6. We have a connected graph $G = (V, E)$, and a specific vertex $u \in V$. Suppose we compute a depth-first search tree rooted at u, and obtain a tree T that includes all nodes of G. Suppose we then compute a breadth-first search tree rooted at u, and obtain the same tree T. Prove that $G = T$. (In other words, if T is both a depth-first search tree and a breadth-first search tree rooted at u, then G cannot contain any edges that do not belong to T.)

We need to prove that: To prove that if T is both a DFS tree and a BFS tree rooted at u, then G cannot contain any edges that do not belong to T.

By way of contradiction(BOC):
T is a BFS tree, so it contians the shortest paths form u to every other vertex in G. Now we say (v, w) is an edge in G but not in T and we know that t contains all the nodes in G. For there to be an edge in G but not in T, there must be a path from u to v and from u to w in T that does not use the edge(v,w). However, we know T is BFS tree and contains all the shortest path, which means (v, w)should included in T if it's the shortest path, and we reach the contradiction.

Time Complexity:
The time complexity of this DFS and BFS is O(V+E), which is the vertices and edges.
DFS explores every vertex once, and examines every edge in the graph in the process.
BFS traverses the graph layer by layer, visiting all nodes and checking all adjacent nodes (edges) from each node visited.

10. A number of art museums around the country have been featuring work by an artist named Mark Lombardi (1951–2000), consisting of a set of intricately rendered graphs. Building on a great deal of research, these graphs encode the relationships among people involved in major political scandals over the past several decades: the nodes correspond to participants, and each edge indicates some type of relationship between a pair of participants. And so, if you peer closely enough at the drawings, you can trace out ominous-looking paths from a high-ranking U.S. government official, to a former business partner, to a bank in Switzerland, to a shadowy arms dealer.

Such pictures form striking examples of *social networks*, which, as we discussed in Section 3.1, have nodes representing people and organizations, and edges representing relationships of various kinds. And the short paths that abound in these networks have attracted considerable attention recently, as people ponder what they mean. In the case of Mark Lombardi's graphs, they hint at the short set of steps that can carry you from the reputable to the disreputable.

Of course, a single, spurious short path between nodes v and w in such a network may be more coincidental than anything else; a large number of short paths between v and w can be much more convincing. So in addition to the problem of computing a single shortest v-w path in a graph G, social networks researchers have looked at the problem of determining the *number* of shortest v-w paths.

This turns out to be a problem that can be solved efficiently. Suppose we are given an undirected graph $G = (V, E)$, and we identify two nodes v and w in G. Give an algorithm that computes the number of shortest v-w paths in G. (The algorithm should not list all the paths; just the number suffices.) The running time of your algorithm should be $O(m + n)$ for a graph with n nodes and m edges.

The algorithm would be BFS. We know BFS contains the shortest path from nodes v to w and they would be at the same level. Therefore, we can mark that level and count the number of times we encounter w from v at that level.

Proof by contradiction that BFS contains shortest path:
Let's sya BFS counts a path from v to w that is longer than the shortest path. Since BFS goes one level at a time, and a node being encountered at a new level would be the shortest, w

would be encountered at the earliest level possible that it can be reached by any path from v. Therefore, longer path would not be in the BFS tree. We reached a contradiction.

Time Complexity Analysis:
o(m+n)becase:

1. Initialization: Setting up a queuefor all the nodes starting from v
2. Traversal: we process each node by dequeuing and enqueuing, and the time complexity is o(v).
3. Edge: We examine all the adjacent edges to explore connectivity and path counting. For the path counting in the context of this problem, BFS need to keep track of the number of shortest path form v to x. We'll keep a count of the number of paths and it takes contant time.

3. You are consulting for a trucking company that does a large amount of business shipping packages between New York and Boston. The volume is high enough that they have to send a number of trucks each day between the two locations. Trucks have a fixed limit W on the maximum amount of weight they are allowed to carry. Boxes arrive at the New York station one by one, and each package i has a weight w_i. The trucking station is quite small, so at most one truck can be at the station at any time. Company policy requires that boxes are shipped in the order they arrive; otherwise, a customer might get upset upon seeing a box that arrived after his make it to Boston faster. At the moment, the company is using a simple greedy algorithm for packing: they pack boxes in the order they arrive, and whenever the next box does not fit, they send the truck on its way.

But they wonder if they might be using too many trucks, and they want your opinion on whether the situation can be improved. Here is how they are thinking. Maybe one could decrease the number of trucks needed by sometimes sending off a truck that was less full, and in this way allow the next few trucks to be better packed.

Prove that, for a given set of boxes with specified weights, the greedy algorithm currently in use actually minimizes the number of trucks that are needed. Your proof should follow the type of analysis we used for the Interval Scheduling Problem: it should establish the optimality of this greedy packing algorithm by identifying a measure under which it "stays ahead" of all other solutions.

