **different** judgment, and the label of **u** is same as the label of **v**:
                - Terminate the loop and return that the judgment is **inconsistent**
- If the while loop ends without returning anything, then return that the judgment is **consistent**

Fulati Aizihaer **Discussion:** 1F

Time: 2:00pm - 3:50pm, Friday

TA name: Parshan

Proof by given a counter example:

- Assume we have three butterflies (x, y, z) and two species
- Assume we have the judgment that:
    - x is a different species as y
    - y is a different species as z
    - x is a different species as z.
- This is inconsistent based on our algorithm because:
    - If x is different species as y, and y is different species as z, it means that the species x and z have to be the same since there are only two species of butterflies.

Time Complexity:

- While loop takes worst case O(v + e) time as it uses BFS, where v is number of nodes and e is the number of edges, which means that it has to go through all of the nodes and the edges in graph
    - Takes O(1) to append values to the visited list
    - Dequeuing the front node takes O(1)
        - Checking if u is in visited list takes O(1)
        - Appending to the list takes O(1)
        - Enqueing takes O(1)
        - Checking label of the edge takes O(1)
        - Adjusting the labels of the nodes take O(1)
- Overall time complexity is O(v + e) where v is the number of nodes (butterflies) and e is the number of edges (judgments). Linear time complexity

Fulati Aizihaer **Discussion:** 1F

Time: 2:00pm - 3:50pm, Friday

TA name: Parshan

3. Exercise 9 on page 110

Algorithm:

- Initialize an empty list 'visited' that tracks visited nodes
- Initialize a list **L[i]**, set **L[0]** to node **s** and set **i** = 0, where **i** represents the layers.
- While **L[i]** is not empty:
    - Initialize an empty **list L[i+1]**
    - For each node **u** inside **L[i]**
        - Examine each node **v** that is incident to **u. (u,v)**
        - If **v** is not on the visited list:
            - Append **v** to the visited list
            - Append **v** to the list **L[i+1]**
    - If the size of **L[i+1]** is equal to 1, then terminate the loop and return the node **v** that is inside **L[i+1]**
    - Increment **i** by one

Proof by contradiction:

- Suppose there is no such node **v** that can destroy all **s-t** paths by removing **v**
- It means that there will be two paths that allow you to get from **s** to **t:**
    - So that even if you delete a node **u** from path 1, it will destroy path 1, however you can still use path 2 to get from **s** to **t**
- However this is not possible because we don't have enough number of nodes to form these two paths since:
    - We have **n** nodes, the distance between **s** and **t** must be at least **n/2 + 1** layer
    - Additionally, nodes **s** and **t** are not included in these layers meaning that there will only be **n - 2** nodes between these layers
    - If we put 2 nodes on each layer between **s** and **t**, then our number of nodes in the graph will exceed n which won't work.
- Therefore, we must have at least one level between the **s** and **t** layer that contain just one node.

Time Complexity:

- Since we are using BFS to traverse through the algorithm, we are iterating through the graph's v (the nodes), and e (the edges) only once.
- Therefore, we have the time complexity of O(v + e).

Fulati Aizihaer **Discussion:** 1F

Time: 2:00pm - 3:50pm, Friday

TA name: Parshan

4. Exercise 11 on page 111

Algorithm:

- Create a directed graph **G** where the nodes represents the computer at a distinct time and the edges represent the communication between two computers at a distinct time:
    - For each triples ($\mathbf{C_i}$, $\mathbf{C_j}$, $\mathbf{t_k}$)
        - Create nodes ($\mathbf{C_i}$, $\mathbf{t_k}$) and nodes ($\mathbf{C_j}$, $\mathbf{t_k}$)
        - Create a directed edge from ($\mathbf{C_i}$, $\mathbf{t_k}$) to ($\mathbf{C_j}$, $\mathbf{t_k}$)
        - Create a directed edge from ($\mathbf{C_j}$, $\mathbf{t_k}$) to ($\mathbf{C_i}$, $\mathbf{t_k}$)
    - For each computer $\mathbf{C_i}$ that communicates more than one time (occurs at several **t**):
        - If the time $\mathbf{t_k} > \mathbf{t_p}$:
            - Create a directed edge from ($\mathbf{C_i}$, $\mathbf{t_k}$) to ($\mathbf{C_i}$, $\mathbf{t_p}$)
- Create a hashmap where the keys are the computers and the values are the timestamps of the communications of that computer

