CS 180
Algorithms and complexity:
Homework 3

Question 1)

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.

Set of intricately rendered graphs. Graphs encode the relationship among people involved in major political scandals. Nodes correspond to the participants, edge indicates some type of relationship between a pair of participants.

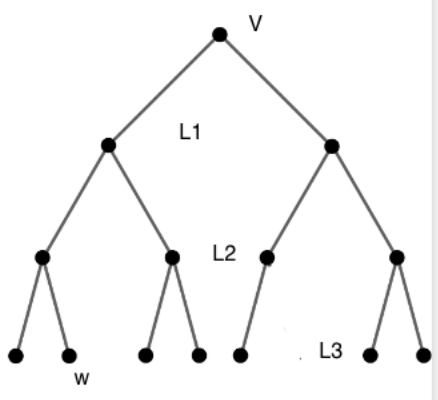
Social networks researchers have looked at the problem of determining the *number* of shortest *v-w* paths.
Give an algorithm that computes the number of shortest v-w paths of an Undirected graph.

For this, we would use a breadth first search algorithm
Steps of the algorithm:

1. Start from the source node *v*. Set counters for all nodes to zero and mark them as unvisited, except the source node which would have counter as one
2. **Breadth-First Search:** Begin a BFS from the source node *v*. For every node discovered at a particular depth, check its neighbors (only check undiscovered nodes). Increment the counter by adding the parent counter to the child counter. If a node has multiple parents, count should be the sum of the parents' counters.
3. **Discovered Node:** Once the entire depth has been explored and all counters have been incremented, mark the nodes at that level as discovered
4. At the end of each BFS level (i.e., after exploring nodes at a particular depth), check if the destination node *w* has been discovered:
    - If *w* is found, increment the main counter by the count of the node *w*. Do not terminate the BFS until all nodes at this depth are explored to ensure all shortest paths are considered.
    - If the BFS completes a level without discovering *w*, continue the BFS to the next depth.
5. Once the BFS level where *w* was first discovered is fully explored, terminate the BFS. The sum of counters that discovered the node *w* at its first depth of discovery is the total number of shortest

paths from *v* to *w*. If required, the shortest path length can be indicated by the BFS depth at which *w* was first discovered. However, if w is not found and all nodes are discovered, return 0

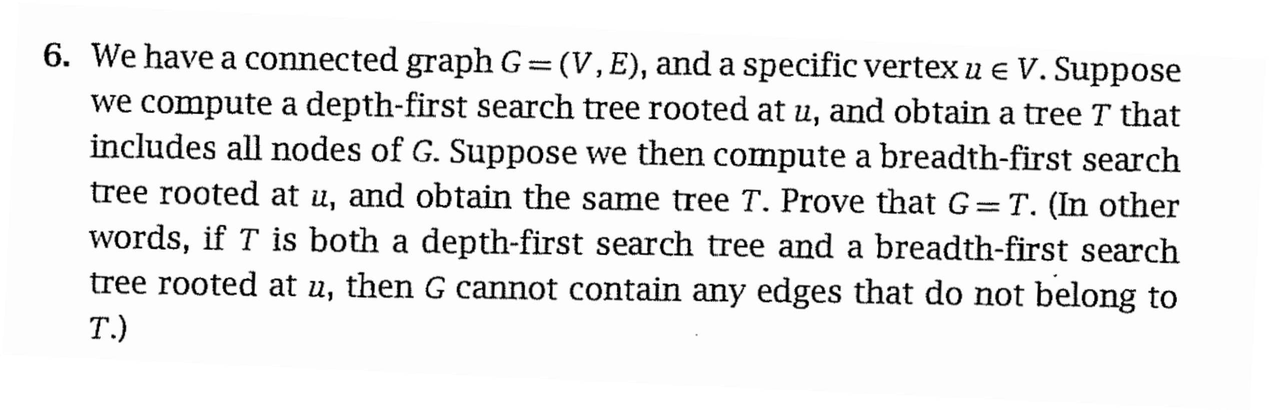
Different scenarios to prove the algorithm:
We consider two scenarios
When the graph is structured as a tree, with v as the root node and v at the end.