This is the current greedy algorithm:
The company uses a simple greedy algorithm to load the trucks:

- Pack boxes in the order they arrive: Each box is considered in sequence.
- Fill the truck until no more boxes fit: If the current box does not fit in the current truck due to weight constraints, the truck is sent off, and a new truck is started.
- Repeat until all boxes are shipped: This process repeats until all boxes are loaded onto trucks and sent.

Proof:

-The greedy algorithm is able to "stay ahead" by always packing the current truck to the maximum possible amount before starting a new truck. The algorithm would have used the smallest number of trucks because if there is another algorithm that uses fewer trucks, then this means that at some point, that algorithm would have packed more boxes into one of the earlier trucks than the greedy algorithm did, which contradicts to the fact that the greedy algorithm we have packs each truck to its fullest capacity before moving to the next truck.

-If the greedy algorithm minimizes the trucks for the first k trucks. By induction, we know that for the (k+1) th ruck it's gonna still be optimal, and this persists until we send out all the boxes.

Time Complexity Analysis:

- Iterating through all boxes: Process each box exactly once to decide whether it can be added to the current truck or if a new truck is needed. This step is linear with respect to the number of boxes, n
- Calculating the load for each truck: Each time we add a box to a truck, we update the current truck's load by adding the weight of the box. This operation is constant time, $O(1)$ for each box.

- Managing trucks: The operation of deciding whether to send a truck off and start a new one (including updating the total number of trucks used) also takes constant time, $O(1)$, each time it happens.

Therefore, the total time complexity is O(n). (We ignore the constant times)

6. Your friend is working as a camp counselor, and he is in charge of organizing activities for a set of junior-high-school-age campers. One of his plans is the following mini-triathalon exercise: each contestant must swim 20 laps of a pool, then bike 10 miles, then run 3 miles. The plan is to send the contestants out in a staggered fashion, via the following rule: the contestants must use the pool one at a time. In other words, first one contestant swims the 20 laps, gets out, and starts biking. As soon as this first person is out of the pool, a second contestant begins swimming the 20 laps; as soon as he or she is out and starts biking, a third contestant begins swimming . . . and so on.)

Each contestant has a projected *swimming time* (the expected time it will take him or her to complete the 20 laps), a projected *biking time* (the expected time it will take him or her to complete the 10 miles of bicycling), and a projected *running time* (the time it will take him or her to complete the 3 miles of running). Your friend wants to decide on a *schedule* for the triathalon: an order in which to sequence the starts of the contestants. Let's say that the *completion time* of a schedule is the earliest time at which all contestants will be finished with all three legs of the triathalon, assuming they each spend exactly their projected swimming, biking, and running times on the three parts. (Again, note that participants can bike and run simultaneously, but at most one person can be in the pool at any time.) What's the best order for sending people out, if one wants the whole competition to be over as early as possible? More precisely, give an efficient algorithm that produces a schedule whose completion time is as small as possible.

Algorithm:
Since only one person can be at the swimming pool at any time, the overall time contestants spend on swimming should be a constant no matter what the order is. We then know that the time we need to compare is the bike+run time.
We will use the greedy algorithm to decide the schedule that minimize the completion time for the triathlon: generate a list of bike+run time of contestants and sort them in ascending order, and they will use the swimming pool one at a time using in this order. By minimizing the bike+running sum, we ensure that the faster cyclists/runners start their cycling and running phases sooner, which maximizes the overlap of activities and minimizes idle time.

Proof by Contradiction:
Assume that there is a better schedule than greedy algorithm where the another contestant 1 who has a loger run+bike time goes before contestant 2 who has a shorter run+bike time. If so and we let 1 goes to the swimming pool before 2, it will take a longer time for the whole game since contestant 2 could have finished the biking+swimming faster and minimize the overall game time.

Time Complexity Analysis:

1. Computing the bike+running time: To compute the sum for n contestants takes O(n) time.
2. Sorting: Soring the contestants based on the bike+running time using heap sort takes o(n log n) time.
3. Scheduling: For scheduling the contestant to start going to the swimming pool we need to go through everyone on the list and it takes o(n) time if we have n contestants.

Therefore, the overall time complexity is calculated by o(n) + o(n log n) + o(n) = o(n log n).

12. Suppose you have n video streams that need to be sent, one after another, over a communication link. Stream i consists of a total of b_i bits that need to be sent, at a constant rate, over a period of t_i seconds. You cannot send two streams at the same time, so you need to determine a *schedule* for the streams: an order in which to send them. Whichever order you choose, there cannot be any delays between the end of one stream and the start of the next. Suppose your schedule starts at time 0 (and therefore ends at time $\sum_{i=1}^{n} t_i$, whichever order you choose). We assume that all the values b_i and t_i are positive integers.

Now, because you're just one user, the link does not want you taking up too much bandwidth, so it imposes the following constraint, using a fixed parameter r:

> *(*) For each natural number $t > 0$, the total number of bits you send over the time interval from 0 to t cannot exceed rt.*

Note that this constraint is only imposed for time intervals that start at 0, *not* for time intervals that start at any other value.