- Suppose a virus is introduced at ($\mathbf{C_a}$, $\mathbf{t_x}$) and we want to find if ($\mathbf{C_b}$, $\mathbf{t_y}$) is infected
- Initialize a list that contains whether or not a node has been visited
- Initialize an empty queue.
- Find the node ($\mathbf{C_a}$, $\mathbf{t_{x'}}$), where time **x'** is when $\mathbf{C_a}$ communicated with someone first time after time **x**
- Mark visited[($\mathbf{C_a}$, $\mathbf{t_{x'}}$)] as visited
- append the node ($\mathbf{C_a}$, $\mathbf{t_{x'}}$) from the first triple onto the queue as starting point
- While the queue is not empty:
    - Dequeue the front node (**C**, **t**) from the queue
    - For each edge e between (**C**, **t**) and (**C'**, **t'**):
        - If (**C'**, **t'**) is not in the visited list:
            - Append (**C'**, **t'**) to the visited list
            - Enqueue (**C'**, **t'**) to the queue
            - If **C'** is $\mathbf{C_b}$ and **t'** is less than $\mathbf{t_y}$
                - Terminate the loop and return that $\mathbf{C_b}$ was infected by time $\mathbf{t_y}$
- If the while loop ends without returning anything, then return that $\mathbf{C_b}$ was not infected by time $\mathbf{t_y}$

Fulati Aizihaer

Discussion: 1F

Time: 2:00pm - 3:50pm, Friday

TA name: Parshan

Proof:

- Suppose we have the following data: (C2, C3, 9), (C4, C3, 7), (C1, C4, 5), (C1, C2, 14),
- Say C1 gets infected at time 4
- Will C2 get infected by time 10?
    - C1 occurs at time 6 first right after it gets infected at time 4
    - At time 6, C1 infects C4
    - C4 then infects C3 at time 7
    - Where then C3 will infect C2 at time 9
- Yes, C2 does get infected before time 10

Time Complexity:

- Creating the nodes take O(n) time where n is the number of computers
- Creating the edge take O(m) time where m is the number of timestamps for each computer
- Creating a hashmap takes O(1) time
- Initializing the list and queue takes O(1) time
- Finding the node ($\mathbf{C_a}$, $\mathbf{t_{x'}}$) takes O(1) time since we are using a hashmap
- In the while loop which uses BFS traversal, it takes O(n+m), where n is the number of nodes and m is the number of edges of the nodes.
- Overall our time complexity for this algorithm is O(n+m), where n is the number of computers and m is the number of timestamps for each computer.

Fulati Aizihaer **Discussion:** 1F

Time: 2:00pm - 3:50pm, Friday

TA name: Parshan

5. Exercise 12 on page 112

Algorithm: Create a directed graph G and compute a topological ordering of G:

- Creating the directed graph G:
  - For each person $\mathbf{P_i}$:
    - Create node $\mathbf{b_i}$ representing birth date
    - Create node $\mathbf{d_i}$ representing death date.
    - Create a directed edge between $\mathbf{b_i}$ to $\mathbf{d_i}$
  - For each fact:
    - If $\mathbf{P_i}$ died before $\mathbf{P_j}$
      - Create a directed edge between $\mathbf{d_i}$ to $\mathbf{b_j}$
    - If $\mathbf{P_i}$ life span overlapped partially with $\mathbf{P_j}$
      - Create a directed edge between $\mathbf{b_i}$ to $\mathbf{d_j}$
      - Create a directed edge between $\mathbf{b_j}$ to $\mathbf{d_i}$

- Compute a topological ordering of G:
- Create a output list to store the proposed dates of birth and death of all the people
  - While there exists a node **v** that has no incoming edges:
    - Delete **v** from G and append **v** to the output list
      - Also delete all the edges **e** that connects **v** to other nodes in G
    - If there are more than one node that has no incoming edges, then arbitrarily pick one of the node **u:**
      - Recursively compute a topological ordering of the **u** and append **u** to the output list after **v**
  - After the while loop finishes running:
    - If there are still nodes in the graph:
      - Return the nodes that are left in the graph (a cycle in G), which shows that no such dates can exist which means the data is not internally consistent.
    - Else:
      - Return the output list which contains the topological ordering of the graph G (containing birth and death dates of the n people), showing that we have a consistent data

---

Fulati Aizihaer **Discussion:** 1F

Time: 2:00pm - 3:50pm, Friday

TA name: Parshan

Proof:

- Assume that G has a topological ordering.
- Using the algorithm above, we cycle through the nodes that doesn't have incoming edges one by one:
    - We append the node to the list then delete the node and its edges
- After a while, if we can't find anymore nodes that has no incoming edges in our graph, we check the graph:
    - If there are no more nodes in the graph, then our graph G is a DAG
        - We output our list of topological ordering of the birth and death dates of all the people. It is internally consistent
    - If there are still nodes in our graph, then our graph contains a cycle.
        - This is not possible because having a cycle means that each of the events in this cycle is waiting for the previous event in the cycle to happen, and since all of the events in the cycle have incoming cycles, nothing will happen. Not internally consistent

---

Time Complexity:

- Creating the graph takes O(v + e):
    - Creating nodes for birth and death dates of each person is O(2v), v is the number of people. It can be said to be about O(v).
    - Creating directed edges based on the facts takes O(e), where e is the number of directed edges.
- Computing Topological Ordering:
    - It takes O(1) to append the nodes onto the list
    - While loop takes worst case O(v + e) time since it uses DFS, where v is number of nodes and e is the number of edges, which means that it has to go through all of the nodes and the edges in graph G.
    - It takes O(v) time to delete node v from G in worst case
    - It takes O(e) to delete all the edges from the graph in the worst case
    - It takes O(v) time to check if there are remaining nodes in the graph
- Overall, it takes 2 * O(v + e) which is about O(v + e) time complexity in the worst case.

Fulati Aizihaer **Discussion:** 1F

Time: 2:00pm - 3:50pm, Friday

TA name: Parshan

6. Given an array **arr[]** of size **N**, the task is to find the minimum number of jumps to reach the last index of the array starting from index 0. In one jump you can move from current index **i** to index **j**, if **arr[i] = arr[j]** and **i != j** or you can jump to **(i + 1)** or **(i – 1)** .
   Note: You can not jump outside of the array at any time.
   Example:
   Input: arr = {100, -23, -23, 404, 100, 23, 23, 23, 3, 404}
   Output: 3
   Explanation: Valid jump indices are 0 - > 4 - > 3 - > 9.

Algorithm:

- If size of the array is equal to 0 or 1
  - Return the jump count as 0 since no jumps are needed
- Create a hashmap where the keys are the values in the array and the values are the list of indices where those values appear.
- Initialize a list that contains boolean values on whether or not the index in the array is already visited
  - Append visited[0] as **true** since its visited, and append false for all other indices
- Initialize an empty queue and append the first index of the array to it
- Initialize an integer count to keep track of the minimum jump taken to reach the end
- while the queue is not empty:
  - Increment the counter by one
  - for each index in queue:
    - Dequeue the first index **i** from the queue
    - If **i** is equal to the last index of the array:
      - Terminate the loop and return the count
    - if **i - 1** is greater than or equal to 0, and visited[i-1] is false:
      - enqueue **i - 1** into the queue
      - visited[i-1] = true
    - if **i + 1** is less than the size of the array, and visited[i+1] is false:
      - if **i + 1** is equal to the last index of the array:
        - Terminate the loop and return the count
      - enqueue **i + 1** into the queue
      - visited[i+1] = true
    - for each index **j** in **hashmap[arr[i]]**:
      - if **j** is not equal to **i** and visited[j] is false:
        - if **j** is equal to the last index of the array:
          - terminate the loop and return count
        - enqueue **j** into the queue
        - Visited[j] = true

        - remove the value **arr[j]** from the hashmap (to avoid revisiting it)
- If the while loop doesn't return anything, then return the count

---

Proof:

- In the example {100, -23, -23, 404, 100, 23, 23, 23, 3, 404}
- Initialize the hashmap, visited list, queue and count
- Use BFS to traverse starting from index 0 of the array
- Explore the potential jumps based on the conditions below:
    - **arr[i] = arr[j]** and **i != j**
    - **(i + 1)**
    - **(i – 1)**
- Mark the visited indices
- Increment the count after traversing through each layer
- When we reach the end of the array, which in this case is value 404 (at index 9), the algorithm terminates and returns the count.
- Therefore, the algorithm above is proven to work correctly.

Time Complexity:

- Checking the size of array takes O(1) time
- Creating a hashmap takes O(n) for all the values in the array
- Initializing a visited list with boolean values take O(n) for all the values in the array
- Initializing a queue and appending the index takes O(1) time
- In the while loop which uses BFS traversal:
    - Time that takes to finish the queue takes O(k), which k is the value for the size of the queue
    - Exploring all of the indices of value takes O(m), where m is the number of indices with same value
    - Removing the values arr[j] from our hashmap takes O(1)
- Overall our time complexity for this algorithm is O(n), where n is the size of the array.