The graph first explore L1, then explores L2. As the count is still zero(w not found), the algorithm continues till L3. Once it finds the node w, it returns the count which would be 1 in this scenario. If more L2 nodes were connected to w, it would return the number of paths that were of length 3 that lead to that node.
If different paths lead to the same node, the algorithm would take note of that using the counter variable.

According to this, the time complexity of the graph would be (m+n) n nodes and m edges(in the worst case scenario), as we aren't repeating any nodes or edges(if they do merge, one of the searches gets deleted though the number of paths leading to that is stored.

Problem 2)



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 have a connected graph G = (V,E). We need to prove that that the trees obtained from both the algorithms are the same. We will prove this using contradiction. According to the definition of a connected graph: Every node is accessible from every other node. We prove this using contradiction

Contradiction Assumption:
- Assume there's an edge (x, y) in G which is not in T.

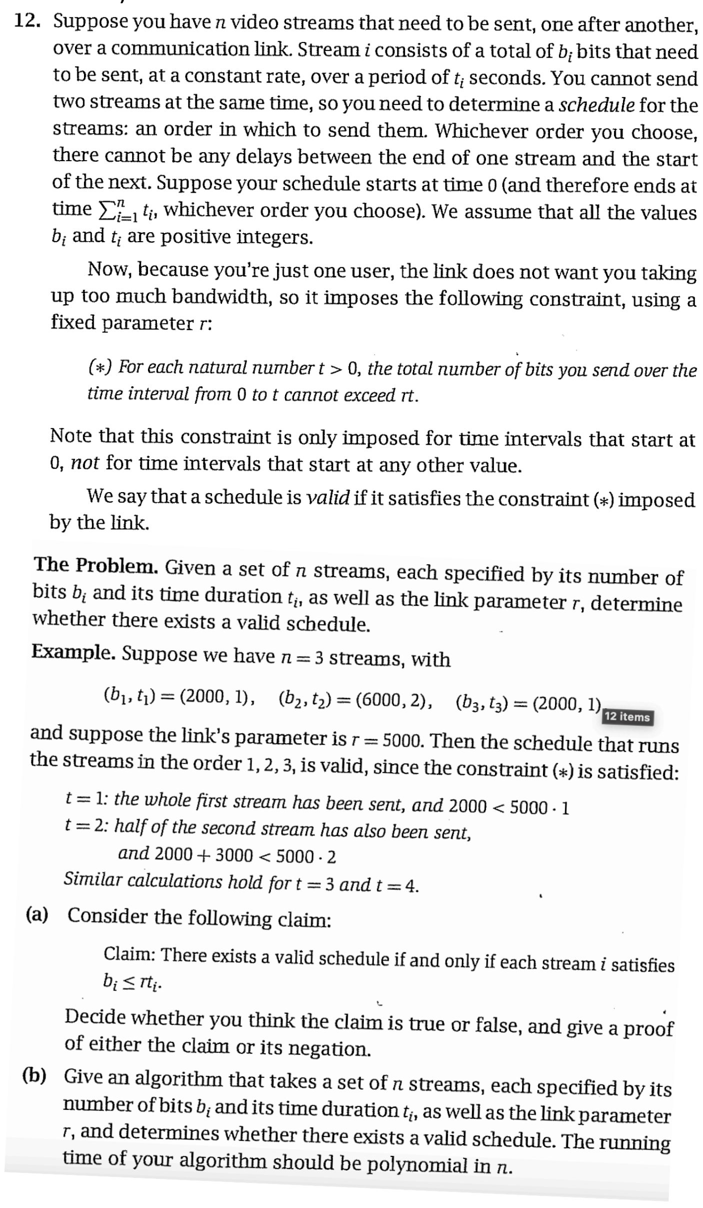
DFS : For DFS starting from vertex u, either x is visited before y, or y is visited before x. Let's assume x was visited first.
- If (x, y) is in G and x was visited before y in DFS, then y should be a child of x in T. But since (x, y) is not in T, this is a contradiction.

