- The instructions are the same as in Homework 1 and 2.

There are 6 questions for a total of 82 points.

---

1. (10 points) Given a strongly connected directed graph, $G = (V, E)$ with positive edge weights along with a particular node $v \in V$. You wish to pre-process the graph so that queries of the form "*what is the length of the shortest path from s to t that goes through v*" can be answered in constant time for any pair of distinct vertices s and t. Your pre-processing algorithm should take the same asymptotic run-time as Dijkstra's algorithm. Analyze the runtime and provide proof of correctness.

2. Counterexamples are effective in ruling out certain algorithmic ideas. In this problem, we will see a few such cases.

   (a) (3 points) Recall the following event scheduling problem discussed in class:

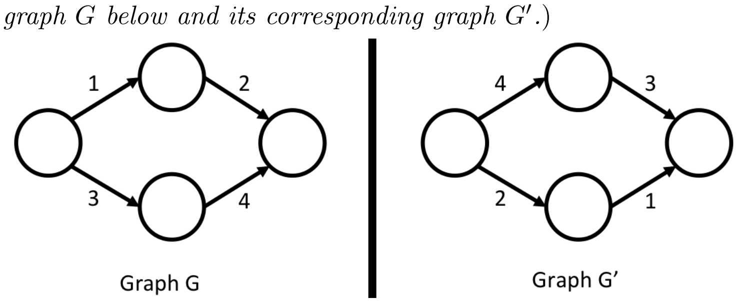
   > You have a conference to plan with n events and an unlimited supply of rooms. Design an algorithm to assign events to rooms in such a way as to minimise the number of rooms.

   The following algorithm was suggested during class discussion.

   `ReduceToSingleRoom`($E_1, ..., E_n$)
   - $U \leftarrow \{E_1, ..., E_n\}$; $i \leftarrow 1$
   - While U is not empty:
     - Use Earliest Finish Time greedy algorithm on events in set U to schedule a subset $T \subseteq U$ of events in room i
     - $i \leftarrow i + 1$; $U \leftarrow U \setminus T$

   Show that the above algorithm does not always return an optimal solution.

   (b) (3 points) A longest simple path from a node s to t in a weighted, directed graph is a simple path from s to t such that the sum of weights of edges in the path is maximised. Here is an idea for finding a longest path from a given node s to t in any weighted, directed graph $G = (V, E)$:

   > Let the weight of the edge $e \in E$ be denoted by $w(e)$ and let w_{max} be the weight of the maximum weight edge in G. Let G' be a graph that has the same vertices and edges as G but for every edge $e \in E$, the weight of the edge is $(w_{max} + 1 - w(e))$. (*For example, consider the graph G below and its corresponding graph G'.*)
   >
   > 
   >
   >
   > Run Dijkstra's algorithm on G' with starting vertex s and return the shortest path from s to t.

   Show that the above algorithm does not necessarily output the longest simple path.

(c) (3 points) Recall that a *Spanning Tree* of a given connected, weighted, undirected graph $G = (V, E)$ is a graph $G' = (V, E')$ with $E' \subseteq E$ such that G' is a tree. The cost of a spanning tree is defined to be the sum of the weight of its edges. A *Minimum Spanning Tree (MST)* of a given connected, weighted, undirected graph is a spanning tree with minimum cost. The following idea was suggested for finding an MST for a given graph in the class.

> Dijkstra's algorithm gives a shortest path tree rooted at a starting node s. Note that a shortest path tree is also a spanning tree. So, simply use Dijkstra's algorithm and return the shortest path tree.

Show that the above algorithm does not necessarily output an MST. In other words, a shortest path tree may not necessarily be an MST. (*For this question, you may consider only graphs with positive edge weights.*)

3. (*Example for "greedy stays ahead"*) Suppose you are placing sensors on a one-dimensional road. You have identified n possible locations for sensors, at distances $d_1 \leq d_2 \leq ... \leq d_n$ from the start of the road, with $0 \leq d_1 \leq M$ and $d_{i+1} - d_i \leq M$. You must place a sensor within M of the start of the road and place each sensor after that within M of the previous one. The last sensor must be within M of d_n. Given that, you want to minimize the number of sensors used. The following greedy algorithm, which places each sensor as far as possible from the previous one, will return a list $d_{i_1} \leq d_{i_2} \leq ... \leq d_{i_k}$ of locations where sensors can be placed.

`GreedySensorMin(`$d_1 ... d_n, M$`)`
- Initialize an empty list
- Initialize $I = 1$, $PreviousSensor = 0$.
- While $(I < n)$:
    - While ($I < n$ and $d_{I+1} \leq PreviousSensor + M$) $I++$
    - If $(I < n)$ Append d_I to list; $PreviousSensor = d_I$;$I++$.
- if list is empty, append d_n to list
- return(list)

In using the "greedy stays ahead" proof technique to show that this is optimal, we would compare the greedy solution $d_{g_1}, ..d_{g_k}$ to another solution, $d_{j_1}, ..., d_{j_{k'}}$. We will show that the greedy solution "stays ahead" of the other solution at each step in the following sense:
<u>Claim</u>: For all $t \geq 1, g_t \geq j_t$.

(a) (5 points) Prove the above claim using induction on step t. Show base case and induction step.

(b) (3 points) Use the claim to argue that $k' \geq k$. (Note that this completes the proof of optimality of the greedy algorithm since it shows that the greedy algorithm places at most as many sensors as any other solution.)

(c) (2 points) In big-O notation, how much time does the algorithm, as written, take? Write a brief explanation.

