Fulati Aizihaer **Discussion:** 1F

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

TA name: Parshan

1. Exercise 10 Page 110

Algorithm:

- We are given an undirected graph G = (V,E), and we have two nodes **v** and **w** in G.
- Initialize a list **'visited'**, and append the initial node **v** to it
- Initialize a **queue** and append the initial node **v** to it
- Initialize a list **L[nodes],** where **L[nodes]** represent the layer where each nodes are at, and set **L[v]** as 0
- Initialize a **count** variable
- While the **queue** is not empty:
    - Dequeue the front node **n** from the queue
    - For every node **n'** that is connected to **n** by an edge
        - If **n'** is not on the visited list:
            - Append **n'** to the visited list
            - Append **n'** to the queue
            - Append **n'** to the list **L[n']** and make it equal to **L[n] + 1**
        - if **n'** is **w**:
            - Increase **count** by one
- Return **count**.

Proof:

- Given a root node **v** at L_0, a bunch of nodes **x** at L_i, and nodes **w** at L_j
- And that path increases by exactly one step from one layer to the next
- In order to find the total amount of shortest path from **v** to **w**.
    - One shortest path will take, from **v** → any node **x** in L_i, then from that **x** → **w**.
    - To count the total number of shortest paths, we take the amount of nodes **x** there is in layer L_i, and count the number of edges from those **x** connected to **w**.
    - So our shortest path is the sum of edges from nodes **x** in layer L_i to **w**

Time-Complexity:

- Since we are using BFS, our run time is O(m+n) where n is the number of nodes and m is the number of edges.
- Increasing the count takes O(1)
- Overall, the runtime of this algorithm is O(m+n)

Fulati Aizihaer **Discussion:** 1F

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

TA name: Parshan

2. Exercise 6 on page 108

Proof: Since we are trying to prove **G** = tree **T**, use **contradiction** to prove that **G** is not a tree.

- Assume that **G**, is not a tree, such that **G** contains an edge **(u,v)** that doesn't exist in the tree **T**.
- Since **T** is a **DFS** tree, one node will be the ancestor of another node, which means that the distance from **u** to **v** will at most be one.
- Since **T** is a **BFS**, the distance between nodes **u** and **v** will also be at most one.
- Therefore, **(u,v)** has to be an edge of **T**, meaning that all the edges in **G** exist in **T**. Meaning that **G** is a Tree, contradicting our initial statement.

Fulati Aizihaer **Discussion:** 1F

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

TA name: Parshan

3. Exercise 12 on page 193

(a):

Proof:

- This statement is **false**
- Assume we have streams (2000,1) and (1000,3), and parameter r = 1000.
    - If we order it as (2000,1), (1000,3), then our schedule would be **invalid** as the first stream doesn't satisfy the * constraint.
        - t = 1: 2000 </ 1000 * 1 NOT VALID.
    - However, if we **swap** the order, and order it as (1000,3), (2000,1), then our schedule would be **valid**.
        - t = 3: 1000 < 1000 * 3
        - t = 4: 1000 + 2000 < 1000 * 4 VALID
- In conclusion, you don't have to have each of the streams to satisfy the constraint, you just have to order them correctly to satisfy the constraint.

(b):

Algorithm:

- Initialize a **bitsPerSeconds** list that contains the <u>number of bits</u> *divided by* <u>time duration</u>, b_i / t_i for each stream **i** in **n** and assign those values to corresponding stream **i**
- Sort the **bitsPerSeconds** list in an increasing order
- For each stream **i** corresponding to its value in the sorted **bitsPerSeconds** list**:**
    - Initialize **totalBits** and **totalTime** values, and set them equal to 0
    - **totalBits** = **totalBits** + b_i
    - **totalTime** = **totalTime** + t_i
    - If **totalBits** >= **r** * **totalTime** , where **r** is the fixed parameter
        - Return "**Invalid**" since there doesn't exist a valid schedule with right order
- Return "**valid**" since there exists a valid schedule with right order

----

Fulati Aizihaer **Discussion:** 1F

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

TA name: Parshan

Proof:

- Assume we have the following set:
    - (b1, t1) = (2000, 1), (b2, t2) = (1000, 4) , (b3, t3) = (3000, 3)
    - With r = 1000
- First initialize the **bitsPerSeconds** list and get:
    - (b1/t1) = 2000, (b2/t2) = 250, (b3/t3) = 1000, → [2000, 250, 1000]
- We sort it and get:
    - (b2/t2) = 250, (b3/t3) = 1000, (b1/t1) = 2000, → [250, 1000, 2000]
      Corresponding **i** values → [(b2,t2), (b3,t3), (b1,t1)]
- Execute the loop based on the sorted order of the list:
    - First iteration:
        - totalBits = 0 + b2(1000)
        - totalTime = 0 + t2(4)
        - Loop continues since 1000 < 1000 * 4
    - Second iteration:
        - totalBits = 1000 + b3(3000)
        - totalTime = 4 + t3(3)
        - Loop continues since 4000 < 1000 * 7
    - Third iteration:
        - totalBits = 4000 + b1(2000)
        - totalTime = 7 + t1(1)
        - Loop continues since 6000 < 1000 * 8
- We have a **valid** schedule with the order above [(b2,t2), (b3,t3), (b1,t1)]

- If we didn't use the algorithm and tried sending them in the order given [(b1,t1), (b2,t2), (b3,t3)]. It would have had an **invalid** schedule since for the first stream, 2000 > 1000 * 1, meaning that it wouldn't **satisfy the constraint.**

Time-Complexity:

- It takes O(n) time complexity to divide number of bits divided by time duration, b_i/t_i for each stream **i** in **n** and put in a list and assign the **i** to the b_i/t_i .
- It takes O(n log n) time complexity to (heap) sort the **bitsPerSeconds** in increasing order
- Overall, the runtime of this algorithm is O(n log n)

Fulati Aizihaer **Discussion:** 1F

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

TA name: Parshan

4. Exercise 3 on page 189

Proof by induction: Our greedy algorithm “stays ahead” of all other solutions, and there are **n** amount of trucks

- Base Case: For **n = 1** truck
  - Consider that there is one truck, using the greedy algorithm, we will fit **x** amount of boxes onto the truck
  - Whereas using all other solutions, we will fit **y** amount of boxes, where **x >= y**;
  - This indicates that the greedy algorithm is better than or equal to any other solutions.
- Inductive Hypothesis: For **n** amount of trucks
  - Assume for **n** amount of trucks, the greedy algorithm will pack **x** amount of boxes, whereas other solutions will pack **y** amount, where **x >= y**
- Inductive Steps: Show that for **n+1** amount of trucks, greedy algorithm will pack more than all other solutions
  - For **n+1** trucks, Greedy algorithm can fit **x+1** amount of boxes
  - Whereas the other solutions can fit **y+1** amount of boxes onto the trucks
  - Since **x+1 >= y+1**, it means that our greedy algorithm “stays ahead” by packing more than or equal amount of boxes compared to all other solutions.
- Therefore, our greedy algorithm can pack more or equal to all other solutions, meaning that it will indeed minimize the number of trucks that are needed.

Fulati Aizihaer **Discussion:** 1F

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

TA name: Parshan

5. Exercise 6 on page 191

Algorithm:

- Assume we have **n** amounts of contestants.
- For each contestant, combine their running time and biking time.
- Sort the contestants in order of slowest to fastest combined running/biking time.
- Then send out the contestants to the **start** of the triathlon, **the pool**, based on that sorted order

----

Proof:

- Assume we have contestants **a** and **b**, where **a** enters before **b.**
- The biking and running combined time of **a** is smaller than that of **b**, shown by following $T_a < T_b$
- Without using our proposed algorithm, the competition goes like the following:
    - **a** will finish the race, then **b** will finish the race after it, and since **b** takes longer running and biking time, b will finish at time **x** and the competition will conclude
- However, if we use our algorithm and change the position of **a** and **b**
    - Such that **b** enters before **a** due to our sort
    - **b** will start first, then **a** will join after it and **a** will finish the race at time **y** and the competition will conclude
    - Since contestant **a** has a shorter biking and running time than **b**
        - **y** < **x**, meaning that:
        - Contestant **a** will finish the race earlier in this race than **b** did in the race prior to changing position (where we didn't use our algorithm), because **a** has shorter running and biking time compared to **b**.
- Therefore, our total time of the competition will decrease when we implement the algorithm above due to the fact that we sort the contestants.