BFS: Vertices are visited level by level. If x and y are on consecutive levels in T and (x, y) is in G, then (x, y) must be in T. This is because BFS would discover y immediately after x. This contradicts our assumption that (x, y) is not in T.

- If x and y are not on consecutive levels in T, then the edge (x, y) would contradict the nature of

BFS, as it would imply a shorter path which BFS should have discovered first.
Given these contradictions in both DFS and BFS arguments, our initial assumption that there exists an edge (x, y) in G which is not in T is wrong.
Therefore, all edges of G are also in T, meaning G = T.

Problem 3)



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 $b_i \le rt_i$.

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.

Need to send n video streams one after the other. Each stream i consists of bi bits sent consistently over a period of ti seconds. No overlap. Also, there should be no delays in between of sending two streams. The total number of bits you send over the time interval from 0 to t can't exceed rt(where r is bandwidth param)

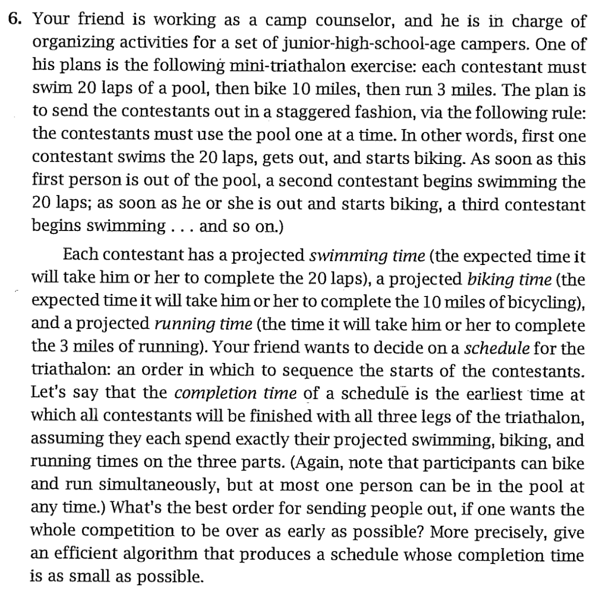
A stream is given in the following format: (bi,ti), where bi can't exceed more than rti. Using a basic induction we can prove that if the previous schedule followed the rules of the byte stream, and the next data set also follows the rules, then the entire schedule follows the bandwidth rule

a) This claim is False, we would prove this using contradiction. Assume the following scenario:
stream 2 = (6000, 1).   Stream 1 = (2000, 1), r= 5000
If placed in the respective orders, this would be an invalid schedule as the first byte stream would fail the requirement (6000 is not less than 1*5000) However, if placed in the reverse order , this would be a valid schedule (as 2000 <= 5000*1, and 8000 <= 5000*2)
Th
We would prove this using mathematical induction. Let the first case be an edge case, where the first byte stream scheduled to run uses the entire bandwidth (5000 bits per second). Thus, the schedule (5000, 1) satisfies the condition.

b) To determine whether the schedule is valid or not, we just have to identify whether the ratio of all bytes sent over n streams over the total time is less than or equal to the bandwidth.
To make the best schedule(which would optimize the amount of valid schedules we can make, we have to sort the streams in terms of shortest byte speed first(in ascending order). This is because the lesser number of byte speed we use in the beginning, the more byte speed we could use for other streams(considering there are streams where the byte speed is greater than the r parameter).
If the ratio of total bytes sent to the total number of time is greater than the ratio r, then there would exist no valid schedule: For example:
Stream 1 = (2000, 1), Stream 2 = 6001,1), where r = 4000
In any ordering, the ratio at the end would be 8001/2, which exceeds 4000, and thus there is no valid schedule that could be formed

For this algorithm, the Big O time complexity is O(n) as we are sampling through n samples linearly, going over the array of byte streams once.

Problem 5)



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.

We want to minimize the overall completion time of the triathlon. The constraint for this problem is that that only one person can use the pool at a time. After swimming, there is no constraint for biking and running.

