The answers must be word-processed or typed. You may substitute formulae and figures with hand-writings. Your submitted algorithms should be in the pseudo-code, not in any specific programming language. Answers deviating from these requirements will be returned without grading.

The answers must be the student’s own work. Idea sharing and referencing to others’ work are not allowed. Plagiarism and other forms of academic dishonesty will be handled within the guidelines of the Student Handbook and reported to the University.

There are 150 points in total.

1. **(15 points)** Let $G = (V, E)$ be a non-directed, edge-weighted graph in which edge weights are all distinct. Assume that edge $(u, v)$ has the second smallest weight. Use the “swapping” technique to prove that every minimum spanning tree contains edge $(u, v)$.

2. **(20 points)** Consider the following problem to update a found minimum spanning tree. Let $G = (V, E)$ be a given non-directed, edge-weighted graph and $T$ be a given minimum spanning tree $T$ for $G$. Assume that an edge $(u, v)$ in $G$ has its weight changed from $w(u, v)$ to $w'$. Edge $(u, v)$ may or may not be in $T$.

Describe algorithm $\text{UpdateMST}(T, T', w', u, v)$ that produces a minimum spanning tree $T'$ from $T$ (i.e., modify $T$ if you need to) based on the weight change of edge $(u, v)$.

**Note:** Your algorithm $\text{UpdateMST}(T, T', w', u, v)$ needs to be written in English + mathematical languages. You will not get credit if it is written in any programming language.

3. **(15 points)** Consider algorithm $\text{Bellman-Ford}$ on the directed graph in Figure 1, using vertex 4 as the source.
(1) Show $d$ and $\pi$ values after each of the $|V| - 1$ passes.

(2) Modify the graph so that it contains at least one negative cycle. Run the Bellman-Ford algorithm again to show how the negative cycle can be detected by the algorithm.

Figure 1: Graph for Question 3.

4. (100 points) This question asks you to design and implement a Google Map-like program using the learned shortest path algorithms.

**The map:** the map is a non-directed graph $G = (V, E)$. Each vertex $v \in V$ represents a traffic intersection and an edge $(u, v) \in E$ represents the two-way street (or road) from interactions $u$ to $v$. The weight on every edge $(u, v)$, denoted with $\tau(u, v)$, is the time to travel between $u$ and $v$. You may think of the map looks like Figure 2 (and perhaps much larger).

Figure 2: Illustration map for Question 4.
The problem: Without doubt, shortest path algorithms can be used to find a “static” shortest path from starting position $s$ to destination $t$. But your Google Map-like algorithm should be able to take real-time traffic information into consideration and to dynamically choose a shortest path from the current position $p$ to the destination $t$. That is, at the current position $p$, the algorithm checks if a travel time change on the street $(u,v)$ and makes an adjustment to the current path.

Requirement 1 (25 points): Describe (in English + mathematical languages) your algorithm that can make an adjustment to the shortest path $p \leadsto t$ when the travel time on street $(u,v)$ increases from $\tau(u,v)$ to $\tau'(u,v)$, i.e., $\tau(u,v) < \tau'(u,v)$. Here $(u,v)$ is assumed to be on the path $p \leadsto t$. You need to assess the time complexity for the adjustment step.

Requirement 2 (25 points): Extended from Requirement 1, now $(u,v)$ may not be on the current path $p \leadsto t$ and travel time $(u,v)$ may also decrease, i.e., it is also possible $\tau(u,v) > \tau'(u,v)$. Your algorithm should consider this situation in adjusting the shortest path $p \leadsto t$. You need to assess the time complexity for the adjustment step.

** You should write algorithms for requirements 1 and 2 separately.

Requirement 3 (50 points): Implement your algorithm in a programming language. The input to the program should be a map as described, including a starting position $s$ and a destination position $t$. The program should accomplish the following tasks.

It consists of a loop such that at each iteration of the loop (which can be control manually or by a system timer), (1) the vehicle moves from interaction $p$ to the next interaction $q$ on the identified shortest path $p \leadsto t$, (2) at position $q$, it checks for travel time changes on streets, and (3) adjust path $q \leadsto t$ based on the time changes.

The time change information can be available by calling a subroutine that randomly identifies one or more streets $(u,v)$ on the map and randomly changes the travel time of these streets. The loop repeats until the vehicle arrives at the destination $t$. The program should output the shortest path at every iteration together with the relevant travel time change information.

More realistic situations to consider but not required: While the above situation assumes that travel time changes on streets are synchronized with the time the vehicle arrives at an intersection, in
reality, the algorithm should checks periodically traffic situations while on the street as well as well just at intersections. We may assume the period to be some time unit and travel time is measured with such unit. So the subroutine used to issue travel time change should also be synchronized with the time period. For example, if the travel time from interaction $p$ to intersection $q$ is $\tau(p, q) = 10$ units, the program should check traffic and adjust path 10 times on the street $(p, q)$. To deal with these situations, an iteration of the loop in your program should correspond to one time unit.