Time-Complexity:

- It takes O(n) time complexity to combine each contestant's running time and biking time
- It takes O(n log n) time complexity to (heap) sort the contestants in order of slowest to fastest running/bike time.
- Overall, the runtime of this algorithm is O(n log n)

Fulati Aizihaer **Discussion:** 1F

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

TA name: Parshan

6. Given a matrix of dimension M * N, where each cell in the matrix can have values 0, 1 or 2 which has the following meaning:

0: Empty cell
1: Cells have fresh oranges
2: Cells have rotten oranges

A rotten orange at index (i,j) can rot other fresh oranges which are its neighbors (up, down, left, and right). If it is impossible to rot every orange then simply return -1.

The task is to find the minimum time required so that all the oranges become rotten.

Example Input:
Input: arr[][C] = { {2, 1, 0, 2, 1}, {1, 0, 1, 2, 1}, {1, 0, 0, 2, 1}};
Output: 2
Explanation: At 0th time frame:
{2, 1, 0, 2, 1}
{1, 0, 1, 2, 1}
{1, 0, 0, 2, 1}
At 1st time frame:
{2, 2, 0, 2, 2}
{2, 0, 2, 2, 2}
{1, 0, 0, 2, 2}
At 2nd time frame:
{2, 2, 0, 2, 2}
{2, 0, 2, 2, 2}
{2, 0, 0, 2, 2}

Fulati Aizihaer **Discussion:** 1F

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

TA name: Parshan

Algorithm:

- Initialize a **countTime** = 0 variable to track the time
- Initialize a **countFresh** = 0 variable to track number of fresh oranges
- Initialize a **queue** to store the rotten oranges
- For each cell **(i, j)** in the matrix:
    - If cell **(i, j)** is equal to **1**, then increment **countFresh**
    - If cell **(i, j)** is equal to **2**, a rotten orange, append that rotten orange **(i, j)** to the **queue**
- While the **queue** is not empty and **countFresh** is greater than 0:
    - Dequeue the front rotten orange **(i, j)** from the queue
    - For each cell **(neighbor_i, neighbor_j)** that is adjacent to **(i, j)** and which exists inside the matrix**:**
        - If **(neighbor_i, neighbor_j)** is equal to **1**, a fresh orange
            - Update that fresh orange to become rotten, **(neighbor_i, neighbor_j)** = **2**
            - Enqueue **(neighbor_i, neighbor_j)** onto the **queue**
            - Decrement **countFresh**
    - Increment **countTime**
- If there are still fresh oranges, return **-1**
- Else, return **countTime** , which represents the minimum time

----

Proof:

- Assume we get the input: Input: arr[][C] = { {2, 1, 0, 2, 1}, {1, 0, 1, 2, 1}, {1, 0, 0, 2, 1}};
- After going through every cell in the matrix, we find the amount of Fresh oranges and store it into **countFresh = 7**, and find amount of rotten oranges and store into the **queue**
- While our queue is not empty, which means that there are still rotten oranges in the queue, we haven't checked its neighbor oranges yet. And while there are still fresh oranges
    - At 0th time frame:
      {2, 1, 0, 2, 1}
      {1, 0, 1, 2, 1}
      {1, 0, 0, 2, 1}
    - We check the adjacent neighbor of that rotten orange and try to find if there are fresh oranges that we can infect and decrease the amount of countFresh.

- ○ Once we finish infecting the adjacent fresh oranges, we then increment the countTime and go to 1st time frame:
  {2, 2, 0, 2, 2}
  {2, 0, 2, 2, 2}
  {1, 0, 0, 2, 2}
- ○ Where we will repeat the same process of infecting fresh oranges again and increment the time again. Where then we will get to 2nd time frame:
  {2, 2, 0, 2, 2}
  {2, 0, 2, 2, 2}
  {2, 0, 0, 2, 2}
- ○ Since there are no more fresh oranges, we then terminate the while loop and return the time it took to rot all the oranges which is countTime = 2;
- ○ If when the while loop terminates and we still have a fresh orange, then we will return -1 since there is an orange that can't be rotted since it's not adjacent to any of the rotten oranges.

Time-Complexity:

- For the adjacency matrix, the algorithm above using BFS will take O(n * m) where n is the number of rows, and m is the number of columns, since we will be visiting each cell at least once