Since contestants can bike and run simultaneously, the time taken for biking and running does not affect the order in which contestants should start. The order is mainly determined by the swimming time because the swimming pool can be used by only one person at a time.
Strategy: Send the contestant with the shortest swimming time first, followed by the next shortest swimming time, and so on. The rationale is that by allowing the quickest swimmers to start first, we will free up the pool faster for the next contestant.
Algorithm steps:

- Sort the contestants based on their swimming time in ascending order.
- Send the contestants out in the sorted order.
- Create a list of contestants with their swimming, biking, and running times.
- Sort the list based on swimming times.
- for i from 0 to n-1 (where n is the number of contestants), Send out contestant[i] for the triathlon.

Sample scenario explanation: Suppose we have 2 contestants A and B. If A swims faster than B, then by sending A first, we ensure that B starts swimming as soon as possible. This is because the constraint is only on swimming. Once they are out of the pool, they can bike and run simultaneously, so their biking and running times don't affect the overall order of the start. By extending this logic to more contestants, it makes sense to sort them based on swimming times. Therefore, the overall completion time of the competition is minimized when contestants are sent out in ascending order of their swimming times.

As we use a sorting algorithm to sort them based on their swimming speed, the time complexity arises from the sorting of the participants in terms of swimming speed. The fastest sorting algorithm(bubble sort) would be able to achieve a complexity of O(n log n), which is the complexity of the algorithm.

Problem 4)

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 *wi*. 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.

For this problem, we need to prove that the greedy algorithm in use minimizes the number of trucks that are needed.
What the greedy algo states:

- Packages are filled into a truck on a First come first serve basis, until the capacity of the truck is met. Once the truck reaches its weight limit W, the next package leaves with the next truck.

- We will prove this using Principle of mathematical induction:

1. If the weight of the n+1th package is such that it can fit in the current truck being filled by the greedy algorithm, it will be added to that truck.
2. If it doesn't fit, the greedy algorithm will start a new truck for the n+1th package.
3. In either scenario, given our inductive hypothesis, we know that the solution up to the *n*th package was optimal. For the *n*+1th package, the greedy algorithm makes the best decision given the current situation.
4. Therefore, the solution for *n*+1 packages will also be optimal. All the other trucks that were full in the n packages solution are optimally packed, so there is no need to move out a package and swap them.
5. If all the previous trucks are filled up to their limit, and a new package k+1 comes in, there would have to be another truck assigned to it.

Thus, this algorithm would produce the most optimal trucking schedule with the least amount of trucks being used.

Problem 6)

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}
For this question, 0 represents no oranges, 1 represents a fresh orange, and 2 represents a rotten orange.

The algorithm for this is:

1) Find the nodes that are labelled 2 and label them as rotten. Do a search an set the fresh_orange_counter to the number of 1's in the array
2) Run a BSF algorithm on each of those nodes labelled as one, which checks the neighbors of that node if they exist(let I,j be coordinates. We would be checking (i-1,j) ; (i+1, j); (i,j-1) , (i,j+1). We go one level at a time for each of the nodes.
3) If you encounter a 1, instantly change that to a 2. For each of the nodes, check if there is a single node with a one. If there is none, terminate the BSF algorithm at that depth. If there are, decrement fresh_orange_counter by the number of 1s found around that node. If encountered a zero, ignore and don't change it.
4) At the end of the algorithm, check if fresh_orange_counter if zero. If yes, return the maximum depth of reached with the BSF algorithm. If not, return negative 1.

This algorithm would successfully find the shortest time it would take for the rotten oranges to infect all the fresh oranges, and would also return -1 if there are fresh oranges that are not in direct contact with any other oranges.
The total time complexity of this algorithm is primarily determined by the Breadth-First Search (BFS) traversal of the entire matrix.

For a matrix of size **m x n**, in the worst case, you might have to traverse all the cells of the matrix. Thus, the time complexity is O(mn).

Additionally, for each node, we’re checking its neighbors, which is a constant-time operation since each cell has at most 4 neighbors. Thus, checking the neighbors doesn't add to the big O.

Hence the overall time complexity of the algorithm is O(mn). The space complexity depends on the size of the BFS queue, which in the worst case can be O(mn) as well, if all cells in the matrix were to be added to the queue. Therefore, the space complexity is also O(mn).