We say that a schedule is *valid* if it satisfies the constraint (*) imposed by the link.

The Problem. Given a set of n streams, each specified by its number of bits b_i and its time duration t_i, as well as the link parameter r, determine whether there exists a valid schedule.

Example. Suppose we have $n = 3$ streams, with

$$(b_1, t_1) = (2000, 1), \quad (b_2, t_2) = (6000, 2), \quad (b_3, t_3) = (2000, 1),$$

and suppose the link's parameter is $r = 5000$. Then the schedule that runs the streams in the order 1, 2, 3, is valid, since the constraint (*) is satisfied:

$t = 1$*: the whole first stream has been sent, and* $2000 < 5000 \cdot 1$
$t = 2$*: half of the second stream has also been sent,*
and $2000 + 3000 < 5000 \cdot 2$
Similar calculations hold for $t = 3$ *and* $t = 4$.

(a) Consider the following claim:

> Claim: There exists a valid schedule if and only if each stream i satisfies $b_i \leq rt_i$.

Decide whether you think the claim is true or false, and give a proof of either the claim or its negation.

(b) Give an algorithm that takes a set of n streams, each specified by its number of bits b_i and its time duration t_i, as well as the link parameter r, and determines whether there exists a valid schedule. The running time of your algorithm should be polynomial in n.

This claim is false. Imagine we have two streams(2,1) and (1, 1000) and r =1, stream(2,1) does not satisfy the condition of the claim that there exists a valid schedule if and only if each stream i satisfies b_i <= rt_1. However, if we order the second stream before the first one, the schedule would be valid since the constraint is: for each natural number t > 0, the total number of bits you send over the time interval from 0 to t cannot exceed rt. The total number of bits in this case does not go over rt.

1. Streams are sorted in ascending order according to their b_i/t_i ratio. A lower ratio means the stream uses less bandwidth per unit of time.
   (By scheduling these streams first, we utilize the available bandwidth more efficiently. It helps prevent early bandwidth peaks that could violate the rt constraint. If a ratio one is scheduled too early, it could consume a large portion of the bandwidth, making it harder to dit subsequent streams without exceeding the allowed rate.)
2. Create the variable B_t and t. B_t represents the total bits that have already been sent till time t. And t is the current time.
3. For each stream i in sorted order, we need to consider if it fulfills the constraint. If B_t + b_i < r * (t +t i), then we can add stream i to the schedule at time t. After the comparison, we need to update the value of B_t and t so that it can iterate to the next stream on the list.
4. After the iteration, if the set of streams n are all added to the schedule, then we can claim that we have a valid schedule. Else there is no valid schedule for the set of streams.

Proof:
Inductive Basis: B_t = 0 and t = 0, and we meet the constraint that B_t < r*t.
Inductive Step: Assume for streams 1 through k-1, the schedule is valid. Since the algorithm checks if B_t + b_k <= r * (t+ t_k) before adding each stream, then by induction it means to total bits sent will never exceed the allowed r * (t + t_k).

Time Complexity:
1. Sorting the streams using heap takes O(n log n) time.
2. Looping through the list of streams takes O(n) time.
Therefore, the overall time complexity is O(n long n) time.

Problem 6

You are given a matrix of dimension $M \times N$, where each cell in the matrix is initialized with values 0, 1, 2, or 3 which has the following meaning:

- 0: This cell contains no orange.
- 1: Cell has a fresh orange.
- 2: Cell has a dirty orange.
- 3: Cell has a rotten orange.

The input matrix represents the state of all cells at day 0. The task is to find the first day all oranges become rotten. A dirty or rotten orange at index (i, j) will make all neighboring fresh oranges dirty by the next day. The neighbors of any cell are the cells directly above, below, to the left, and to the right (assuming the neighbor is in the bounds of the matrix). All dirty oranges will become rotten by the next day. If it is impossible to rot every orange then simply return -1.

We will use BFS to solve this problem.

1. Identify all initially rotten and dirty oranges and place them into a queue as starting points for the BFS. Count fresh oranges to know when all are rotten or dirty.
2. BFS. Process each orange in the queue. For both rotten and dirty oranges in the queue, check their neighbors.
   If a neighbor is a fresh orange, we make it dirty. Enqueue the orange with an incremented day count.
   Also if an orange is dirty on a given day, it should become rotten the next day.
3. If the fresh orange we recorded becomes 0 then we return -1.

Proof:

Inductive Base:

- The algorithm begins by enqueuing all initially rotten and dirty oranges. Counting fresh oranges allows us to track when the process can be considered complete or determine if it's impossible to rot all oranges.

Inductive Step:

- For each rotten or dirty orange, the algorithm correctly simulates the spreading of rot or dirt, since a dirty or rotten orange at index (i, j) will make all neighboring fresh oranges dirty by the next day.

By induction, all the oranges should be rotten by the end of the algorithm.

.

Time Complexity:
It’s O(r*c) because in the worst case, we need to visit all cells in the grid (to discover all fresh oranges can be made rotten) and each cell will be examined at least once.