4. (*Example for "modify the solution"*) You have n cell phone customers who live along a straight highway at distances $D[1] < D[2] < ... < D[n]$ from the starting point. You need to have a cell tower within Δ distance of each customer, and want to minimize the number of cell towers.

(*For example, consider* $\Delta = 3$ *and there are* 3 *customers (i.e.,* $n = 3$*) with* $D[1] = 3, D[2] = 7, D[3] = 10$. In this case, you can set up two cell towers, one at 6 and one at 10.)

Here is a greedy strategy for this problem.

> **Greedy strategy**: Set up a tower at a distance d at the farthest edge of the connectivity range for the customer closest to the starting point. That is, $d = \Delta + D[1]$. Note that all customers within Δ distance of this tower at d are covered by this tower. Then, recursively set up towers for the remaining customers (who are not covered by the first tower).

We will show that the above greedy strategy gives an optimal solution using modify-the-solution. For this, we will first need to prove the following exchange lemma.

Exchange Lemma: *Let G denote the greedy solution and let g_1 be the location of the first cell phone tower set up by the greedy algorithm. Let OS represent any solution that does not have a cell phone tower at g_1. Then there exists a solution OS' that has a cell phone tower set up at g_1, and OS' has the same number of towers as OS.*

Proof. Let $OS = \{o_1, ..., o_k\}$. That is, the locations of the cell phone towers as per solution OS is $o_1 < o_2 < ... < o_k$. We ask you to complete the proof of the exchange lemma below.

(a) (1 point) Define OS'.

(b) (2 points) OS' is a valid solution because ... (*justify why OS' provides coverage to all customers.*)

(c) (2 points) The number of cell phone towers in OS' is at most the number of cell phone towers in OS because... (*justify*)

We will now use the above exchange lemma to argue that the greedy algorithm outputs an optimal solution for any input instance. We will show this using mathematical induction on the input size (i.e., the number of customers). The base case for the argument is trivial since, for $n = 1$, the greedy algorithm opens a single tower, which is indeed optimal.

(a) (3 points) Show the inductive step of the argument.

Having proved the correctness, we now need to give an efficient implementation of the greedy strategy and give time analysis.

(a) (5 points) Give an efficient algorithm implementing the above strategy and give a time analysis for your algorithm.

5. (20 points) A town has n residents labelled $1, ..., n$. Amid a virus outbreak, the town authorities realize that hand sanitizer has become an essential commodity. They know that every resident in this town requires at least T integer units of hand sanitizer. However, at least $\lceil \frac{n}{2} \rceil$ residents do not have enough sanitizer. On the other hand, there may be residents who may have at least T units. They want to implement a sharing strategy with very few new supplies coming in for the next few weeks. At the same time, they do not want too many people to get close to each other to implement sharing. So, they come up with the following idea:

   > Try to *pair up* residents (based on the amount of sanitizer they possess) such that:
   >
   > 1. A resident is either unpaired or paired with exactly one other resident.
   > 2. Residents in a pair together should possess at least $2T$ units of sanitizer.
   > 3. The number of unpaired residents with less than T units of sanitizer is minimized.

   Once such a pairing is obtained, the unpaired residents with less than T units of sanitizer can be managed separately. The town authorities have conducted a survey and know the amount of sanitizer every resident possesses. You are asked to design an algorithm for this problem. You are given as input integer n, integer T, and integer array $P[1...n]$ where $P[i]$ is the number of units of sanitizer that resident i possesses. You may assume that $0 \leq P[1] \leq P[2] \leq ... \leq P[n]$. Your algorithm should output a pairing as a list of tuples $(i_1, j_1), (i_2, j_2), ..., (i_k, j_k)$ of maximum size such that (i) For all $t = 1, ..., k$, $P[i_t] + P[j_t] \geq 2T$ and (ii) $i_1, ..., i_k, j_1, ..., j_k$ are distinct. Give proof of correctness of your algorithm and discuss running time.

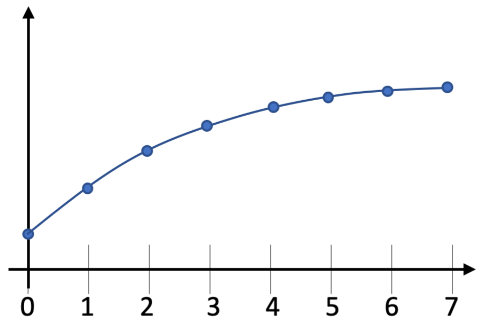


Figure 1: The shape of the curve for any R_i that follows properties (A) and (B) will be as in the figure above. This follows the *law of diminishing returns*. Even though the reward increases with every added hour, the extra reward for an additional hour keeps decreasing.

6. (20 points) (*Time splitting*): You want to divide H hours between k given tasks. That is, assign $h[i]$ hours to task i such that for all i, $h[i]$ is an integer and $\sum_{i=1}^{k} h[i] = H$. The reward that you get by spending $h[i]$ hours on task i is given by $R_i(h[i])$. These reward functions $R_1(.), R_2(.), ..., R_k(.)$ satisfy the following properties (also see Figure 1):

   (A) For all i, $R_i(0) \leq R_i(1) \leq R_i(2) \leq ... \leq R_i(H)$.

   (B) For all i, $R_i(1) - R_i(0) \geq R_i(2) - R_i(1) \geq R_i(3) - R_i(2) \geq ... \geq R_i(H) - R_i(H-1)$.

   Given integers H, k, and such reward functions $R_1, R_2, ..., R_k$, design a greedy algorithm that outputs the time division $h[1], h[2]..., h[k]$ that maximises the total reward $\sum_{i=1}^{k} R_i(h[i])$. Give proof of correctness and discuss running time (as a function of H and k).